М.П. Левин

# Параллельное программирование с использованием OpenMP

ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

# Параллельное программирование с использованием OpenMP

2-е издание, исправленное

Левин М.П.

Национальный Открытый Университет "ИНТУИТ"
2016

УДК [004.272:004.42](07)
ББК 15
Л36
Параллельное программирование с использованием OpenMP / Левин М.П. - М.: Национальный Открытый Университет "ИНТУИТ", 2016 (Основы информационных технологий)
ISBN 978-5-94774-857-4

В курсе изложены сведения о параллельном программировании с использованием OpenMP для современных параллельных высокопроизводительных вычислительных систем с общей памятью.

Курс может быть использован в процессе подготовки и переподготовки специалистов в области информационных технологий и программирования, а также в процессе обучения студентов и аспирантов высших учебных заведений по аналогичным направлениям.

# Введение

В настоящей лекции будут кратко рассмотрены основные типы современных параллельных вычислительных систем и средства параллельного программирования для них.

# Классификация параллельных архитектур

Подробный анализ и обзор современных архитектур параллельных вычислительных систем можно найти в книгах [1.1-1.2]. В этой лекции рассматриваются лишь те особенности архитектур, которые необходимо учитывать при создании параллельных программ. Отметим, что эти особенности в значительной мере определяют инструментарий параллельного программирования. Под инструментарием параллельного программирования понимается набор алгоритмических языков (в том числе и расширения стандартных алгоритмических языков), использующихся для написания параллельных программ, а также специализированные библиотеки функций, адаптированные к архитектуре конкретной параллельной вычислительной системы. Кроме того, к инструментарию параллельного программирования относятся средства отладки и оптимизации быстродействия параллельных программ, а также средства визуализации и представления результатов параллельных вычислений. В настоящее время различные производители системного программного обеспечения предлагают множество разнообразных инструментов, которые могут быть использованы в процессе создания параллельных программ. Поэтому для получения максимального эффекта необходимо хорошо ориентироваться при выборе конкретных инструментов параллельного программирования среди множества существующих.

В настоящее время существуют различные методы классификации архитектур параллельных вычислительных систем. Подробно различные методы классификации архитектур параллельных вычислительных систем изложены в [1.3].

Одна из возможных классификаций архитектур параллельных вычислительных систем состоит в разделении параллельных вычислительных систем по типу памяти (классификация Джонсона). В

этом случае в качестве одного из классов можно выделить параллельные вычислительные системы с распределенной памятью (distributed memory) или массивно-параллельные системы (MPP). Обычно такие системы состоят из набора вычислительных узлов - каждый из них содержит один или несколько процессоров, локальную память, прямой доступ к которой невозможен из других узлов, коммуникационный процессор или сетевой адаптер, а также может содержать жесткие диски и устройства ввода/вывода. В массивнопараллельных системах могут быть и специализированные управляющие узлы и узлы ввода/вывода. Узлы в массивно-параллельных системах связаны между собой через коммуникационную среду (высокоскоростная сеть, коммутаторы либо их различные комбинации).

В качестве второго класса можно выделить архитектуры параллельных вычислительных систем с общей памятью (shared memory) или симметричные мультипроцессорные системы (SMP). Такие системы, как правило, состоят из нескольких однородных процессоров и массива общей памяти. Каждый из процессоров имеет прямой доступ к любой ячейке памяти, причем скорость доступа к памяти для всех процессоров одинакова. Обычно процессоры подключаются к памяти с помощью общей шины либо с помощью специальных коммутаторов.

Однако кроме двух вышеперечисленных классов существуют и некоторые их гибриды.

В качестве первого гибридного класса назовем системы с неоднородным доступом к памяти (Non-Uniform Memory Access) - так называемые NUMA-системы. Такие параллельные вычислительные системы обычно состоят из однородных модулей, каждый из которых содержит один или несколько процессоров и локальный для каждого модуля блок памяти. Объединение модулей между собой осуществляется при помощи специальных высокоскоростных коммутаторов (Numalink). В таких вычислительных системах адресное пространство является общим. Прямой доступ к удаленной памяти, т. е. к локальной памяти других модулей, поддерживается аппаратно. Однако время доступа процессора одного модуля к памяти другого модуля заметно больше времени доступа к локальной памяти исходного модуля. Это время может существенно различаться и зависит главным образом от

топологии соединения модулей. Очевидно, что этот третий класс является некоторой комбинацией первого и второго классов.

В качестве второго гибридного класса отметим различные комбинации соединения систем трех вышеперечисленных типов. Такие соединения могут иметь весьма сложный иерархический характер. Возможно также соединение кластеров или их отдельных частей через Интернет. Такие вычислительные системы развиваются в рамках международного проекта GRID (см., например, [1.4-1.5]).

## Современные направления развития параллельных вычислительных систем

В настоящее время началось широкое использование двуядерных микропроцессоров Pentium D, Xeon, Itanium 2, Opteron и IBM POWER5. В ближайшей перспективе ожидается появление процессоров и с большим количеством ядер. Представители компании Intel уже объявили о предстоящем начале выпуска в ближайшее время 6-ядерных процессоров Clovertown. К 2009 году компания Intel предполагает начать производство 256- ядерных процессоров. Фирма IBM в 2007 году начала выпуск процессора Cell с 9 ядрами, каждое из которых выполняет одновременно по два потока. Кластер, построенный на этих процессорах, был продемонстрирован на выставке CeBIT в марте 2006 года в Ганновере.

Отметим, что все ядра - как некоторых уже выпускаемых процессоров, например Pentium D и IBM POWER5, так и перспективных процессоров Cell - могут выполнять одновременно по два параллельных потока. Таким образом, даже персональный компьютер с одним процессором Pentium D позволяет выполнять одновременно до 4 процессов. То есть обычный персональный компьютер с одним из таких процессоров является параллельной вычислительной системой с общей памятью (SMP-системой). Поэтому освоение эффективного инструментария для программирования на таких системах является очень актуальной и важной задачей подготовки и переподготовки специалистов в области программирования.

В связи с развитием параллельных возможностей современных

микропроцессоров в настоящее время можно выделить три основных направления в развитии параллельных вычислительных систем.

Во-первых, это многоядерность (Multicore), когда несколько ядер находятся в корпусе одного процессора.

Во-вторых, это технологии с явным параллелизмом команд (EPIC - Explicitly Parallel Instruction Computing). Архитектура всех современных 64-разрядных процессоров Intel построена на использовании технологии EPIC. Параллелизм сегодня стал главным ресурсом наращивания вычислительной мощности компьютеров. Существенным препятствием к параллельному выполнению программ являются так называемые точки ветвления, в которых решается вопрос, по какому из нескольких возможных путей пойдет выполнение программы после этой точки. Чем более совершенным является механизм предсказания ветвлений, тем лучше может быть распараллелена программа. При наличии избыточных вычислительных ресурсов можно начать выполнять сразу два возможных направления работы программы, не дожидаясь, пока ее основная ветвь дойдет до точки ветвления. После того как программа достигнет точки ветвления, можно уже окончательно выбрать результаты, полученные по одной из возможных ветвей. Сущность EPIC заключается в том, что блок предсказаний ветвлений выносится из аппаратной логики процессора в компилятор. Анализируя программу, компилятор сам определяет параллельные участки и дает процессору явные инструкции по их выполнению - отсюда и следует название архитектуры EPIC. Модернизация компилятора гибче и проще, чем модификация аппаратной части параллельной вычислительной системы. Например, можно установить новую версию компилятора и посмотреть, как это отразится на скорости работы прикладных программ, создаваемых с помощью этого компилятора. При этом следует иметь в виду, что изменить устройство процессора в уже работающей вычислительной системе невозможно. Установленный процессор можно только заменить на новый, подходящий для данной системы процессор, если таковой имеется в наличии. Поэтому архитектура EPIC позволяет более эффективно модифицировать работающие вычислительные системы за счет установки более совершенных версий компилятора.

В-третьих, это многопоточность (TLP - Thread Level Parallelism), когда в

каждом ядре процессора выполняется одновременно несколько потоков, конкурирующих между собой.

Главная задача в развитии вышеперечисленных направлений - существенное повышение производительности вычислительных систем. Без развития параллельных технологий решить задачу повышения производительности вычислительных систем в настоящее время не представляется возможным, поскольку современные технологии микроэлектроники подошли к технологическому барьеру, препятствующему дальнейшему существенному увеличению тактовой частоты работы процессора и изготовлению процессоров по технологиям, существенно меньшим 15 нанометров.

## Межузловые соединения в параллельных системах

Большую роль в современных высокопроизводительных вычислительных системах играют топологии соединения процессоров и межузловые соединения. На рис. 1.1 представлены некоторые типичные топологии соединения вычислительных узлов в высокопроизводительных вычислительных системах. Отметим, что выбор топологии в соответствии со спецификой решаемых задач зачастую позволяет существенно повысить быстродействие системы при решении конкретного класса задач. Естественно, переход к решению другого класса задач может потребовать изменения топологии соединения узлов системы.

Коммуникационные технологии играют важную роль в быстродействии вычислительных систем. В настоящее время на практике получили распространение следующие типы протоколов межузловых соединений: Ethernet, Myrinet, Infiniband, SCI, Quadrics и Numalink для систем с NUMA-памятью, разрабатываемых фирмой Silicon Graphics. На физическом уровне межузловые соединения организованы с помощью специальных кабелей, коммутаторов и интерфейсных плат, соответствующих конкретному типу межузлового соединения. Интерфейсные платы должны быть установлены в каждый узел кластера. Как правило, протоколу межузлового соединения соответствуют специальные системные библиотеки функций обмена сообщениями, поставляемые в комплекте с кабелями и интерфейсными платами. Для получения максимальной производительности вычислительной системы

необходимо использовать именно эти библиотеки в процессе разработки программного обеспечения, поскольку применение иных библиотек может порой привести к весьма значительному снижению производительности вычислительной системы.

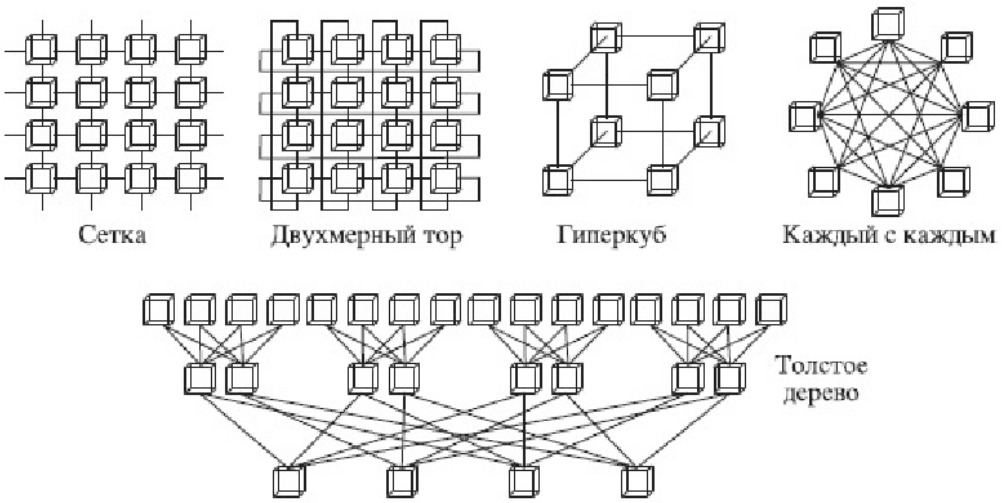



Рис. 1.1. Топологии соединения вычислительных узлов в высокопроизводительных вычислительных системах

## Инструменты создания параллельных программ

В настоящее время при создании параллельных программ, которые предназначены для многопроцессорных вычислительных систем MPP с распределенной памятью, для обмена данными между параллельными процессами, работающими в разных узлах системы, широко применяется интерфейс обмена сообщениями (Message Passing Interface, MPI). При использовании этого интерфейса обмен данными между различными процессами в программе осуществляется с помощью механизма передачи и приемки сообщений. Кроме MPI при создании параллельных программ возможны и другие методы обмена, например, основанные на использовании функций программной системы PVM (Parallel Virtual Machine) или функций библиотеки SHMEM, разработанной компаниями Cray и Silicon Graphics.

При создании параллельных программ, предназначенных для

многопроцессорных вычислительных систем с общей памятью, в настоящее время для обмена данными между процессорами обычно используются либо методы многопоточного программирования с помощью нитей ( `threads` ), либо директивы OpenMP [1.6-1.13]. Директивы OpenMP являются специальными директивами для компиляторов. Они создают и организуют выполнение параллельных процессов (нитей), а также обмен данными между процессами. Отметим также, что обмен данными между процессами в многопроцессорных вычислительных системах с общей памятью может осуществляться и с применением функций MPI и SHMEM.

В конце 2005 года компания Intel объявила о создании приложения Cluster OpenMP для компьютеров с процессорами собственного производства. Cluster OpenMP реализует расширение OpenMP, позволяет объявлять области данных доступными всем узлам кластера и тем самым распространяет методы OpenMP на создание параллельных программ для параллельных вычислительных систем с разделенной памятью. Это средство обладает рядом преимуществ по сравнению с MPI, главным из которых является простота разработки программ для Cluster OpenMP, поскольку передача данных между узлами кластера происходит неявно, по протоколу Lazy Release Consistency. Отметим, что компания Intel не является изобретателем этого подхода - ранее такие попытки предпринимались в Японии (см., например, [1.14]) и некоторых других странах).

Основными алгоритмическими языками параллельного программирования с использованием OpenMP в настоящее время являются Fortran и C/C++. Существуют многочисленные версии компиляторов, разработанные различными производителями программного обеспечения, позволяющие быстро и легко организовать параллельные вычисления с помощью методов OpenMP. Обычно директивы распараллеливания OpenMP применяются для распараллеливания последовательных программ. При этом последовательные программы не теряют своей работоспособности. Вернуться к последовательной версии очень просто: достаточно просто убрать опцию - `openmp` в вызове компилятора при компиляции программы.

Кроме компиляторов, при создании параллельных программ широко

используются специализированные отладчики и средства настройки и оптимизации программ, а также различные средства автоматического распараллеливания.

В настоящее время на рынке программного обеспечения представлен достаточно широкий спектр коммерческих программных продуктов для автоматического распараллеливания. Как правило, эти системы состоят из графических средств представления параллельных программ, препроцессоров для генерации программ на языках Fortran (77, 90, 95) и C/C++, специализированных трассировщиков и отладчиков, а также средств визуализации.

Компания Intel разрабатывает не только процессоры для параллельных вычислительных систем, но и целый спектр инструментов для отладки, оптимизации и настройки эффективной работы параллельных программ. В настоящее время в компании разработаны компиляторы Fortran и C/C++ с возможностями создания параллельных программ средствами OpenMP на платформах Linux и Windows, а также средства анализа производительности программ VTune Performance Analyzer и Intel Thread Checker, позволяющие находить критические секции в многопоточных программах и существенно ускорять их выполнение. Кроме того, разработаны библиотеки стандартных функций Intel Math Kernel Library, Intel Cluster Math Kernel Library и Intel Integrated Performance Primitives, позволяющие существенно ускорить работу параллельных программ по обработке как числовой, так и графической информации. В конце 2005 года компания анонсировала появление продукта Intel Cluster OpenMP, позволяющего распространить технологии OpenMP на кластерные системы с разделенной памятью.

При отладке параллельных программ также используется целый ряд специализированных отладчиков. Среди них отметим:

- idb - символический отладчик, разработанный корпорацией Intel. Поставляется вместе с компиляторами. Поддерживает отладку программ, написанных на алгоритмических языках Fortran и C/C++;
- gdb - свободно распространяемый отладчик GNU. Поддерживает отладку программ, написанных на алгоритмических языках C/C++ и Modula-2, а также на алгоритмическом языке Fortran 95 при

установке дополнительного модуля gdb95;

- ddd - графический интерфейс к отладчику gdb и некоторым другим отладчикам;
- TotalView - лицензионный графический отладчик для отладки приложений в средах OpenMP и MPI.

# Основные конструкции OpenMP

Настоящая лекция посвящена изложению основ параллельного программирования с использованием OpenMP. В начале обсуждаются основные принципы программирования в OpenMP и рассматривается принципиальная схема программирования. Приводятся конкретные реализации управляющих директив OpenMP для программ, написанных на алгоритмических языках Fortran и C/C++. Перечисляются основные правила применения директив OpenMP, использующихся для описания данных и организации параллельных вычислений. Обсуждаются вопросы видимости данных и корректности доступа к данным. Рассматриваются методы распараллеливания циклов и контроля распределения работы между процессорами. Приводятся способы балансировки работы процессоров с помощью директив OpenMP, а также задания внешних переменных окружения с помощью функций OpenMP.

# Основные принципы OpenMP

OpenMP - это интерфейс прикладного программирования для создания многопоточных приложений, предназначенных в основном для параллельных вычислительных систем с общей памятью. OpenMP состоит из набора директив для компиляторов и библиотек специальных функций. Стандарты OpenMP разрабатывались в течение последних 15 лет применительно к архитектурам с общей памятью. Описание стандартов OpenMP и их реализации при программировании на алгоритмических языках Fortran и C/C++ можно найти в [2.1-2.6]. Наиболее полно вопросы программирования на OpenMP рассмотрены в монографиях [2.7-2.8]. В последние годы весьма активно разрабатывается расширение стандартов OpenMP для параллельных вычислительных систем с распределенной памятью (см., например, работы [2.9]). В конце 2005 года компания Intel анонсировала продукт Cluster OpenMP, реализующий расширение OpenMP для вычислительных систем с распределенной памятью. Этот продукт позволяет объявлять области данных, доступные всем узлам кластера, и осуществлять передачу данных между узлами кластера неявно с помощью протокола Lazy Release Consistency.

OpenMP позволяет легко и быстро создавать многопоточные приложения на алгоритмических языках Fortran и C/C++. При этом директивы OpenMP аналогичны директивам препроцессора для языка C/C++ и являются аналогом комментариев в алгоритмическом языке Fortran. Это позволяет в любой момент разработки параллельной реализации программного продукта при необходимости вернуться к последовательному варианту программы.

В настоящее время OpenMP поддерживается большинством разработчиков параллельных вычислительных систем: компаниями Intel, Hewlett-Packard, Silicon Graphics, Sun, IBM, Fujitsu, Hitachi, Siemens, Bull и другими. Многие известные компании в области разработки системного программного обеспечения также уделяют значительное внимание разработке системного программного обеспечения с OpenMP. Среди этих компаний отметим Intel, KAI, PGI, PSR, APR, Absoft и некоторые другие. Значительное число компаний и научно-исследовательских организаций, разрабатывающих прикладное программное обеспечение, в настоящее время использует OpenMP при разработке своих программных продуктов. Среди этих компаний и организаций отметим ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, а также и российские компании ТЕСИС, Центральную геофизическую экспедицию и российские научно-исследовательские организации, такие как Институт математического моделирования РАН, Институт прикладной математики им. Келдыша РАН, Вычислительный центр РАН, Научно-исследовательский вычислительный центр МГУ, Институт химической физики РАН и другие.

## Принципиальная схема программирования в OpenMP

Любая программа, последовательная или параллельная, состоит из набора областей двух типов: последовательных областей и областей распараллеливания. При выполнении последовательных областей порождается только один главный поток (процесс). В этом же потоке инициируется выполнение программы, а также происходит ее завершение. В последовательной программе в областях распараллеливания порождается также только один, главный поток, и этот поток является единственным на протяжении выполнения всей программы. В параллельной программе в областях распараллеливания

порождается целый ряд параллельных потоков. Порожденные параллельные потоки могут выполняться как на разных процессорах, так и на одном процессоре вычислительной системы. В последнем случае параллельные процессы (потоки) конкурируют между собой за доступ к процессору. Управление конкуренцией осуществляется планировщиком операционной системы с помощью специальных алгоритмов. В операционной системе Linux планировщик задач осуществляет обработку процессов с помощью стандартного карусельного ( `round-robin` ) алгоритма. При этом только администраторы системы имеют возможность изменить или заменить этот алгоритм системными средствами. Таким образом, в параллельных программах в областях распараллеливания выполняется ряд параллельных потоков. Принципиальная схема параллельной программы изображена на рис.2.1.

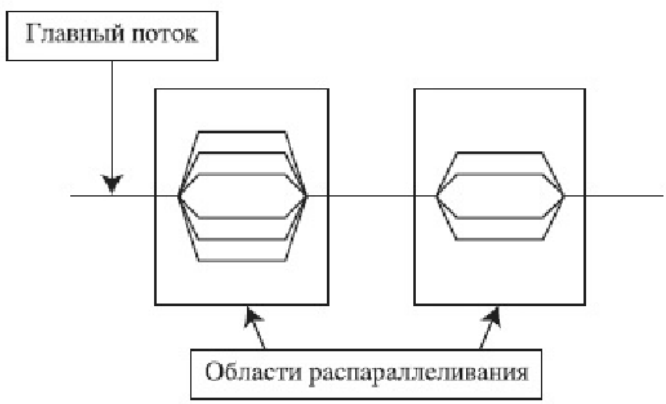



Рис. 2.1. Принципиальная схема параллельной программы

При выполнении параллельной программы работа начинается с инициализации и выполнения главного потока (процесса), который по мере необходимости создает и выполняет параллельные потоки, передавая им необходимые данные. Параллельные потоки из одной параллельной области программы могут выполняться как независимо друг от друга, так и с пересылкой и получением сообщений от других параллельных потоков. Последнее обстоятельство усложняет разработку программы, поскольку в этом случае программисту приходится заниматься планированием, организацией и синхронизацией посылки сообщений между параллельными потоками. Таким образом, при

разработке параллельной программы желательно выделять такие области распараллеливания, в которых можно организовать выполнение независимых параллельных потоков. Для обмена данными между параллельными процессами (потоками) в OpenMP используются общие переменные. При обращении к общим переменным в различных параллельных потоках возможно возникновение конфликтных ситуаций при доступе к данным. Для предотвращения конфликтов можно воспользоваться процедурой синхронизации ( `synchronization` ). При этом надо иметь в виду, что процедура синхронизации - очень дорогая операция по временным затратам и желательно по возможности избегать ее или применять как можно реже. Для этого необходимо очень тщательно продумывать структуру данных программы.

Выполнение параллельных потоков в параллельной области программы начинается с их инициализации. Она заключается в создании дескрипторов порождаемых потоков и копировании всех данных из области данных главного потока в области данных создаваемых параллельных потоков. Эта операция чрезвычайно трудоемка - она эквивалентна примерно трудоемкости 1000 операций. Эта оценка чрезвычайно важна при разработке параллельных программ с помощью OpenMP, поскольку ее игнорирование ведет к созданию неэффективных параллельных программ, которые оказываются зачастую медленнее их последовательных аналогов. В самом деле: для того чтобы получить выигрыш в быстродействии параллельной программы, необходимо, чтобы трудоемкость параллельных процессов в областях распараллеливания программы существенно превосходила бы трудоемкость порождения параллельных потоков. В противном случае никакого выигрыша по быстродействию получить не удастся, а зачастую можно оказаться даже и в проигрыше.

После завершения выполнения параллельных потоков управление программой вновь передается главному потоку. При этом возникает проблема корректной передачи данных от параллельных потоков главному. Здесь важную роль играет синхронизация завершения работы параллельных потоков, поскольку в силу целого ряда обстоятельств время выполнения даже одинаковых по трудоемкости параллельных потоков непредсказуемо (оно определяется как историей конкуренции параллельных процессов, так и текущим состоянием вычислительной

системы). При выполнении операции синхронизации параллельные потоки, уже завершившие свое выполнение, простаивают и ожидают завершения работы самого последнего потока. Естественно, при этом неизбежна потеря эффективности работы параллельной программы. Кроме того, операция синхронизации имеет трудоемкость, сравнимую с трудоемкостью инициализации параллельных потоков, т. е. эквивалентна примерно трудоемкости выполнения 1000 операций.

На основании изложенного выше можно сделать следующий важный вывод: при выделении параллельных областей программы и разработке параллельных процессов необходимо, чтобы трудоемкость параллельных процессов была не менее 2000 операций деления. В противном случае параллельный вариант программы будет проигрывать в быстродействии последовательной программе. Для эффективной работающей параллельной программы этот предел должен быть существенно превышен.

## Синтаксис директив в OpenMP

Основные конструкции OpenMP - это директивы компилятора или прагмы (директивы препроцессора) языка C/C++. Ниже приведен общий вид директивы OpenMP прагмы.

```
#pragma omp конструкция [предложение [предложение] … ]
```

**Пример 2.2. Общий вид OpenMP прагмы языка C/C++**

В языке Fortran директивы OpenMP являются строками-комментариями специального типа. Ниже в примере 2.3 приведены примеры таких строк в общем виде. Для обычных последовательных программ директивы OpenMP не изменяют структуру и последовательность выполнения операторов. Таким образом, обычная последовательная программа сохраняет свою работоспособность. В этом и состоит гибкость распараллеливания с помощью OpenMP.

Большинство директив OpenMP применяется к структурным блокам. Структурные блоки - это последовательности операторов с одной точкой входа в начале блока и одной точкой выхода в конце блока.

```
c$omp конструкция [предложение [предложение] … ]
!$omp конструкция [предложение [предложение] … ]
*$omp конструкция [предложение [предложение] … ]
```

**Пример 2.3. Общий вид директив OpenMP в программе на языке Fortran**

В примерах 2.4 и 2.5 показаны примеры структурного и неструктурного блоков соответственно во фрагментах программ, написанных на языке Fortran. Видно, что в первом примере рассматриваемый блок является структурным, поскольку имеет одну точку входа и одну точку выхода. Во втором примере рассматриваемый блок не является структурным, поскольку имеет две точки входа. Следовательно, использование конструкций OpenMP для его распараллеливания некорректно. Из этих примеров видно, что директивы OpenMP, применяемые к структурным блокам, аналогичны операторным скобкам begin и end в алгоритмических языках Algol и Pascal.

```
c$omp parallel

   10 wrk (id) = junk (id)
      res (id) = wrk (id)**2
      if (conv (res)) goto 10

c$omp end parallel

         print *, id
```

**Пример 2.4. Пример структурного блока в программе на языке Fortran**

```
c$omp parallel

   10 wrk (id) = junk (id)
   30 res (id) = wrk (id)**2
      if (conv (res)) goto 20
      goto 10

c$omp end parallel
```

```
    if (not_done) goto 30
  20 print *, id
```

**Пример 2.5. Пример неструктурного блока в программе на языке Fortran**

В следующем фрагменте программы (Пример 2.6) показан общий вид основных директив OpenMP на языке Fortran.

```
c$omp  parallel
c$omp& shared (var1, var2, …)
c$omp& private (var1, var2, …)
c$omp& firstprivate (var1, var2, …)
c$omp& lastprivate (var1, var2, …)
c$omp& reduction (operator | intrinsic: var1, var2, …)
c$omp& if (expression)
c$omp& default (private | shared | none)
      [Структурный блок программы]
c$omp end parallel
```

**Пример 2.6. Общий вид основных директив OpenMP на языке Fortran**

В рассматриваемом фрагменте предложение OpenMP `shared` используется для описания общих переменных. Как уже отмечалось ранее, общие переменные позволяют организовать обмен данными между параллельными процессами в OpenMP. Предложение `private` используется для описания внутренних (локальных) переменных для каждого параллельного процесса. Предложение `firstprivate` применяется для описания внутренних переменных параллельных процессов, однако данные в эти переменные импортируются из главного потока. Предложение `reduction` позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков. Предложение `if` является условным оператором в параллельном блоке. И, наконец, предложение `default` определяет по умолчанию тип всех переменных в последующем параллельном структурном блоке программы. Далее механизм работы этих предложений будет рассмотрен более подробно. В заключение отметим, что символ амперсанд &в шестой позиции строки используется для продолжения длинных директив OpenMP, не

умещающихся на одной строке в программах на Fortran (см. рис.2.6).

Ниже во фрагменте программы (Пример 2.7) приведен общий вид основных директив OpenMP на языке C/C++.

```
# pragma omp parallel                    \
        private (var1, var2, …)          \
        shared (var1, var2, …)           \
        firstprivate (var1, var2, …)     \
        lastprivate (var1, var2, …)      \
        copyin (var1, var2, …)           \
        reduction (operator: var1, var 2, …) \
        if (expression)                  \
        default (shared | none)          \
{
  [ Структурный блок программы]
}
```

**Пример 2.7. Общий вид основных директив OpenMP на языке C/C++**

По сравнению с предыдущим, в рассматриваемом фрагменте появилось еще одно дополнительное предложение OpenMP - `copyin`. Оно позволяет легко и просто передавать данные из главного потока в параллельные. В Fortran также существует аналогичная возможность, однако механизм ее реализации несколько иной. Подробнее эти возможности будут рассмотрены далее. Кроме того, отметим, что вместо Fortran-директивы OpenMP

```
c$omp end parallel
```

в C/C++ используются обычные фигурные скобки. Для продолжения длинных директив на следующих строках в программах на C/C++ применяется символ "обратный слэш" в конце строки (см. пример 2.7).

Рассмотренные в настоящем разделе директивы OpenMP не охватывают весь спектр директив. В следующих разделах рассмотрим более подробно основные директивы OpenMP и механизм их работы.

## Особенности реализации директив OpenMP

В этом разделе рассмотрим некоторые специфические особенности реализации директив OpenMP.

Количество потоков в параллельной программе определяется либо значением переменной окружения `OMP_NUM_THREADS`, либо специальными функциями, вызываемыми внутри самой программы. Задание переменной окружения `OMP_NUM_THREADS` в операционной системе Linux осуществляется следующим образом с помощью команды

```
export OMP_NUM_THREADS=256
```

Здесь было задано `256` параллельных потоков.

Просмотреть состояние переменной окружения `OMP_NUM_THREADS` можно, например, с помощью следующей команды:

```
export | grep OMP_NUM_THREADS
```

Каждый поток имеет свой номер `thread_number`, начинающийся от нуля (для главного потока) и заканчивающийся `OMP_NUM_THREADS-1`. Определить номер текущего потока можно с помощью функции `omp_get_thread_num()`.

Каждый поток выполняет структурный блок программы, включенный в параллельный регион. В общем случае синхронизации между потоками нет, и заранее предсказать завершение потока нельзя. Управление синхронизацией потоков и данных осуществляется программистом внутри программы. После завершения выполнения параллельного структурного блока программы все параллельные потоки за исключением главного прекращают свое существование.

Пример ветвления во фрагменте программы, написанной на языке C/C++, в зависимости от номера параллельного потока показан в примере 2.8.

```
#pragma omp parallel
{
    myid = omp_get_thread_num ( ) ;
    if (myid == 0)
      do_something ( ) ;
```

```
    else
      do_something_else (myid) ;
}
```

**Пример 2.8. Пример ветвления в программе в зависимости от номера параллельного потока**

В этом примере в первой строке параллельного блока вызывается функция `omp_get_thread_num`, возвращающая номер параллельного потока. Этот номер сохраняется в локальной переменной `myid`.Далее в зависимости от значения переменной `myid` вызывается либо функция `do_something()` в первом параллельном потоке с номером 0, либо функция `do_something_else(myid)` во всех остальных параллельных потоках.

В OpenMP существуют различные режимы выполнения (Execution Mode) параллельных структурных блоков. Возможны следующие режимы:

- Динамический режим (Dynamic Mode). Этот режим установлен по умолчанию и определяется заданием переменной окружения OMP_DYNAMIC в операционной системе Linux. Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux:

  export OMP_DYNAMIC

  Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

  omp_set_dynamic()

  Отметим, что на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует. В динамическом режиме количество потоков определяется самой операционной системой в соответствии со значением переменной окружения `OMP_NUM_THREADS`.В процессе выполнения параллельной программы при переходе от одной области распараллеливания к другой эта переменная может изменять свое значение;

- Статический режим (Static Mode). Этот режим определяется

заданием переменной окружения `OMP_STATIC` в операционной системе Linux. В этом случае количество потоков определяется программистом. Задать переменную окружения `OMP_STATIC` можно, например, с помощью следующей команды операционной системы Linux:

```
export OMP_STATIC
```

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

```
omp_set_static()
```

Однако на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует;

- Параллельные структурные блоки могут быть вложенными, но компилятор иногда по ряду причин может выполнять их и последовательно в рамках одного потока. Вложенный режим выполнения параллельных структурных блоков определяется заданием переменной окружения `OMP_NESTED=[FALSE|TRUE]` (по умолчанию задается `FALSE` ) в операционной системе Linux. Задать эту переменную можно, например, с помощью следующей команды операционной системы Linux:

```
setenv OMP_NESTED TRUE
```

Кроме того, существует возможность задать этот режим и саму переменную внутри программы, вызвав функцию

```
omp_set_nested( TRUE | FALSE )
```

Однако на компьютерах Silicon Graphics S350 и Kraftway G-Scale S350 вторая возможность отсутствует.

## Директивы OpenMP

Теперь перейдем к подробному рассмотрению директив OpenMP и

механизмов их реализации.

## Директивы shared, private и default

Эти директивы (предложения OpenMP) используются для описания типов переменных внутри параллельных потоков.

Предложение OpenMP

```
shared( var1, var2, …, varN )
```

определяет переменные `var1, var2, …, varN` как общие переменные для всех потоков. Они размещаются в одной и той же области памяти для всех потоков.

Предложение OpenMP

```
private( var1, var2, …, varN )
```

определяет переменные `var1, var2, …, varN` как локальные переменные для каждого из параллельных потоков. В каждом из потоков эти переменные имеют собственные значения и относятся к различным областям памяти: локальным областям памяти каждого конкретного параллельного потока.

В качестве иллюстрации использования директив OpenMP `shared` и `private` рассмотрим фрагмент программы (Пример 2.9). В этом примере переменная `a` определена как общая и является идентификатором одномерного массива. Переменные `myid` и `x` определены как локальные переменные для каждого из параллельных потоков. В каждом из параллельных потоков локальные переменные получают собственные значения. После чего при выполнении условия `x<1.0` значение локальной переменной `x` присваивается `myid` -й компоненте общего для всех потоков массива `a`. Значение `x` будет неопределенным, если не определить `x` как переменную типа `private`. Отметим, что значения `private` -переменных не определены до и после блока параллельных вычислений.

```
c$omp parallel shared (a)
c$omp& private (myid, x)
      myid = omp_get_thread_num ( )
      x = work (myid)
      if (x < 1.0) then
        a (myid) = x
      endif
```

**Пример 2.9. Пример использования директив OpenMP shared и private в параллельной области программы**

Предложение OpenMP

```
default ( shared | private | none )
```

задает тип всех переменных, определяемых по умолчанию в последующем параллельном структурном блоке как `shared`, `private` или `none`. Например, если во фрагменте программы (примере 2.9) вместо

```
private( myid, x )
```

написать

```
default( private )
```

то определяемые далее по умолчанию переменные `myid` и `x` будут автоматически определены как `private`.

## Директивы firstprivate и lastprivate

Директивы (предложения) OpenMP `firstprivate` и `lastprivate` используются для описания локальных переменных, инициализируемых внутри параллельных потоков. Переменные, описанные как `firstprivate`, получают свои значения из последовательной части программы. Переменные, описанные как `lastprivate`, сохраняют свои значения при выходе из параллельных потоков при условии их последовательного выполнения.

Предложение OpenMP

```
firstprivate( var1, var2, ..., varN )
```

определяет переменные `var1, var2, ..., varN` как локальные переменные для каждого из параллельных потоков, причем инициализация значений этих переменных происходит в самом начале параллельного структурного блока по значениям из предшествующего последовательного структурного блока программы.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.10). В этом примере в каждом параллельном потоке используется своя переменная `c`, но значение этой переменной перед входом в параллельный блок программы берется из предшествующего последовательного блока.

```
program first
    integer :: myid, c
      integer, external :: omp_get_thread_num
      c=98
!$omp parallel private (myid)
!$omp& firstprivate (c)
      myid = omp_get_thread_num ( )
      write (6, *) 'T: ', myid, 'c=', c
!$omp end parallel
      end program first
```

**Пример 2.10. Пример использования директивы OpenMP firstprivate в параллельной области программы**

```
T:1 c=98
T:3 c=98
T:2 c=98
T:0 c=98
```

Предложение OpenMP

```
lastprivate( var1, var2, ..., varN )
```

определяет переменные `var1, var2, ..., varN` как локальные

переменные для каждого из параллельных потоков, причем значения этих переменных сохраняются при выходе из параллельного структурного блока при условии, что параллельные потоки выполнялись в последовательном порядке.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.11). В этом примере локальная переменная `i` принимает различные значения в каждом потоке параллельного структурного блока. После завершения параллельного структурного блока переменная `i` сохраняет свое последнее значение, полученное в последнем параллельном потоке, при условии последовательного выполнения всех параллельных потоков, т. е. `n=N+1`.

```
c$omp  do shared ( x )
c$omp& lastprivate ( i)
      do i = 1, N
        x (i) = a
      enddo
      n = i
```

**Пример 2.11. Пример использования директивы OpenMP lastprivate в параллельной области программы**

## Директива if

Директива (предложение) OpenMP `if` используется для организации условного выполнения потоков в параллельном структурном блоке.

Предложение OpenMP

```
if( expression )
```

определяет условие выполнения параллельных потоков в последующем параллельном структурном блоке. Если выражение `expression` принимает значение `TRUE` (Истина), то потоки в последующем параллельном структурном блоке выполняются. В противном случае, когда выражение `expression` принимает значение `FALSE` (Ложь), потоки в последующем параллельном структурном блоке не выполняются.

В качестве иллюстрации рассмотрим фрагмент программы (Пример 2.12).

```
c$omp parallel do if (n .ge. 2000)
      do i = 1, n
        a (i) = b (i)*c + d (i)
      enddo
```

**Пример 2.12. Пример использования директивы OpenMP if в параллельной области программы**

В этом примере цикл распараллеливается только в том случае ( `n>2000` ), когда параллельная версия будет заведомо быстрее последовательной. Напомним, что трудоемкость образования параллельных потоков эквивалентна примерно трудоемкости 1000 операций.

## Директива reduction

Директива OpenMP `reduction` позволяет собрать вместе в главном потоке результаты вычислений частичных сумм, разностей и т. п. из параллельных потоков последующего параллельного структурного блока.

В предложении OpenMP

```
reduction( operator | intrinsic: var1 [, var2,..., varN])
```

определяется `operator` - операции ( `+`, `-`, `*`, `/` и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока. Кроме того, определяется список локальных переменных `var1, var2, ..., varN`, в котором будут сохраняться соответствующие частичные значения. После завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.

В качестве иллюстрации рассмотрим фрагмент программы (пример 2.13).

```
c$omp  do shared (x) private (i)
c$omp& reduction (+ : sum)
      do i = 1, N
        sum = sum + x (I)
      enddo
```

**Пример 2.13. Пример вычисления суммы с использованием директивы OpenMP reduction в параллельной области программы**

В этом примере в каждом параллельном потоке определена локальная переменная `sum` для вычисления частичных сумм. После завершения параллельных потоков все локальные переменные `sum` суммируются, а результат сохраняется в одноименной общей (глобальной) переменной `sum`.

Во фрагменте программы (пример 2.14) вычисляется минимальное значение по результатам вычисления частичных минимумов, вычисленных в параллельных потоках. В этом примере в каждом параллельном потоке определяются локальные минимумы по результатам сравнения значений `x(i)` и `gmin`. После завершения всех параллельных потоков вычисляется значение общей переменной `gmin` по результатам сравнения значений локальных переменных `gmin` в параллельных потоках.

```
c$omp  do shared (x) private (i)
c$omp& reduction (min : gmin)
      do i = 1, N
        gmin = min (gmin, x (i))
      enddo
```

**Пример 2.14. Пример вычисления минимума с использованием директивы OpenMP reduction в параллельной области программы**

В языке Fortran в качестве параметров `operator` и *`intrinsic`* в предложениях `reduction` допускаются следующие операции и функции:

- параметр `operator` может быть одной из следующих арифметических или логических операций: `+`, `-`, `*`, `.and.`, `.or.` `.eqv.`, `.neqv.`;

- параметр *intrinsic* может быть одной из следующих функций: max, min, iand, ior, ieor.

  В языке C/C++ в качестве параметров operator и *intrinsic* в предложениях reduction допускаются следующие операции и функции:

- параметр operator может быть одной из следующих арифметических или логических операций: +, -, *, &, ^, &&, |
- параметр *intrinsic* может быть одной из следующих функций: max, min ;
- указатели и ссылки в предложениях reduction использовать строго запрещено!

## Директива copyin

Директива OpenMP copyin используется для передачи данных из главного потока в параллельные потоки, называемой миграцией данных. Механизмы реализации этой директивы в языках C/C++ и Fortran различны. Рассмотрим далее реализации этой директивы поочередно в C/C++ и Fortran.

Предложение OpenMP в языке C/C++

```
copyin( var1 [, var2[, ...[, varN]]] )
```

определяет список локальных переменных var1, var2, ..., varN, которым присваиваются значения из одноименных общих переменных, заданных в глобальном потоке.

В Fortran в качестве параметров директивы OpenMP copyin задаются имена common -блоков ( cb1, cb2, ..., *cbN* ), в которых описаны мигрирующие переменные

```
copyin( /cb1/ [, /cb2/ [, ...[, /cbN/ ]]] )
```

т.е. такие переменные из глобального потока, которые передают свои значения своим копиям, определенным в параллельных потоках в

качестве локальных переменных.

В качестве иллюстрации миграции данных рассмотрим фрагмент Fortran-программы (Пример 2.15). В этом примере мигрирующая целочисленная переменная `x` хранится в `common` -блоке `mine`. Значение этой общей переменной задается в главном потоке. В каждом параллельном потоке используется своя локальная переменная x, которой присваивается значение общей переменной `x`.

```
integer : : x, tid
      integer, external :: omp_get_thread_num ( )
      common/mine/ x
! $omp threadprivate (/mine/)
      x=33
      call omp_set_num_threads (4)
! $omp parallel private (tid) copyin (/mine/)
      tid=omp_get_thread_num ( )
      print *, 'T: ',tid, ' x = ', x
! $omp end parallel
```

**Пример 2.15. Пример использования директивы OpenMP copyin для организации миграции данных в Fortran-программе**

Результаты работы программы:

```
T: 1 x = 33
T: 2 x = 33
T: 0 x = 33
T: 3 x = 33
```

## Директива for

Директива OpenMP `for` используется для организации параллельного выполнения петель циклов в программах, написанных на языке C/C++.

Предложение OpenMP в программе на C/C++

```
#pragma omp for
```

означает, что оператор `for`, следующий за этим предложением, будет выполняться в параллельном режиме, т. е. каждой петле этого цикла будет соответствовать собственный параллельный поток.

В качестве иллюстрации использования директивы OpenMP `for` рассмотрим фрагмент программы на языке C (Пример 2.16). В этом примере петли цикла оператора `for` реализуются как набор параллельных потоков. По умолчанию в конце цикла реализуется функция синхронизации *`barrier`* (барьер). Для отмены функции *`barrier`* следует воспользоваться предложением OpenMP *`nowait`*.

```
#pragma omp parallel shared (a, b) private (j)
{
#pragma omp for
     for (j=0; j<N; j++)
       a [j] = a [j] + b [j];
```

**Пример 2.16. Пример использования директивы OpenMP for для организации параллельной обработки петель циклов в программе на языке C/C++**

## Директива do

Директива OpenMP `do` используется для организации параллельного выполнения петель циклов в программах, написанных на языке Fortran. Предложения OpenMP в программе на языке Fortran

```
c$omp [ parallel ] do [ предложение [ предложение [ … ]]]
                  do loop
 [c$omp end do [nowait]]
```

означает, что петли ( `loop` ) оператора `do` будут реализованы как параллельные процессы. Последнее предложение, содержащее `end do`, завершает параллельный цикл `do`. Оно не является обязательным, но может включать предложение *`nowait`* для отмены функции синхронизации *`barrier`*, неявно присутствующей в конце цикла `do`. В качестве предложений в директиве `do` возможны следующие конструкции:

- `private( list )`
- `firstprivate( list )`
- `lastprivate( list )`
- `reduction( operator : list ) o ordered`
- `schedule( kind [,chunk_size] )`
- *`nowait`*

Большинство из этих OpenMP-конструкций уже было рассмотрено ранее. А некоторые, такие как `ordered` и `schedule( kind [,chunk_size] )`, будут рассмотрены далее.

Пример фрагмента программы на языке Fortran с применением директивы OpenMP `do` для распараллеливания выполнения петель цикла приведен в примере 2.17.

```
c$omp parallel do
    do i = 1, n
      a (i) = b (i) *c+ d (i)
    enddo
```

**Пример 2.17. Пример использования директивы OpenMP do для организации параллельной обработки петель циклов в программе на языке Fortran**

## Директива workshare

Директива OpenMP `workshare` используется для организации параллельного выполнения петель циклов только в программах, написанных на языках Fortran 90/95. Эта директива появилась в стандарте OpenMP начиная с версии 2.0.

Синтаксис предложения OpenMP workshare в программах на языках Fortran 90/95 имеет следующий вид:

```
c$omp [ parallel ] workshare
                loop
 c$omp end workshare [nowait]
```

В результате все петли `loop` реализуются как параллельные процессы. Распределение петель по процессам осуществляется компилятором. Последнее предложение, содержащее `end workshare`, завершает параллельное выполнение петель `loop` с неявно присутствующей синхронизацией *barrier*. Для отмены синхронизации следует включить предложение *nowait*.

В следующем примере иллюстрируется использование директивы `workshare`:

```
c$omp parallel workshare
      A = A + B
 c$omp end workshare
```

В этом примере `A` и `B` являются массивами одинаковой размерности, например `N`. Отметим, что рассмотренный пример можно также переписать, например, в следующем виде с использованием оператора цикла и OMP-предложения `parallel do`:

```
c$omp parallel do private ( I )
      do I = 1, N
        A (I) = A (I) + B (I)
      enddo
 c$omp end parallel do
```

Важно отметить, что в компиляторах Fortran компании Intel директива `workshare` не поддерживается. Отметим также, что компиляторы Fortran 90/95 компаний IBM и SUN поддерживают эту директиву. Также еще раз отметим, что в программах на языках C/C++ директива `workshare` недопустима.

## Директива sections

Директива OpenMP `sections` используется для выделения участков программы в области параллельных структурных блоков, выполняющихся в отдельных параллельных потоках.

Описание синтаксиса предложения OpenMP `sections` в программе

на языке C/C++ приведено в примере 2.18.

```
#pragma omp sections [предложение [предложение …]]
{
#pragma omp section
    structured block
[#pragma omp section
    structured block
…
]
}
```

**Пример 2.18. Синтаксис предложения OpenMP sections в программе на C/C++**

Каждый структурный блок ( `structured block` ), следующий за прагмой

```
#pragma omp sections
```

выполняется в отдельном параллельном потоке. Общее же число параллельных потоков равно количеству структурных блоков (section), перечисленных после прагмы

```
#pragma omp sections [ предложение [ предложение … ] ]
```

В качестве предложений допускаются следующие OpenMP директивы: `private(list)`, `firstprivate(list)`, `lastprivate(list)`, `reduction(operator :list)` и *`nowait`*.

Описание синтаксиса предложения OpenMP `sections` в программе на языке Fortran приведено в примере 2.19.

```
c$omp  sections [предложение [, предложение] …]
c$omp  section
       code block
[c$omp section
       another code block
[c$omp section
```

```
    …]]
c$omp end sections [nowait]
```

**Пример 2.19. Синтаксис предложения OpenMP sections в программе на языке Fortran**

В качестве предложений допускаются все перечисленные выше директивы OpenMP.

Для большей ясности рассмотрим фрагмент программы на языке Fortran, приведенный в примере 2.20. В этом примере каждая параллельная секция выполняется в отдельном параллельном потоке. Всего в параллельной области определено три параллельных потока, реализующих в данном случае функциональную декомпозицию.

Отметим, что выражение `c$omp end sections` неявно реализует функцию синхронизации *barrier*. Для ее отмены следует воспользоваться предложением *nowait*.

```
c$omp parallel
c$omp sections
c$omp section
      call computeXpart ( )
c$omp section
      call computeYpart ( )
c$omp section
      call computeZpart ( )
c$omp end sections
c$omp end parallel
      call sum ( )
```

**Пример 2.20. Использование предложения OpenMP sections в программе на языке Fortran**

## Директива single

Директива OpenMP `single` используется для выделения участков программы в области параллельных структурных блоков, выполняющихся только в одном из параллельных потоков. Во всех

остальных параллельных потоках выделенный директивой `single` участок программы не выполняется, однако параллельные процессы, выполняющиеся в остальных потоках, ждут завершения выполнения выделенного участка программы, т. е. неявно реализуется процедура синхронизации. Для предотвращения этого ожидания в случае необходимости можно использовать предложение OpenMP *`nowait`*, как будет показано ниже.

Описание синтаксиса предложения OpenMP `single` в программе на языке C/C++ приведено в примере 2.21.

```
#pragma omp single [предложение [предложение …] ]
      structured block
```

**Пример 2.21. Синтаксис предложения OpenMP single в программе на языке C/C++**

Структурный блок ( `structured block` ), следующий за прагмой

```
#pragma omp single
```

выполняется только в одном из потоков. В качестве предложений допускаются следующие OpenMP директивы: `private(list)`, `firstprivate(list)`. Описание синтаксиса предложения OpenMP `single` в программе на языке Fortran приведено в примере 2.22.

```
c$omp single [предложение [предложение …] ]
    structured block
c$omp end single [nowait]
```

**Пример 2.22. Синтаксис предложения OpenMP single в программе на языке Fortran**

Структурный блок ( `structured block` ), следующий за директивой

```
c$omp single
```

выполняется только в одном из параллельных потоков. В качестве предложений допускаются следующие OpenMP директивы: `private(list)`, `firstprivate(list)`.

# Загрузка и синхронизация в OpenMP

В настоящей лекции рассматривается загрузка, синхронизация и балансировка параллельных потоков в OpenMP.

Важно отметить, что неудачное решение проблем загрузки, синхронизации и балансировки параллельных потоков может свести на нет все усилия по разработке параллельной версии программы. В результате параллельная версия программы может оказаться не быстрее, а существенно медленнее последовательной версии программы. В связи с этим при разработке параллельной версии программы проблемам загрузки, синхронизации и балансировки процессов следует уделять самое пристальное внимание. В случае разбалансировки, когда не все процессоры завершают свою работу одновременно, эффективность использования многопроцессорных параллельных вычислительных систем резко снижается и может не оправдывать затрат на приобретение таких дорогостоящих систем.

Рассмотрению процессов балансировки процессов в вычислительных системах посвящено много литературы (см., например, работы [3.1-3.14]). В настоящее время можно выделить две основных группы методов решения проблем балансировки: статические и динамические. В основе статических методов лежит принцип геометрического параллелизма, когда вычислительный процесс представляется в виде взвешенного графа, а затем этот граф разбивается на эквивалентные части [3.1-3.9]. Для решения таких задач используются различные спектральные и эвристические подходы, а также их комбинации. Отметим, что для построения графов, описывающих вычислительный процесс, важно иметь и учитывать как можно больше априорной информации о процессе, поскольку в этом случае повышается точность описания.

В методах динамической балансировки априорная информация не используется. В них процессы загружаются в освободившиеся процессоры по мере освобождения последних [3.10]. При этом более равномерная загрузка процессоров получается при достаточно большом количестве независимых процессов. Решение о загрузке процессоров может приниматься по-разному: либо в одном отдельном управляющем процессе, либо в разных управляющих процессах, каждый из которых

ассоциирован с вычислительным процессом. Последний подход известен также как метод коллективного решения. Этот подход развивается и разработан в Институте математического моделирования РАН [3.11]. Апробация динамической балансировки, основанной на идеях коллективного решения, показала эффективность подхода при решении различных классов вычислительных задач математической физики.

Отметим также, что в библиотеке PVM имеются средства управления процессами (Process Management), работающие в рамках системы управления ресурсами (General Resource Manager - GRM) [3.12]. Использование этих средств совместно с библиотекой PVM MPI позволяет существенно улучшить балансировку и производительность параллельных кластеров [3.13-3.14].

## Синхронизация процессов в OpenMP

Проблема синхронизации параллельных потоков важна не только для параллельного программирования с использованием OpenMP, но и для всего параллельного программирования в целом. Проблема состоит в том, что любой структурный параллельный блок по определению имеет одну точку выхода, за которой обычно находится последовательный структурный блок. Вычисления в последовательном блоке, как правило, могут быть продолжены, если завершены все процессы в параллельном структурном блоке и их результаты корректно переданы в последовательный блок. Именно для обеспечения такой корректной передачи данных и необходима процедура синхронизации параллельных потоков.

В предшествующей лекции при изучении директив работы с циклами проблема синхронизации уже затрагивалась. Во-первых, было указано, что эта процедура является весьма трудоемкой и сопоставима с трудоемкостью инициализации параллельных потоков (т. е. эквивалентна примерно трудоемкости 1000 операций). Поэтому желательно пользоваться синхронизацией как можно реже.

Во-вторых, было отмечено, что неявно (по умолчанию) синхронизация параллельных процессов обеспечивается при выполнении циклов в параллельном режиме. Была упомянута директива `nowait` для

устранения неявной синхронизации при завершении циклов. Однако пользоваться этой директивой следует весьма и весьма аккуратно, предварительно проанализировав порядок работы программы и убедившись, что отмена синхронизации не приведет к порче данных и непредсказуемым результатам.

Механизм работы синхронизации можно описать следующим образом. При инициализации набора параллельных процессов в программе устанавливается контрольная точка (аналогичная контрольной точке в отладчике), в которой программа ожидает завершения всех порожденных параллельных процессов. Отметим, что пока все параллельные процессы свою работу не завершили, программа не может продолжить работу за точкой синхронизации. А поскольку все современные высокопроизводительные процессоры являются процессорами конвейерного типа, становится понятной и высокая трудоемкость процедуры синхронизации. В самом деле, пока не завершены все параллельные процессы, программа не может начать подготовку загрузки конвейеров процессоров. Вот это-то и ведет к большим потерям при синхронизации процессов, аналогичных потерям при работе условных операторов в обычной последовательной программе.

Всего в OpenMP существует шесть типов синхронизации:

- `critical`,
- `atomic`,
- *`barrier`*,
- `master`,
- `ordered`,
- `flush`.

Далее подробно рассмотрим эти типы.

## Синхронизация типа atomic

Этот тип синхронизации определяет переменную в левой части оператора присваивания, которая должна корректно обновляться

несколькими нитями. В этом случае происходит предотвращение прерывания доступа, чтения и записи данных, находящихся в общей памяти, со стороны других потоков.

Для задания этого типа синхронизации в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp atomic
<операторы программы>
```

В программах, написанных на языке Fortran, задание синхронизации типа `atomic` осуществляется с помощью следующего предложения OpenMP:

```
c$omp atomic
<операторы программы>
```

Отметим, что синхронизация `atomic` является альтернативой директивы *`reduction`*. Применяется эта синхронизация только для операторов, следующих непосредственно за определяющей ее директивой. Синхронизация `atomic` - очень дорогая операция с точки зрения трудоемкости выполнения программы. Она выполняется автоматически по умолчанию при завершении циклов в параллельном режиме. Для того чтобы ее исключить, следует использовать директиву *`nowait`*.

Пример установки синхронизации типа `atomic` приведен во фрагменте программы в примере 3.1.

```
integer, dimension (8) :: a, index
     data index/ 1, 1, 2, 3, 1, 4, 1, 5 /
c$omp parallel private (I), shared (a, index)
c$omp do
     do I =1, 8
c$omp atomic
        a(index(I)) = a(index(I)) + index(I)
     enddo
c$omp end parallel
```

**Пример 3.1. Установка синхронизации типа atomic**

В приведенном примере в параллельном режиме синхронизируются только вычисления оператора

```
a(index(I)) = a(index(I)) + index(I)
```

## Синхронизация типа critical

Этот тип синхронизации используется для описания структурных блоков, выполняющихся только в одном потоке из всего набора параллельных потоков.

Для задания синхронизации типа `critical` в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp critical [ name ]
<структурный блок программы>
```

В программах, написанных на языке Fortran, задание синхронизации типа `critical` осуществляется с помощью следующего предложения OpenMP:

```
c$omp critical [ name ]
< структурный блок программы >
c$omp end critical [ name ]
```

Здесь `name` - имя критической секции ( `critical section` ). Разные критические секции независимы, если они имеют разные имена. Не поименованные критические секции относятся к одной и той же секции.

Пример использования синхронизации типа `critical` приведен во фрагменте программы (пример 3.2). В этом примере определены две критических секции `name1` и `name2`, каждая из которых выполняется только в одном из параллельных потоков. Этот тип синхронизации очень важен для достижения пиковой производительности программ.

```
integer :: cnt1, cnt2
c$omp parallel private (i)
c$omp& shared (cnt1, cnt2)
```

```
c$omp do
   do i = 1, n
      ... do work ...
      if (condition1) then
c$omp critical (name1)
         cnt1 = cnt1 + 1
c$omp end critical (name1)
      else
c$omp critical (name1)
         cnt1 = cnt1 - 1
c$omp end critical (name1)
      endif
      if (condition2) then
c$omp critical (name2)
         cnt2 = cnt2+1
c$omp end critical (name2)
      endif
   enddo
c$omp end parallel
```

**Пример 3.2. Пример синхронизации типа critical**

## Синхронизация типа barrier

Синхронизация типа *barrier* устанавливает режим ожидания завершения работы всех запущенных в программе параллельных потоков при достижении точки *barrier*.

Для задания синхронизации типа *barrier* в OpenMP в программах, написанных на языке C/C++, используется прагма

```
#pragma omp barrier
```

В программах, написанных на языке Fortran, задание синхронизации типа *barrier* осуществляется с помощью следующего предложения OpenMP:

```
c$ omp barrier
```

Пример использования синхронизации типа *barrier* приведен во фрагменте программы (пример 3.3). В этом примере синхронизируется выполнение блока операторов <assignment>. Следующий блок <dependent work> начинает свою работу во всех параллельных потоках лишь только после того, как во всех параллельных потоках будут завершено выполнение блока <assignment>.

Отметим, что неявно синхронизация типа *barrier* по умолчанию устанавливается в конце циклов. Для ее отмены можно воспользоваться директивой OpenMP *nowait*. Вопросы применения этой директивы подробно рассматривались в предыдущей лекции в разделе, посвященном операторам циклов.

```
c$omp parallel
c$omp do
      do i = 1, n
        <assignment>
c$omp barrier
        <dependent work>
      enddo
c$omp end parallel
```

**Пример 3.3. Пример синхронизации типа barrier**

Здесь же отметим, что на языке Fortran определение директивы *nowait* выглядит так:

```
c$omp end do nowait
```

а на языке C/C++ - так:

```
#pragma omp for nowait
```

## Синхронизация типа master

Синхронизация типа master используется для определения структурного блока программы, который будет выполняться исключительно в главном

потоке (параллельном потоке с нулевым номером) из всего набора параллельных потоков. Этот структурный блок следует за директивой

```
#pragma omp master
```

в программах, написанных на языке C/C++. В программах на языке Fortran синхронизация типа master устанавливается следующим образом:

```
c$omp master
<структурный блок>
c$omp end master
```

В примере 3.4 приведен пример фрагмента программы, иллюстрирующий использование синхронизации типа `master`. В этом примере операторы `print` и `read` выполняются исключительно в главном потоке.

```
! $omp parallel shared (c, scale)
! $omp& private (j, myid)
      myid = omp_get_thread_num ( )
! $omp master
      print *, 'T: ', myid, ' enter scale'
      read *, scale
! $omp end master
! $omp barrier
! $omp do
      do j = 1, N
        c (j) = scale * c (j)
      enddo
! $omp end do
! $omp end parallel
```

**Пример 3.4. Пример синхронизации типа master**

## Синхронизация типа ordered

Синхронизация типа `ordered` используется для определения потоков в параллельной области программы, которые выполняются в порядке, соответствующем последовательной версии программы.

```
c$omp parallel default (shared) private (I, J)
c$omp do ordered
      do I = 1, N
        do J = 1, M
          Z(I) = Z(I) + X (I, J) * Y(J, I)
        enddo
c$omp ordered
        if (I<21) then
          print*, 'Z (',I,') = ', Z (I)
        endif
c$omp end ordered
      enddo
```

**Пример 3.5. Пример программы с синхронизацией типа ordered**

Результаты работы программы:

```
Z (1) = 1007.167786
Z (2) = 1032.933350
Z (3) = 1033.125610
Z (4) = 1009.944641
Z (5) = 1016.547302
Z (6) = 1005.789124
Z (7) = 1025.048584
Z (8) = 1003.904358
Z (9) = 995.5405273
Z (10) = 991.2892456
Z (11) = 1011.334167
Z (12) = 1010.631897
Z (13) = 1009.581848
Z (14) = 976.2397461
Z (15) = 978.1119385
Z (16) = 977.7111816
Z (17) = 971.2011719
Z (18) = 998.4275513
Z (19) = 1018.487000
Z (20) = 978.0640259
```

В программах на языке C/C++ синхронизация типа `ordered` описывается следующим образом:

```
#pragma omp ordered
<структурный блок>
```

а в программах, написанных на языке Fortran, синхронизация типа `ordered` устанавливается так:

```
c$omp ordered
<структурный блок>
c$omp end ordered
```

В примере 3.5 приведен пример фрагмента программы, иллюстрирующий применение синхронизации типа `ordered`.

Видно, что в приведенном примере вывод результатов элементов массива `Z` осуществляется в порядке возрастания индексов элементов массива, как в обычной последовательной программе.

## Синхронизация типа flush

Синхронизация типа `flush` используется для обновления значений локальных переменных, перечисленных в качестве аргументов этой команды, в оперативной памяти. После выполнения этой директивы все переменные, перечисленные в этой директиве, имеют одно и то же значение для всех параллельных потоков.

В программах на языке C/C++ синхронизация типа `flush` описывается следующим образом:

```
#pragma omp flush(var1, [var2, [..., varN ]])
```

На языке Fortran эта директива синхронизации типа `flush` выглядит так:

```
c$omp flush var1, [var2, [..., varN ]])
```

Здесь `var1, var2, ..., varN` - список переменных, значения которых сохраняются в оперативной памяти в момент выполнения директории `flush`.

Пример фрагмента программы, иллюстрирующий использование синхронизации типа `flush`, приведен в примере 3.6. В этом примере значения переменной `done` записываются в оперативную память для того, чтобы все другие процессы имели доступ к актуальным значениям этой переменной.

```
program flush
      integer, parameter :: M=1600000
      integer, dimension (M) :: c
      integer :: stop, sum, tid
      integer, dimension (0:1) :: done
      integer, external :: omp_get_thread_num

      call omp_set_num_threads (2)
      c = 1
      c (345) = 9
!$omp parallel default (private) shared (done, c, stop)
      tid=omp_get_ thread_num ( )
      done (tid) = 0
      if (tid == 0) then
        neigh = 1
      else
        neigh = 0
      end if
!$omp barrier
      if (tid == 0) then
        do j = 1, M
          if (c j) == 9) stop = j
        enddo
      endif
      done (tid) = 1
!$omp flush (done)
      do while (done(neigh) .eq. 0)
!$omp flush (done)
      enddo
      if (tid == 1) then
        sum=0
        do j = 1, stop - 1
          sum = sum + c (j)
```

```
          enddo
        endif
!$omp end parallel
      end program flush
```

**Пример 3.6. Пример синхронизации типа flush**

## Загрузка процессов в OpenMP. Директива schedule

Проблема загрузки параллельных потоков является важной проблемой не только для параллельного программирования с использованием OpenMP, но и для всего параллельного программирования в целом. Эта проблема тесно связана с проблемой балансировки загрузки процессоров параллельных высокопроизводительных вычислительных систем, а также с проблемой повышения эффективности работы параллельных программ. Понятно, что для высокопроизводительной параллельной вычислительной системы успешное решение проблемы балансировки загрузки процессоров является ключом к решению задачи повышения эффективности вычислительной системы в целом. Как известно, процессы по-разному могут использовать вычислительные возможности процессоров. Так, на стадии интенсивных арифметических вычислений коэффициент загрузки процессоров обычно близок к 100%. На стадии интенсивных операций ввода/вывода коэффициент загрузки процессоров меняется в диапазоне от нескольких процентов до десятков процентов. Кроме того, процессоры могут простаивать и в том случа е, когда процессы, переданные им, уже обработаны, а другие процессоры все еще продолжают обрабатывать процессы той же задачи. Для исключения простоев или уменьшения времени простоев применяют различные методы балансировки процессов. Существующие в OpenMP различные методы загрузки процессов также могут быть применены для улучшения балансировки работы параллельных вычислительных систем.

Для распределения работы между процессами в OpenMP имеется директива `schedule` с параметрами, позволяющими задавать различные режимы загрузки процессоров. Ниже приведен общий вид предложения `schedule` в OpenMP.

```
schedule( type [ , chunk ] )
```

Здесь `type` - параметр, определяющий тип загрузки, а *`chunk`* - параметр, который определяет порции данных, пересылаемых между процессами (по умолчанию значение параметра *`chunk`* равно `1` ).

В OpenMP параметр `type` принимает одно из следующих значений:

- `static`,
- `dynamic`,
- `guided`,
- `runtime`.

## Загрузка типа static

В этом случае вся совокупность загружаемых процессов разбивается на равные порции размера *`chunk`*, и эти порции последовательно распределяются между процессорами (или потоками, которые затем и выполняются на этих процессорах) с первого до последнего и т. д.

В качестве примера использования загрузки типа `static` рассмотрим фрагмент программы на языке Fortran, приведенный в примере 3.7.

```
c$omp do shared (x) private (i)
c$omp& schedule (static, 1000)
      do i = 1, 12000
        ... work ...
      enddo
```

**Пример 3.7. Пример загрузки schedule(static, chunk=1000)**

Схема распределения загрузки процессов по процессорам (или потокам ( `threads` ), которые затем выполняются на процессорах), соответствующая этому примеру, приведена на рис.3.1. Как видно из этой схемы, все 12000 процессов разбиты на 12 порций по 1000 процессов ( *`chunk`*`=1000` ). Загрузка порций в потоки ( `threads` ) происходит последовательно. Сначала порции в порядке нумерации загружаются в первый, второй, третий и четвертый потоки, а затем загрузка производится опять в том же порядке, начиная с первого потока и т. д.

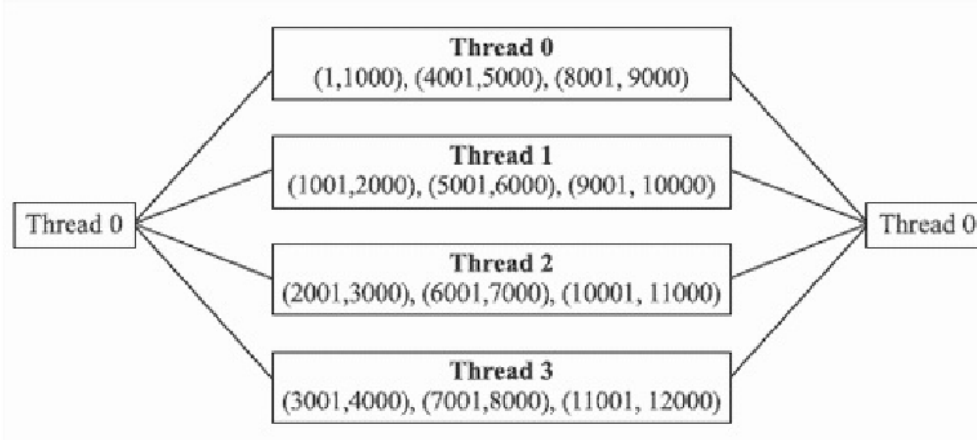



Рис. 3.1. Схема загрузки процессоров для программы, приведенной в примере 3.7

## Загрузка типа dynamic

В этом случае вся совокупность загружаемых процессов, как и в предыдущем варианте, разбивается на равные порции размера *chunk*, но эти порции загружаются последовательно в освободившиеся потоки (процессоры).

Пример применения загрузки типа `dynamic` приведен в следующем фрагменте программы (пример 3.8).

```
c$omp do shared (x) private (i)
c$omp& schedule (dynamic, 1000)
      do i = 1, 10000
        ... work ...
      enddo
```

**Пример 3.8. Пример загрузки schedule(dynamic, chunk=1000)**

## Загрузка типа guided

В этом случае вся совокупность загружаемых процессов разбивается на

порции, размер которых определяется операционной системой динамически и не превышает размер *chunk*. Загрузка порций происходит, как и в динамическом режиме, в первый освободившийся поток, затем следующий освободившийся поток и т. д.

Пример применения загрузки типа `guided` приведен в следующем фрагменте программы, изображенном на пример 3.9.

```
c$omp do shared (x) private (i)
c$omp& schedule (guided, 55)
      do i = 1, 12000
        ... work ...
      enddo
```

**Пример 3.9. Пример загрузки schedule(guided, chunk=55)**

В этом режиме обеспечивается достаточно сбалансированная загрузка потоков с небольшими задержками при завершении параллельной обработки.

## Загрузка типа runtime

В этом случае загрузка порций процессов по потокам (или процессорам) определяется значением переменной окружения `OMP_SCHEDULE`. Значение этой переменной проверяется перед каждой загрузкой процессов в потоки во время работы программы. По умолчанию оно `static`.

Два примера задания режимов загрузки типа `runtime` с помощью команд операционной системы Linux приведены ниже.

```
$ setenv OMP_SCHEDULE static,1000
$ setenv OMP_SCHEDULE dynamic
```

В первом примере через значение переменной окружения в качестве загрузки `runtime` задается режим `static` с размером порции равным `1000`. Во втором же примере в качестве загрузки `runtime` задается режим `dynamic` с размером порции, по умолчанию заданным равным

1.

# Дополнительные возможности OpenMP

В этой лекции рассматриваются дополнительные возможности, которыми можно воспользоваться при написании программ с применением OpenMP. Первый раздел настоящей лекции посвящен рассмотрению вопросов задания переменных окружения с помощью функций библиотеки реального времени runtime OpenMP. Во втором разделе рассматриваются вопросы применения директивы threadprivate при организации передачи данных в программах, написанных с использованием OpenMP. В третьем разделе лекции приводится устаревшая конструкция передачи данных в директиве OpenMP parallel do. Наконец, четвертый раздел настоящей лекции посвящен изучению вопросов блокировки в OpenMP.

## Задание переменных окружения с помощью функций runtime OpenMP

В предыдущих лекциях уже затрагивались вопросы задания переменных окружения, определяющих режимы работы параллельных частей программ. Было показано, что для задания их значений можно воспользоваться командами операционной системы Linux или директивами OpenMP. Однако в OpenMP существует еще одна возможность задания переменных окружения: с помощью функций библиотеки `runtime` OpenMP. Обращение к этим функциям осуществляется непосредственно в программе и ничем не отличается от вызова обычных функций. Однако следует иметь в виду, что подобная возможность существует не во всех реализациях OpenMP. Так, в реализациях OpenMP, поддерживаемых компанией Silicon Graphics, такая возможность отсутствует. Это же замечание относится и к реализациям OpenMP на многопроцессорных вычислительных системах G-Scale s-350 компании Kraftway.

В программах, написанных на C/C++ с использованием OpenMP, существуют следующие возможности задания переменных окружения с помощью средств библиотеки реального времени `runtime` OpenMP.

С помощью вызова функции

```
(void) omp_set_num_threads(int num_threads)
```

можно задать число потоков в области параллельных вычислений, т. е. определить значение переменной окружения `OMP_NUM_THREADS`. При завершении работы эта функция не возвращает в программу никаких значений.

Функция

```
int omp_get_num_threads()
```

напротив, возвращает в программу целочисленное значение, равное количеству параллельных потоков в текущий момент времени.

Следующая функция

```
int omp_get_max_threads()
```

возвращает в программу целочисленное значение, равное максимальному количеству параллельных потоков, которое может быть возвращено функцией `omp_get_num_threads`.

Для определения номера параллельного потока в текущий момент времени программы можно воспользоваться функцией

```
int_omp_get_thread_num()
```

Она возвращает целочисленное значение в диапазоне от `0` до `OMP_NUM_THREADS-1`.

Определить количество процессоров, доступных программе в текущей точке, можно с помощью функции

```
int omp_get_num_procs()
```

Следующая функция позволяет идентифицировать, в какой области программы (параллельной или последовательной) в текущий момент времени проводятся вычисления

```
(int/logical) omp_in_parallel()
```

Она возвращает значение TRUE или 1 в точке параллельной области программы и FALSE или 0 в точке последовательной области программы.

Задать или отменить динамический режим работы программы можно, воспользовавшись функцией

```
(void) omp_set_dynamic( TRUE | FALSE )
```

Для задания динамического режима следует использовать параметр TRUE, а для его отмены - параметр FALSE. При этом в списке переменных окружения определяется или удаляется переменная окружения OMP_DYNAMIC или задается ее значение, соответственно равное TRUE или FALSE.

Следующая функция позволяет идентифицировать, какой режим работы параллельной части программы (динамический или статический) осуществляется в текущий момент времени при вызове функции

```
(int/logical) omp_get_dynamic()
```

Эта функция возвращает значение TRUE или 1, если режим динамический, и FALSE или 0 в случае статического режима.

Задать или отменить вложенный режим параллельной обработки процессов в параллельной области программы можно с помощью функции

```
(void) omp_set_nested( TRUE | FALSE )
```

Она устанавливает или отменяет вложенный режим параллельной обработки. При этом в списке переменных окружения определяется или удаляется переменная окружения OMP_NESTED либо задается ее значение, соответственно равное TRUE или FALSE.

Следующая функция позволяет идентифицировать, установлен или нет вложенный режим параллельной обработки в момент вызова функции

```
(int/logical) omp_get_nested()
```

Эта функция возвращает значение `TRUE` или `1` для вложенного режима параллельной обработки и значение `FALSE` или `0` при отсутствии такового.

При обращении ко всем вышеперечисленным функциям в заголовки программ на языке C/C++ необходимо включать строку:

```
#include <omp.h>
```

В программах, написанных на языке Fortran, существуют одноименные аналогичные функции. При этом функции, возвращающие значения, реализованы в виде функций, а функции, устанавливающие значения переменных окружения и не возвращающие никаких значений, реализованы в виде подпрограмм. Тип функций в программах, написанных на языке Fortran, должен быть определен в соответствии с типом возвращаемых данных.

## Передача данных с помощью директивы threadprivate

Проблема передачи данных в параллельных программах, написанных с использованием OpenMP, уже рассматривалась в предыдущих разделах. Однако был изложен лишь вариант передачи (миграции) данных из главного потока программы в параллельные. В этом же разделе мы остановимся на проблеме передачи данных между параллельными потоками из одного параллельного структурного блока программы в другой, минуя промежуточный последовательный структурный блок. Для реализации такого механизма передачи данных в OpenMP имеется специальная директива `threadprivate`.

Директива `threadprivate` применяется к глобальным переменным программы, которые нужно сделать локальными переменными для каждого из параллельных потоков. Кроме того, эта директива позволяет передавать локальные данные из одного параллельного структурного блока в другой, а также сохранять локальные данные из параллельных потоков на протяжении всей программы. В программах, написанных на языке Fortran, для этого необходимо поместить переменные в common-блок, а затем описать этот блок с помощью директивы `threadprivate` следующим образом:

```
c$omp threadprivate (/cb1/, [/cb2/ [ , ..., /cbN/]]
```

Здесь `cb1, cb2, ..., cbN` - имена `common` -блоков, для которых создаются локальные копии во всех параллельных потоках. В результате с первой же параллельной области, следующей за директивой `threadprivate`, будут созданы локальные копии `common` -блоков. Важно отметить, что в процессе передачи данных сохраняется соответствие между значениями передаваемых переменных и номерами параллельных потоков.

Пример фрагмента программы, написанной на языке Fortran с использованием директивы OpenMP `threadprivate`, приведен в примере 4.1.

Как видно из этого примера, для передачи данных из одного параллельного структурного блока в другой в данном примере используется переменная `x`, определенная в common-блоке `mine`. Имя этого common-блока описано с помощью директивы `threadprivate`. В первом параллельном структурном блоке программы вычисляются и выводятся на печать значения переменной `x` в каждом из четырех параллельных потоков. После завершения первого параллельного блока значения этой переменной выводятся в следующем последовательном блоке программы. Отметим, что в этом случае напечатанное значение соответствует главному потоку программы (его индекс *`tid`* всегда равен `0` ), в который передает свое значение `x` первый параллельный поток с индексом *`tid`*, равным `0`.

Во втором параллельном структурном блоке значения переменной `x` не изменяются и выводятся на печать в точном соответствии с индексами параллельных потоков *`tid`*, установленных в первом параллельном блоке с индексом *`tid`*=`0`.

В программах, написанных на языке Fortran, предложение `threadprivate` имеет следующий вид:

```
program region
        integer, external :: omp_get_thread_num
        integer :: tid, x
        common/mine/ x
```

```
!$omp threadprivate (/mine/)
      call omp_set_num_threads (4)
!$omp parallel private (tid)
      tid = omp_get__thread_num ( )
      x = tid * 10 + 1
      print *,"T:", tid," inside first parallel region x =",x
!$omp end parallel
      print *,"T:", tid," outside parallel region x =",x
!$omp parallel private (tid)
      tid = omp_get_thread_num( )
      print *,"T:", tid," inside next parallel region x =",x
!$omp end parallel
    end program region
```

**Пример 4.1. Фрагмент программы на языке Fortran с использованием директивы OpenMP threadprivate**

Результаты работы программы:

```
T: 0 inside first parallel region x = 1
T: 1 inside first parallel region x = 11
T: 2 inside first parallel region x = 21
T: 3 inside first parallel region x = 31
T: 0 outside parallel region      x = 1
T: 0 inside next parallel region  x = 1
T: 2 inside next parallel region  x = 21
T: 3 inside next parallel region  x = 31
T: 1 inside next parallel region  x = 11
```

Здесь `cb1, cb2, ..., cbN` - список переменных, для которых создаются локальные копии во всех параллельных потоках.

Отметим, что элементами списка не могут быть ссылки или переменные, не полностью определенные. Кроме того, адреса констант также недопустимы в списке.

Все переменные, включенные в список директив `threadprivate`, должны быть проинициализированы до начала первого параллельного блока, и их инициализация возможна только один раз на протяжении всей программы. Кроме того, переменные, включенные в список, не

могут фигурировать в других предложениях OpenMP, за исключением `copyin`, `schedule` или `if`. Категорически запрещено их использование в директивах `private`, `firstprivate`, `lastprivate`, `shared` и *`reduction`*.

Не допускается применение передачи значений переменных из одного параллельного структурного блока в другой в динамическом ( `dynamic` ) режиме работы параллельной программы. На всем протяжении действия директивы `threadprivate` должен быть установлен статический режим работы параллельной программы.

В примере 4.2 приведен пример фрагмента параллельной программы, написанной на языке C/C++ с использованием директивы OpenMP `threadprivate`. В этом примере для отмены динамического режима явно вызывается функция из библиотеки реального времени `runtime` OpenMP `omp_set_dynamic` с параметром `0`.

```
#include <omp.h>

int alpha [10], beta[10], i

#pragma omp threadprivate (alpha)

main ( )
{
/*Выключение динамического режима*/

     omp_set_dynamic (0);

/*1-й параллельный блок */

#pragma omp parallel private (i, beta)
     for (i=0; i<10; i++)
       alpha[i] = beta[i] = i;

/*2-й параллельный блок*/

#pragma omp parallel
     printf ("alpha[3]=%d and beta [3] =%d\n", alpha[3], beta[3]);
```

```
}
```

**Пример 4.2. Фрагмент параллельной программы на языке C/C++ с использованием директивы OpenMP threadprivate**

## Устаревшая конструкция передачи данных в директиве parallel do в OpenMP

В OpenMP имеется возможность передачи данных из главного потока в параллельные и назад через вызов функции в параллельном режиме. Эта конструкция считается устаревшей и в настоящее время редко используется на практике. Однако далее для полноты изложения все же рассмотрим возможность применения этой конструкции в программах с OpenMP.

Пример параллельной программы, написанной на языке Fortran с использованием вызова функции для передачи данных в параллельные потоки, приведен в примере 4.3.

```
program orphan
        integer, parameter :: M=8
        integer, dimension(M) :: x
        integer :: myid, i
        common/global/ x, myid, i
        call omp_set_num_threads (4)
!$omp parallel shared (x)
        call work ( )
!$omp end parallel
        write (6, *) x
      end program orphan

      subroutine work ( )
        integer, parameter :: M=8
        integer, dimension (M) :: x
        integer, external :: omp_get_thread_num
        common/global/ x, myid, i
!$omp do private (i, myid) ordered
        do i = 1. M
          myid = omp_get_thread_num ( )
```

```
            write (6, *) "T:", myid, " i=", i
            x (i) = myid
          enddo
!$omp end do
        return
      end subroutine work
```

**Пример 4.3. Пример передачи данных в директиву parallel do в OpenMP с помощью вызова подпрограммы**

В этом примере в управляющей программе *orphan* определяется число параллельных потоков `OMP_NUM_THREADS=4`. Затем в параллельном режиме осуществляется обращение к подпрограмме `work`. В этой подпрограмме цикл `do i=1,M` при `M=8` выполняется в параллельном режиме так: 8 петель этого цикла распределяются в последовательном порядке согласно директиве orderedпо потокам и потоки выполняются в последовательном порядке. В конце программы в последовательном структурном блоке осуществляется вывод элементов массива `x`. Поэтому результат работы этой программы будет таким, как показано в примере 4.4.

```
T:      0 i=   1
T:      0 i=   2
T:      1 i=   3
T:      1 i=   4
T:      2 i=   5
T:      2 i=   6
T:      3 i=   7
T:      3 i=   8
 0     0  1  1   2   2
 3  3
```

**Пример 4.4. Результат работы программы, приведенной в примере 4.3**

Почему же такая конструкция редко применяется на практике? Дело в том, что здесь в параллельной области программы вызывается подпрограмма `work`, а обращение к подпрограмме имеет почти такую же трудоемкость, как и открытие параллельных потоков. Поэтому вместо вызова подпрограмм или функций желательно подставлять их

текст в определенное место программы, т. е. делать `inline` - подстановку. В противном случае зачастую можно получить неэффективную параллельную программу.

## Функции блокировки в OpenMP

Функции блокировки играют очень важную роль при написании параллельных программ. Для последовательных программ необходимость функций блокировки обусловлена в основном многопользовательским режимом, когда на одном процессоре одновременно могут выполняться несколько заданий различных пользователей. В такой ситуации очень важно обеспечить корректный доступ к данным и их корректную модификацию со стороны различных задач или пользователей. В параллельном программировании даже одно задание порождает несколько параллельных процессов, каждый из которых может оперировать с одними и теми же данными. Для обеспечения корректности доступа к данным и их корректной модификации в OpenMP существуют специальные функции блокировки, обеспечивающие решение задач корректности доступа, записи и обновления данных.

Сначала рассмотрим функции блокировки в OpenMP в программах, написанных на языке C/C++.

Функция

```
void omp_init_lock ( omp_lock_t *lock )
```

предназначена для инициализации блокировки объекта с указателем `lock`. Она не возвращает никаких значений.

Следующая функция

```
void omp_destroy_lock ( omp_lock_t *lock )
```

гарантирует, что объект с указателем `lock` в данный момент не инициализирован.

С помощью функции

```
void omp_set_lock ( omp_lock_t *lock )
```

можно организовать блокировку выполнения потока до тех пор, пока объект открыт для операций чтения-записи. После завершения операций чтения-записи объект разблокируется и продолжается выполнение потока.

Для открытия заблокированного объекта с указателем `lock` можно воспользоваться функцией

```
void omp_unset_lock ( omp_lock_t *lock )
```

Функция

```
int omp_test_lock ( omp_lock_t *lock )
```

позволяет попытаться заблокировать объект, не прерывая выполнения потока. Она возвращает значение `TRUE` или `1`, если попытка завершилась успешно. В противном случае возвращается значение `FALSE` или `0`.

Напомним, что при обращении к функциям блокировки в заголовке текста программы должно содержаться включение описания этих функций:

```
#include < omp.h >
```

В примере 4.5 приведен пример фрагмента программы на языке C/C++, иллюстрирующий применение функций блокировки.

```
#include <omp.h>
void main ( )
{
    omp_lock_t lock;
    int myid;
    omp_init_lock (&lock);
#pragma omp parallel shared(lock) private(myid)
    {
      myid = omp_get_thread_num ( );
      omp_set_lock (&lock);
```

```
        printf ("Hello from thread %d\n", myid);
        omp_unset_lock (&lock);
        while (! omp_test_lock (&lock))
        {
           skip (myid);
        }
        do_work (myid);
        omp_unset_lock (&lock);
      }
      omp_destroy_lock (&lock);
}
```

**Пример 4.5. Пример фрагмента программы на языке C/C++, иллюстрирующий применение функций блокировки**

В программах, написанных на языке Fortran, переменная `lock` должна быть целочисленной типа `KIND` и иметь размерность, достаточную для записи адреса.

Подпрограмма

```
subroutine omp_init_lock ( omp_lock_t lock )
```

как и в языке C/C++, используется для инициализации блокировки объекта с указателем `lock`.

Следующая подпрограмма

```
subroutine omp_destroy_lock ( omp_lock_t lock )
```

обеспечивает отсутствие блокировки объекта с указателем `lock` в данный момент времени.

Подпрограмма

```
subroutine omp_set_lock ( omp_lock_t lock )
```

устанавливает блокировку выполнения потока до тех пор, пока объект открыт для операций чтения-записи. После завершения операций чтениязаписи объект блокируется и продолжается выполнение потока.

С помощью подпрограммы можно открыть заблокированный объект с указателем `lock`:

```
subroutine omp_unset_lock ( omp_lock_t lock )
```

Функция

```
logical omp_test_lock ( omp_lock_t *lock )
```

как и в языке C/C++, позволяет попытаться заблокировать объект, не прерывая выполнения потока. Она возвращает значение `TRUE`, если попытка завершилась успешно. В противном случае возвращается значение `FALSE`.

# Отладка программ в OpenMP

В этой лекции рассматривается ряд проблем, возникающих при отладке параллельных программ в OpenMP. Первый раздел настоящей лекции посвящен рассмотрению условий состязательности. Во втором разделе изложена проблема мертвой блокировки. И, наконец, в третьем разделе приведены средства автоматизированной отладки в OpenMP.

При написании и отладке программ в OpenMP необходимо обеспечить выполнение целого ряда специфических требований, без которых невозможно гарантировать корректность работы программы. Далее рассмотрим эти требования подробнее.

Во-первых, необходимо обеспечить, чтобы все используемые в программах библиотеки были безопасны с точки зрения потокового выполнения. При этом следует иметь в виду, что все стандартные библиотеки всегда удовлетворяют этому требованию. Таким образом, важно обеспечить выполнение этого требования только по отношению к тем библиотекам, которые создает сам разработчик или его коллеги.

Разрабатывая и отлаживая программы в OpenMP, необходимо помнить, что операции ввода/вывода в параллельных потоках завершаются в непредсказуемом порядке. Поэтому большое внимание следует уделять проблеме синхронизации параллельных процессов, помня при этом, что синхронизация - очень дорогая операция. Поэтому ею следует пользоваться как можно реже и только в случае крайней необходимости.

При разработке программ с использованием OpenMP следует внимательно прослеживать пересечение локальных переменных типа `private` в параллельных потоках с глобальными переменными. Кроме того, необходимо тщательно отслеживать своевременное обновление значений переменных в общей памяти. Для этого по мере необходимости следует пользоваться директивой OpenMP `flush`.

Проектируя программы, следует помнить, что удалить неявно установленные барьеры ( *`barrier`* ) в операторах цикла можно с помощью предложения OpenMP *`nowait`*.

## Условия состязательности

При работе программ с общей памятью возникает ряд специфических ошибок, одна из которых связана с так называемыми условиями состязательности ( *race conditions* ). Ошибки, связанные с условиями состязательности, состоят в непредсказуемом времени завершения параллельных потоков, из-за чего возможны непредсказуемые результаты вычислений. Автоматическое распараллеливание в таких ситуациях затруднено, поскольку компилятору либо трудно, либо вообще невозможно распознать взаимозависимости по данным, возникающие в процессе параллельных вычислений при непредсказуемом времени завершения работы параллельных потоков. В таких ситуациях важно правильно определить последовательность выполнения параллельных потоков, чтобы избежать непредсказуемых результатов.

В качестве примера возникновения условия состязательности рассмотрим фрагмент программы, показанный в примере 5.1.

```
c$omp parallel sections
      A = B + C
c$omp section
      B = A + C
c$omp section
      C = B + A
c$omp end parallel sections
```

**Пример 5.1. Фрагмент параллельной программы с возникновением условия состязательности**

Результат вычислений в этом примере будет зависеть от того, в какой последовательности выполняются параллельные потоки. При этом никакой диагностики компилятора о возникновении условия состязательности в программе не выдается. Это существенно усложняет отладку параллельных приложений, если программист заранее не позаботится об исключении таких ситуаций. Устранить некорректность в вычислениях в рассматриваемом примере можно, например, с помощью синхронизации параллельных потоков или с помощью директив загрузки параллельных процессов в OpenMP. Ниже в примере 5.2 приведен пример одной из возможных модификаций фрагмента программы (пример 5.1), в котором устранены недостатки, связанные с наличием условия состязательности. В этой модификации для

предотвращения условия состязательности введен счетчик событий ICOUNT и использована директива OpenMP FLUSH для организации корректного обращения к данным. Счетчик событий ICOUNT устанавливает последовательность вычислений: сначала вычисляется A=B+C, потом B=A+C и лишь затем C=B+A. Использование сочетания операторов IF и GOTO в сочетании с директивой OpenMP FLUSH позволило в этом примере провести синхронизацию трех параллельных процессов.

Еще один пример возникновения условия состязательности в программе представлен во фрагменте в примере 5.3. В этом примере условие состязательности возникает из-за неправильного в данном случае использования предложения OpenMP *NOWAIT*. Для корректной работы программы достаточно просто исключить предложение *NOWAIT*.

```
ICOUNT = 0
c$omp parallel sections
      A = B + C
      ICOUNT = 1
c$omp flush ICOUNT
c$omp section
 1000 CONTINUE
c$omp flush ICOUNT
      IF (ICOUNT .LT. 1)GOTO 1000
      B = A + C
      ICOUNT = 2
c$omp flush ICOUNT
c$omp section
 2000 CONTINUE
c$omp flush ICOUNT
      IF (ICOUNT .LT. 2)GOTO 2000
      C = B + A
c$omp end parallel sections
```

**Пример 5.2. Пример фрагмента параллельной программы с устранением условия состязательности**

```
c$omp parallel shared (X)
c$omp& private (TMP)
```

```
      ID=omp_get_thread_num ()
c$omp do reduction (+ : X)
      DO 100 I = 1, 100
      TMP = WORK(I)
      X = X + TMP
 100  CONTINUE
c$omp end do nowait
      Y( ID ) = X
c$omp end parallel
```

**Пример 5.3. Пример возникновения условия состязательности в программе из-за неправильного использования директивы NOWAIT**

В примере 5.4 условия состязательности можно избежать, описав переменную `TMP` в параллельной области как `private`.

```
REAL TMP, X
c$omp parallel do reduction (+ : X)
      DO 100 I = 1, 100
      TMP = WORK(I)
      X = X + TMP
  100 CONTINUE
c$omp end do
      Y(ID) = X
c$omp end parallel
```

**Пример 5.4. Пример возникновения условия состязательности в программе из-за отсутствия описания переменной TMP как private**

## Мертвая блокировка

В процессе выполнения программ с общей памятью возможна ситуация, когда один из параллельных потоков ожидает освобождения доступа к объекту, который никогда не будет открыт. Такая ситуация, возникающая в параллельной программе, называется мертвой блокировкой ( `deadlock` ).

В качестве иллюстрации мертвой блокировки рассмотрим фрагмент программы на языке Fortran, приведенный в примере 5.5.

```
call omp_init_lock (lcka)
    call omp_init_lock (lckb)
c$omp parallel sections
    call omp_set_lock (lcka)
    call omp_set_lock (lckb)
    call useAandB (res)
    call omp_unset_lock (lckb)
    call omp_unset_lock (lcka)
c$omp section
    call omp_set_lock (lckb)
    call omp_set_lock (lcka)
    call useBandA (res)
    call omp_unset_lock (lcka)
    call omp_unset_lock (lckb)
c$omp end parallel sections
```

**Пример 5.5. Пример возникновения мертвой блокировки в параллельной программе**

В этом примере при выполнении параллельных потоков возможна ситуация, когда объект A заблокирован в одном параллельном потоке, а объект B - в другом. В результате возникает мертвая блокировка и выполнение подпрограмм `useAandB` и `useBandA` не происходит. Однако это не единственная неприятная ситуация, возникающая в этой программе.

Возможно, что при выполнении параллельных потоков в одном потоке окажутся заблокированными оба объекта `A` и `B`, тогда при выполнении рассматриваемой программы возникает условие состязательности, рассмотренное в предыдущем разделе данной лекции. Для исправления возникшей в программе ошибочной ситуации следует избавиться от различных вложенных блокировок.

Рассмотрим еще один пример мертвой блокировки, который возможен при работе программы, показанной в примере 5.6.

```
call omp_init_lock (LCKA)
c$omp parallel sections
c$omp section
    call omp_set_lock (LCKA)
```

```
      IVAL = dowork ()
      if (IVAL .EQ. TOL) then
        call omp_unset_lock (LCKA)
      else
        call error (IVAL)
      endif
c$omp section
      call omp_set_lock (LCKA)
      call use_B_and_A (RES)
      call omp_unset_lock (LCKA)
c$omp end sections
```

**Пример 5.6. Пример возникновения мертвой блокировки в параллельной программе**

В этом примере при выполнении параллельных потоков возможно возникновение как мертвой блокировки, так и условия состязательности. Мертвая блокировка возникает, если в первом параллельном потоке установлена блокировка объекта `A`, а `IVAL` не равно `TOL`. В этом случае установить блокировку объекта `A` во втором параллельном потоке невозможно, поскольку программа ждет отмены блокировки, установленной в первом параллельном потоке.

Если же в первом параллельном потоке установлена блокировка объекта `A`, а `IVAL` равно `TOL`, то в программе возникает условие состязательности.

Для исправления возникшей ситуации необходимо устранить не только мертвую блокировку, но и условия состязательности.

## Средства автоматизированной отладки в OpenMP

Актуальность развития многопоточных приложений обусловлена быстрым развитием и все более широким распространением микропроцессоров, реализующих многоядерные (multicore) и многопоточные (hyper-threading) технологии.

Поскольку многопоточная программа существенно сложнее однопоточной, вести отладку таких приложений весьма и весьма

непросто. Применение обычных (стандартных) отладчиков не может решить и малой доли проблем, возникающих при отладке многопоточных приложений. Поэтому становится весьма актуальной задача создания автоматизированных средств отладки с наглядным графическим интерфейсом. Одним из таких средств является разработка компании Intel - программа Intel Thread Checker. Эта программа имеет развитый графический интерфейс и позволяет быстро и эффективно решать основные проблемы отладки многопоточных программ, находя и устраняя сложные, невидимые на первый взгляд ошибки.

С помощью программы Intel Thread Checker разработчик может в автоматическом режиме осуществить поиск ошибок в многопоточной программе, проанализировать условия доступа к данным в параллельных потоках, выявить тупиковые ситуации в программе и установить причины зависания потоков. Кроме того, с помощью этого инструмента разработчик может провести модернизацию параллельной программы, руководствуясь подсказками программы Intel Thread Checker.

Программа Intel Thread Checker работает под управлением операционных систем Windows и Linux и позволяет вести отладку с Windows-компьютера на удаленных Linux-компьютерах.

В процессе анализа программы разработчика программа Intel Thread Checker находит строки текста с ошибками и отмечает сами ошибки. Кроме того, программа Intel Thread Checker находит некорректно используемые переменные и их описания. Справочная система программы не только отображает причины найденных ошибок, но и предлагает возможные пути их устранения.

Программа Intel Thread Checker под управлением операционной системы Windows работает в среде разработки с любыми компиляторами компаний Intel и Microsoft. Кроме того, Intel Thread Checker работает в среде разработки Microsoft Visual Studio .NET и позволяет анализировать программы, оттранслированные с помощью следующих версий компиляторов Microsoft: Visual C++ .NET Compiler версий 2002 или 2003, а также Visual C++ Compiler 6.0. Программа поддерживает анализ стека вызовов, функций интерфейса Win32 API, исполняемых библиотек языка C, директив OpenMP. Таким образом, анализ охватывает большинство проблем, с которыми на практике сталкивается

разработчик многопоточных программ.

Программа Intel Thread Checker позволяет выводить информацию об ошибках с различными уровнями детализации. Всего в программе имеется шесть различных уровней детализации сообщений об ошибках: от ошибок и предупреждений до содержательных комментариев. Все возникающие ошибки можно классифицировать по категориям, присвоив им определенный статус. Список ошибок можно отсортировать различными способами, например по степени их важности. Затем можно установить причины ошибок и даже получить рекомендации по их устранению. Все это реализовано в удобном графическом режиме.

Программа Intel Thread Checker позволяет не только находить, анализировать и исправлять ошибки в многопоточных параллельных программах, но и анализировать полный ход выполнения программы, находя и выделяя критические пути. Последующий анализ критического пути выполнения многопоточной программы позволяет установить ее узкие места ( `bottle neck` ), устранить их и тем самым повысить быстродействие программы.

Программа Intel Thread Checker полностью совместима с технологией OpenMP. Однако при этом она может обнаруживать и ошибки в однопоточной версии программы. Для отображения дополнительных сведений об указателях при запуске компилятора следует использовать настройку `/Qtcheck` компилятора Intel.

Если режим OpenMP не используется, программа Intel Thread Checker распознает практически все многопоточные версии стандартных функций Win32 API и языка C/C++. Так, программа определяет время создания, временной остановки и время прерывания потоков. Кроме того, она выделяет подпотоки ( `fiber` ), критические участки ( `critical paths` ), функции событий, сблокированные (атомарные) функции, объекты блокировки ( *`mutex`* ), объекты синхронизации, пулы потоков, функции синхронизации и другие.

На рис.5.1 схематически изображена поэтапная схема проектирования параллельной программы с перечислением инструментов, которые могут быть применены на каждом из этапов.

Как видно из этой схемы, на этапе разработки параллельной программы в качестве средств разработки могут использоваться компиляторы Intel и программа Intel VTune Performance Analyzer; на этапе тестирования и отладки - программа Intel Thread Checker и тесты разработчика; на этапе настройки - программы Intel VTune Performance Analyzer и Intel Thread Profiler. Отметим, что программы Intel Thread Checker, Intel VTune Performance Analyzer и Intel Thread Profiler входят в состав пакета программ Intel Threading Tools. Программа Intel VTune Performance Analyzer является инструментом настройки производительности программ, а программа Intel Thread Profiler - инструментом настройки работы параллельных потоков.

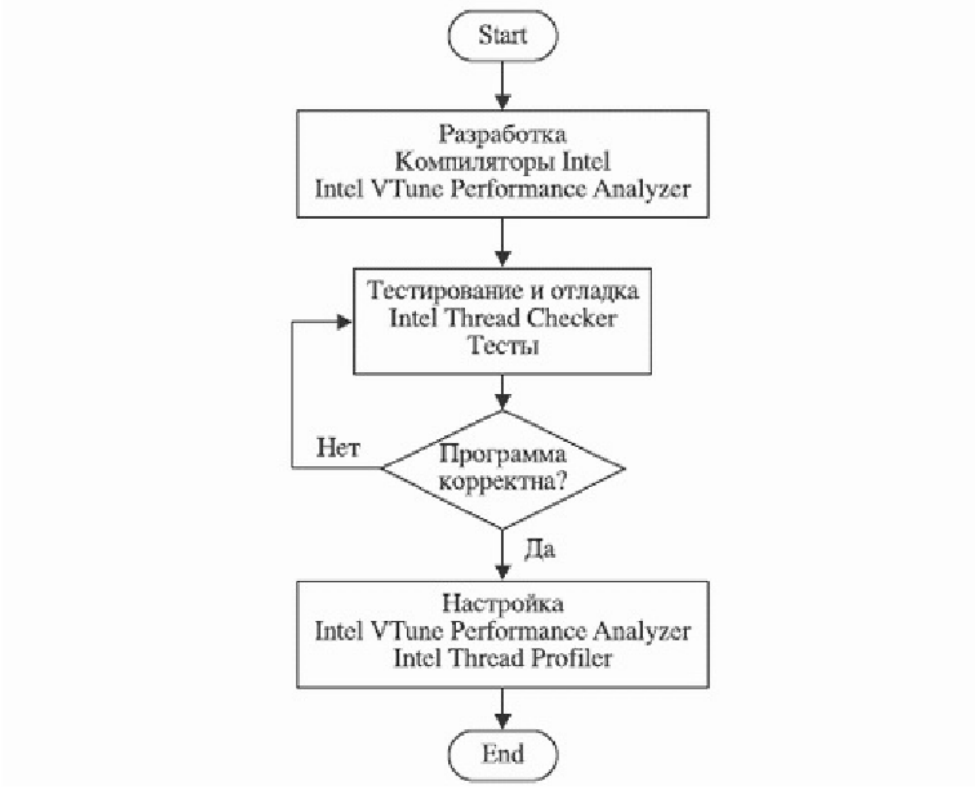



Рис. 5.1. Поэтапная схема проектирования параллельной программы и средства разработки, используемые на этапах

На рис.5.2 изображено окно запуска и настройки программы Intel Thread Profiler.

После запуска в программе Thread Profiler можно выбрать один из четырех видов графического представления информации:

1. Overview (общий вид);
2. Bar Chart (диаграмма полос);
3. Speedup Plot (диаграмма ускорения);
4. Legend (условные обозначения).

Определив `Activity` (различные реализации программы), можно перейти к анализу графической информации, соответствующей анализируемой версии программы.

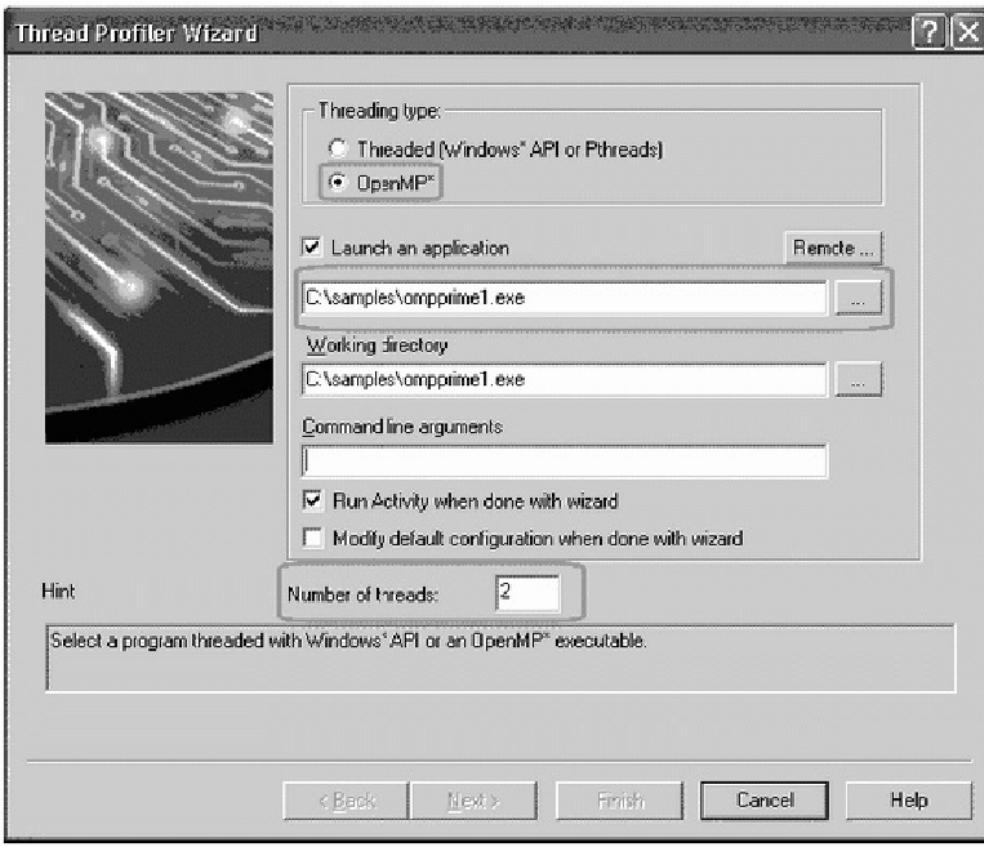



Рис. 5.2. Открытие проекта и выбор настроек в программе Intel Thread Profiler

На рис.5.3 показан экран общего вида путеводителя по проекту в

программе Thread Profiler.

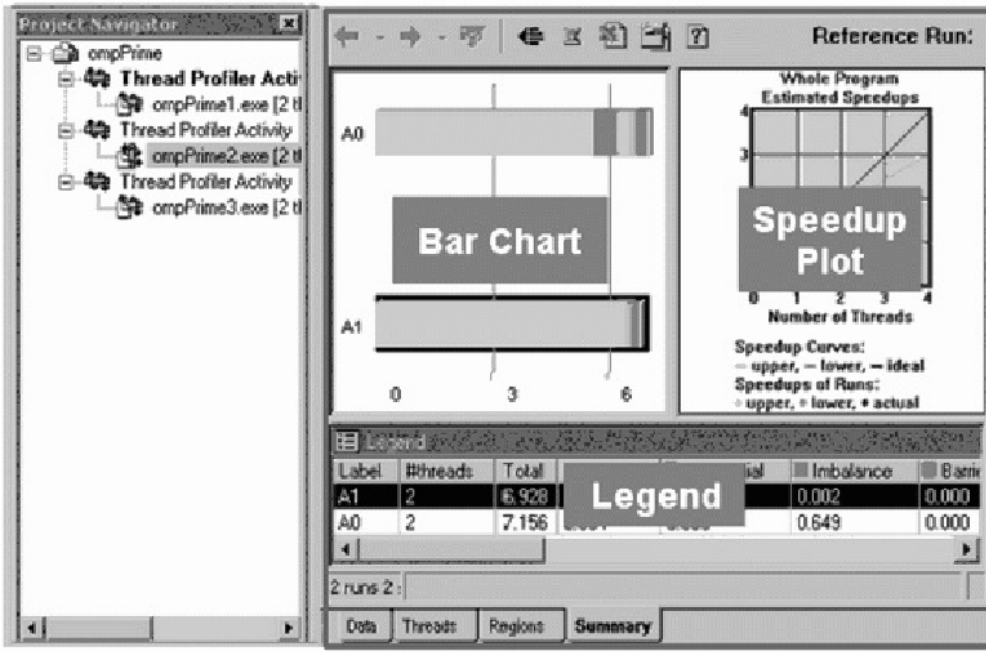



Рис. 5.3. Экран общего вида путеводителя по проекту в программе Intel Thread Profiler

На этом экране в сжатой форме представлены три основных окна программы: Bar Chart (диаграмма полос), Speedup Plot (диаграмма ускорения) и Legend (условные обозначения). Закладки Data, Threads, Regions позволяют осуществить более подробный анализ работы исследуемой программы.

На рис.5.4 представлен подробный анализ диаграммы полос (Bar Chart) для исходной (A0) и модифицированной (A1) версий исследуемой программы в Intel Thread Profiler.

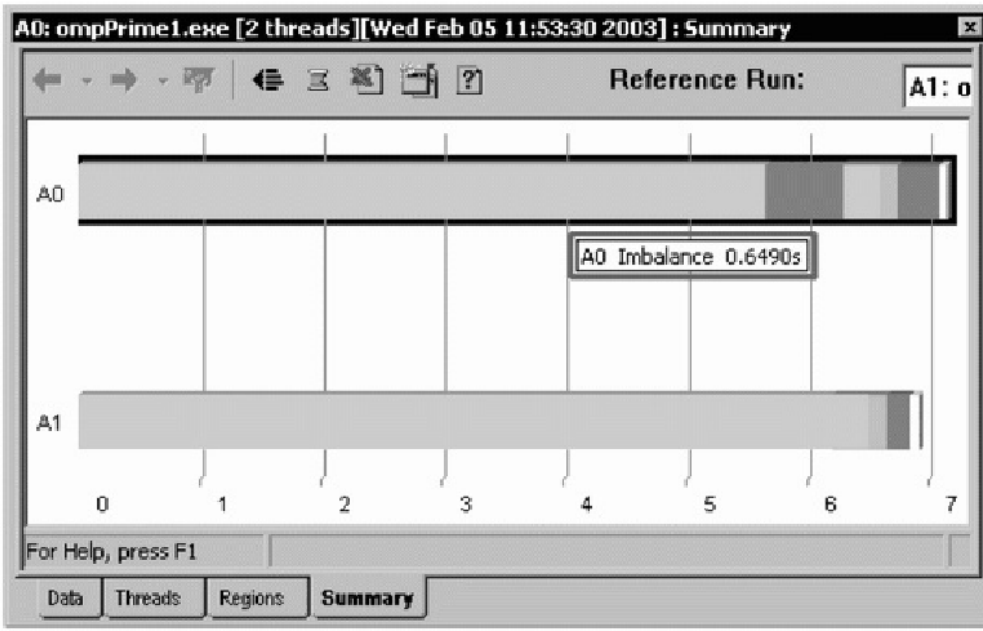



Рис. 5.4. Анализ диаграммы для исходной (A0) и модифицированной (A1) версий исследуемой программы в Intel Thread Profiler

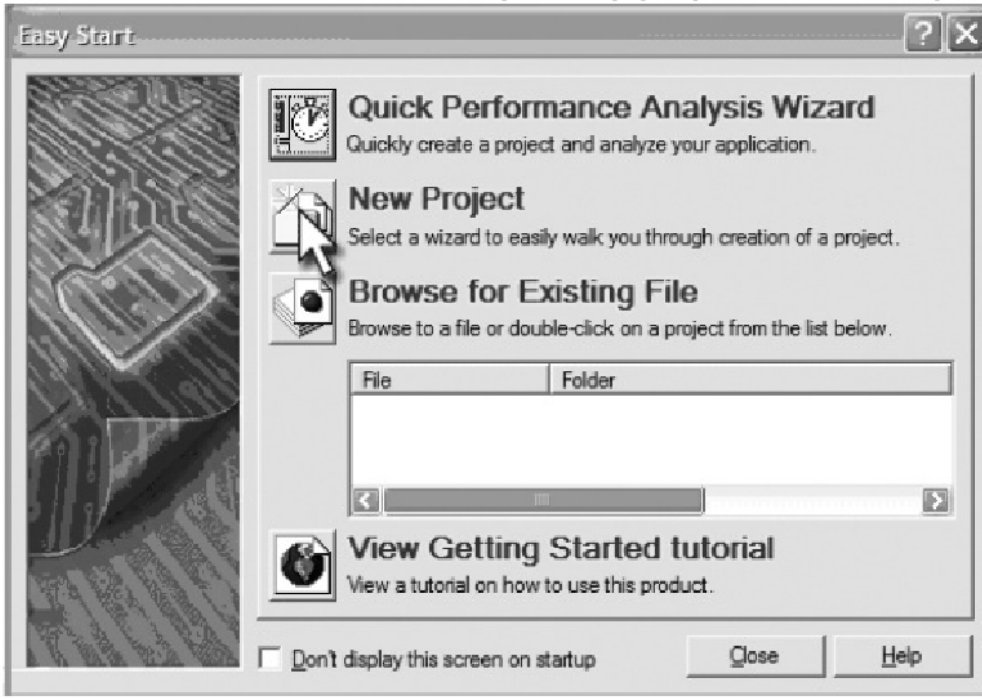
Как видно из этого рисунка, в исходной версии программы установлен дисбаланс потоков, устранить который можно после модификации программы.

Теперь рассмотрим основные моменты запуска и работы с программой Intel Thread Checker. На рис.5.5 показаны этапы создания нового проекта.

После запуска программы Intel Thread Checker на экране появляются окна с результатами анализа исследуемой программы (см. рис.5.6). Нажав на красный флажок в верхней строке меню, увидим следующую картину:

Эта картина иллюстрирует анализ загрузки памяти, полученный с помощью программы Intel Thread Checker. Щелкнув левой клавишей мыши на одной из функций в графе Context, можно открыть окно с ассемблерным текстом выбранной функции, а также с текстом на языках C/C++ или Fortran. Это окно показано на рис.5.7. Далее можно выбирать различные режимы просмотра и анализа процессов в

параллельной программе. В качестве примера на рис.5.8 приведена трассировка стека в анализируемой программе.



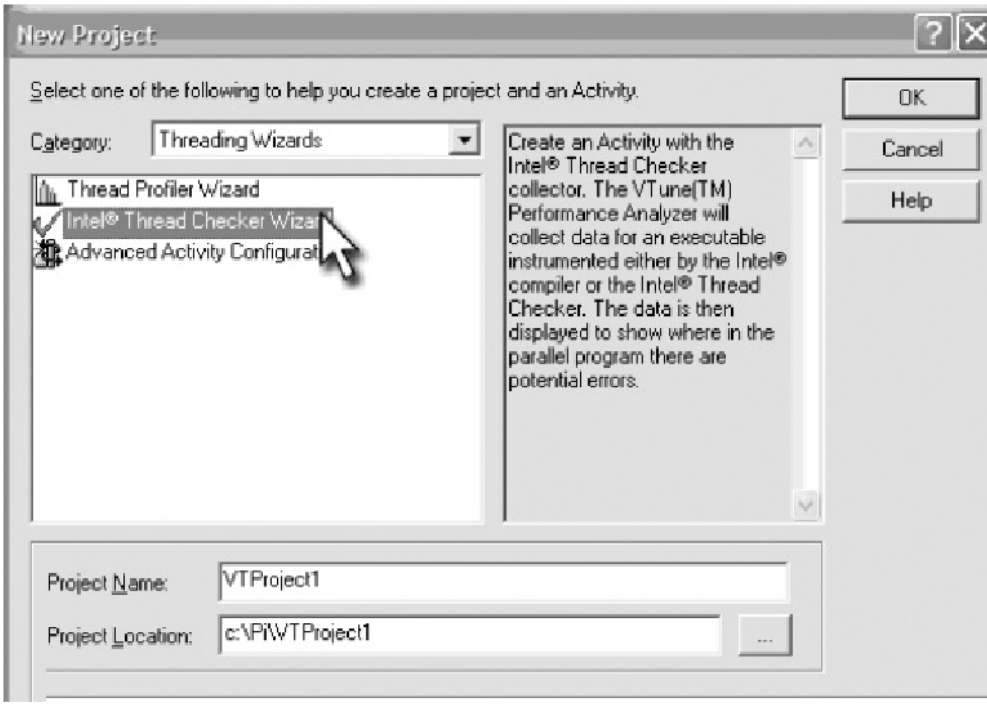




Рис. 5.5. Создание нового проекта в программе Intel Thread Checker

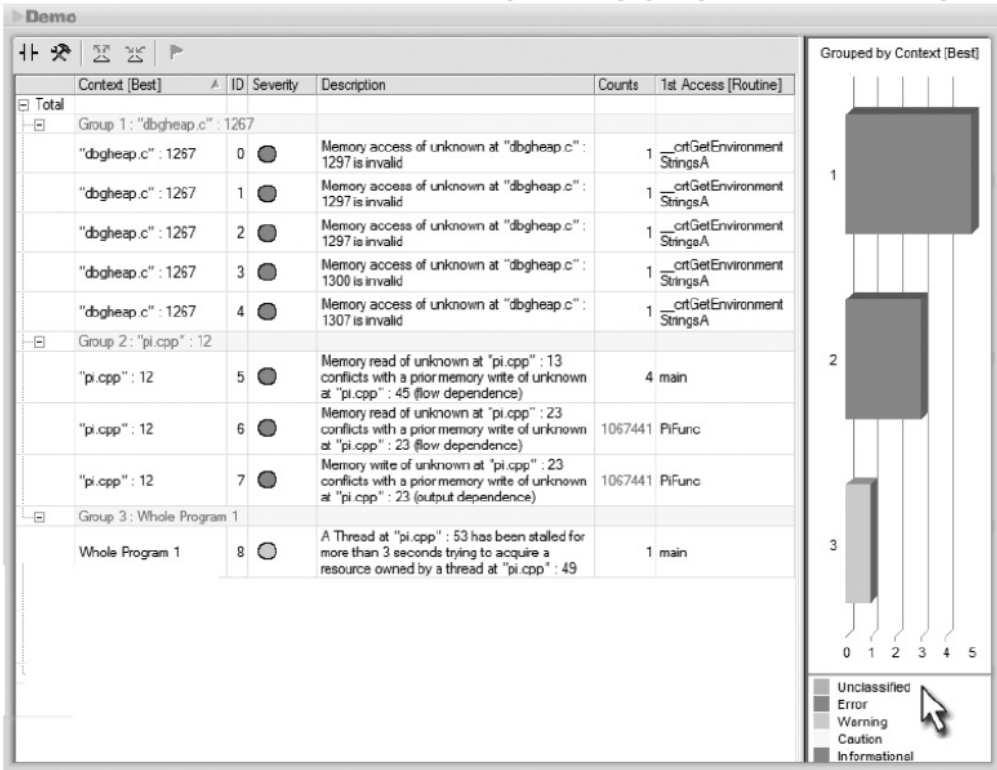



Рис. 5.6. Анализ загрузки памяти в программе Intel Thread Checker

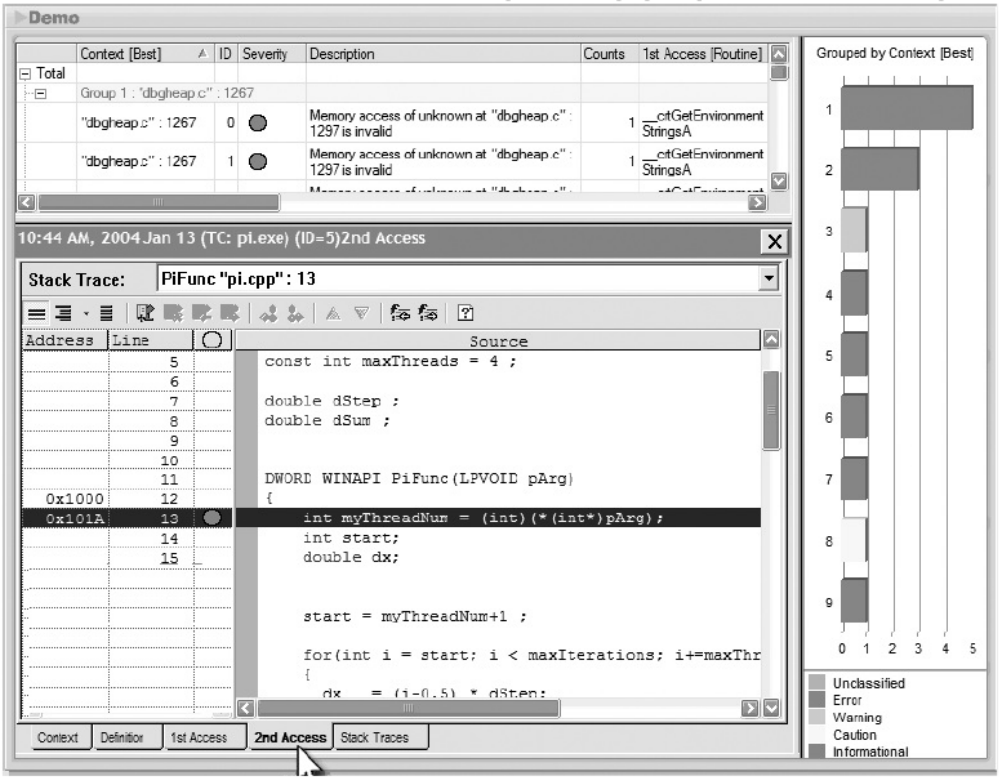



Рис. 5.7. Анализ загрузки памяти в программе Intel Thread Checker

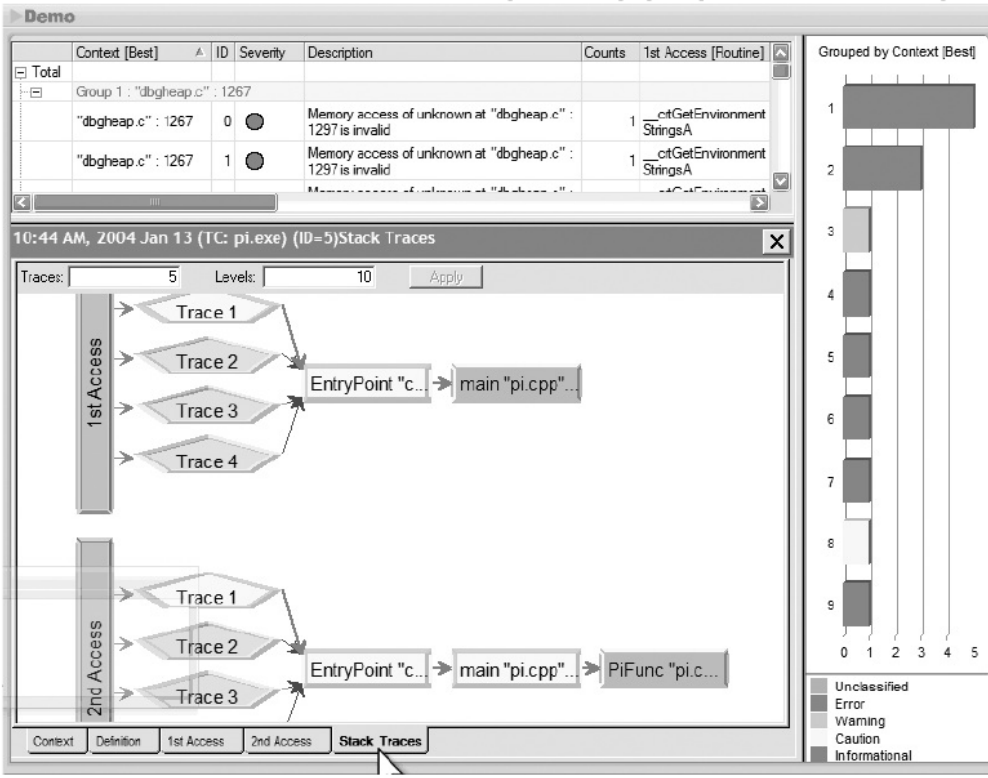



Рис. 5.8. Анализ трассировки стека в программе Intel Thread Checker

В заключение этого раздела скажем, что интерфейс программы Intel VTune Performance Analyzer аналогичен уже рассмотренному интерфейсу программы Intel Thread Profiler. Также отметим, что были рассмотрены лишь интерфейсы программ, работающих в операционной системе Windows. Для аналогичных программ, работающих в операционной системе Linux, пока не разработано единой графической оболочки. Запуск этих программ в операционной системе Linux в настоящее время осуществляется из командной строки с помощью специальных настроек. Результаты работы программ записываются в специальный раздел на жестком диске, и лишь затем можно запустить программы для просмотра в графическом режиме полученных результатов.

## Отладка многопоточных программ с помощью отладчика TotalView

Наличие современных автоматизированных средств отладки многопоточных программ не устраняет необходимости использования в процессе разработки и обычных средств отладки - сервисных программ-отладчиков. Однако стандартные отладчики ddd, idb и т. п. не очень удобны для многопоточных приложений, поскольку разрабатывались в основном для приложений однопоточных. На рынке системного программного обеспечения существуют и специальные отладчики, предназначенные для отладки многопоточных приложений. К ним как раз и относится отладчик TotalView, разрабатываемый и поддерживаемый компанией Etnus. Он ориентирован в первую очередь на анализ и настройку многопоточных параллельных программ. Отладчик TotalView - коммерческий продукт, и для работы с ним нужно приобрести лицензию. Однако существует и его бесплатная учебная версия, позволяющая отлаживать параллельные программы, в которых реализуется до 8 параллельных потоков. Далее подробнее остановимся на возможностях этого отладчика.

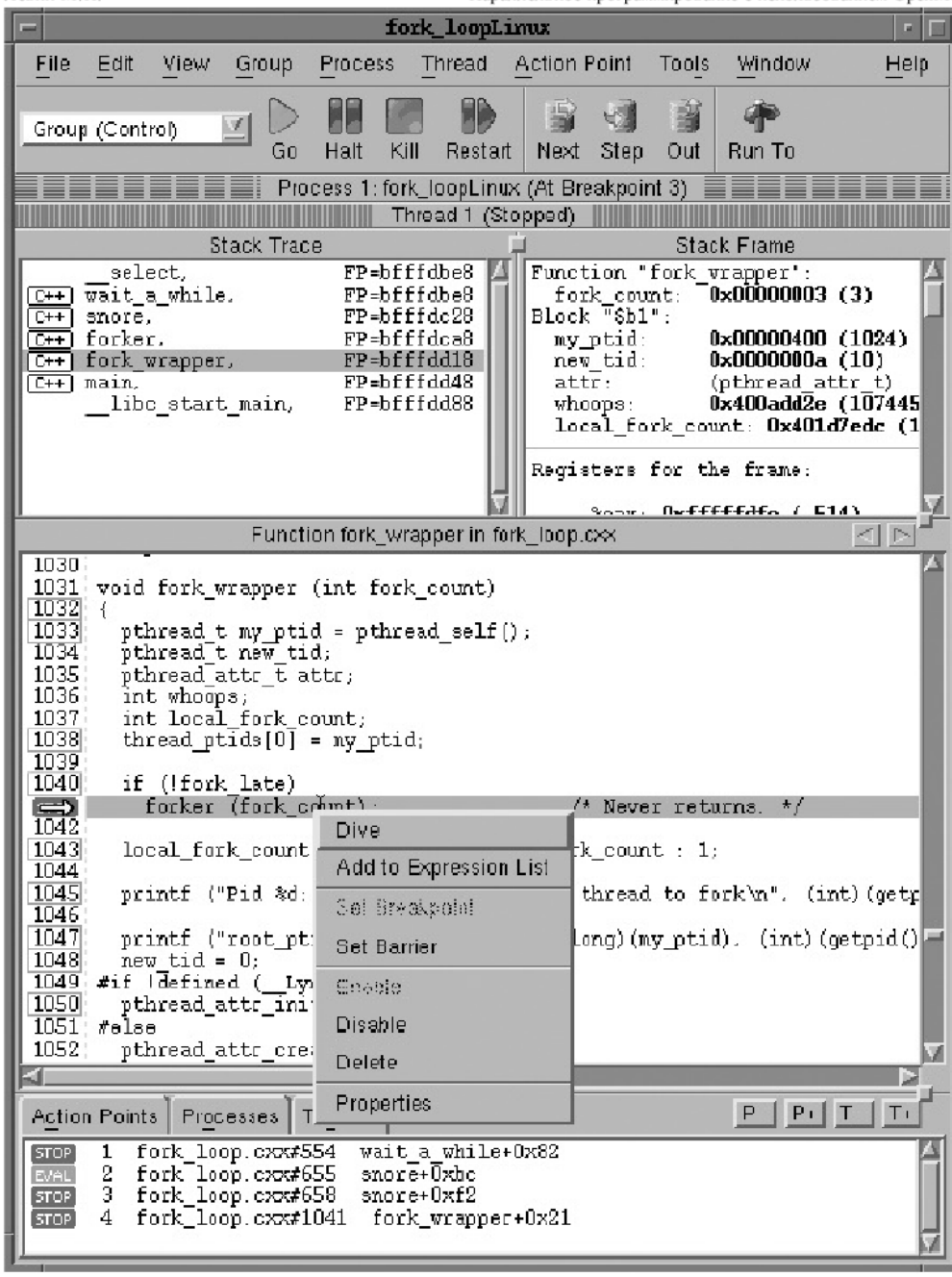



Рис. 5.9. Основное окно отладчика TotalView

После запуска отладчика из командной строки операционной системы Linux на экране компьютера появляется основное окно программы,

изображенное на рис.5.9.

Загрузить программу для отладки можно в меню File. После загрузки можно начать отладку, например, воспользовавшись командой Step, которую можно найти на второй сверху панели инструментов. Кроме того, можно в окне просмотра текста программы установить курсор на нужную строку и нажать на кнопку Run To на панели инструментов отладчика TotalView. В результате программа выполнится до той строки, на которой установлен курсор. Еще один способ запуска программы под управлением отладчика состоит в расстановке точек прерывания (Break Points). Для установки точки прерывания надо щелкнуть левой кнопкой мыши на номере строки в окне просмотра программы. В результате в окне просмотра вместо номера строки появится желтая стрелка на красном фоне (обозначение точки прерывания). Действуя аналогичным образом, можно расставить точки прерывания по всей отлаживаемой программе. Затем, нажимая на кнопку Go на панели инструментов, можно пройти по всей программе с остановками в точках прерывания.

Отметим, что в этом диалоговом окне можно определять не только функции, но и любые вычисления с переменными программы, определенными в точке активности. При этом в диалоговом окне надо задать алгоритмический язык, на котором написана программа, нажав соответствующую кнопку выбора (Check Box), а также активировать кнопку Evaluate.

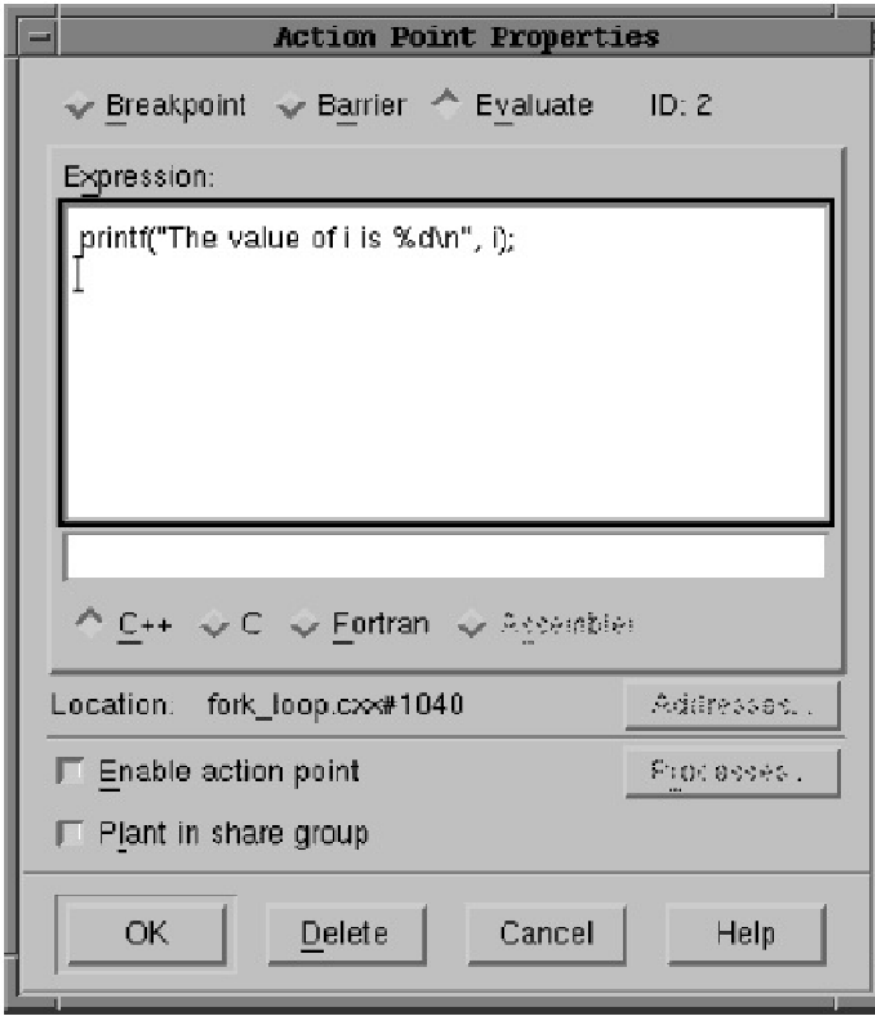



Рис. 5.10. Диалог определения функции print в точке активности

Просматривать значения переменных в точке прерывания можно в окнах Stack Trace и Stack Frame в основном окне отладчика, изображенном на рис.5.10. В нижнем фрагменте этого окна есть возможности для идентификации параллельного потока, в котором просматриваются переменные, а также средства перелистывания потоков. Щелчки левой кнопки мыши на данных в окне Stack Frame инициируют открытие всплывающих окон со значениями исследуемых переменных отлаживаемой программы, как показано на рис.5.11.

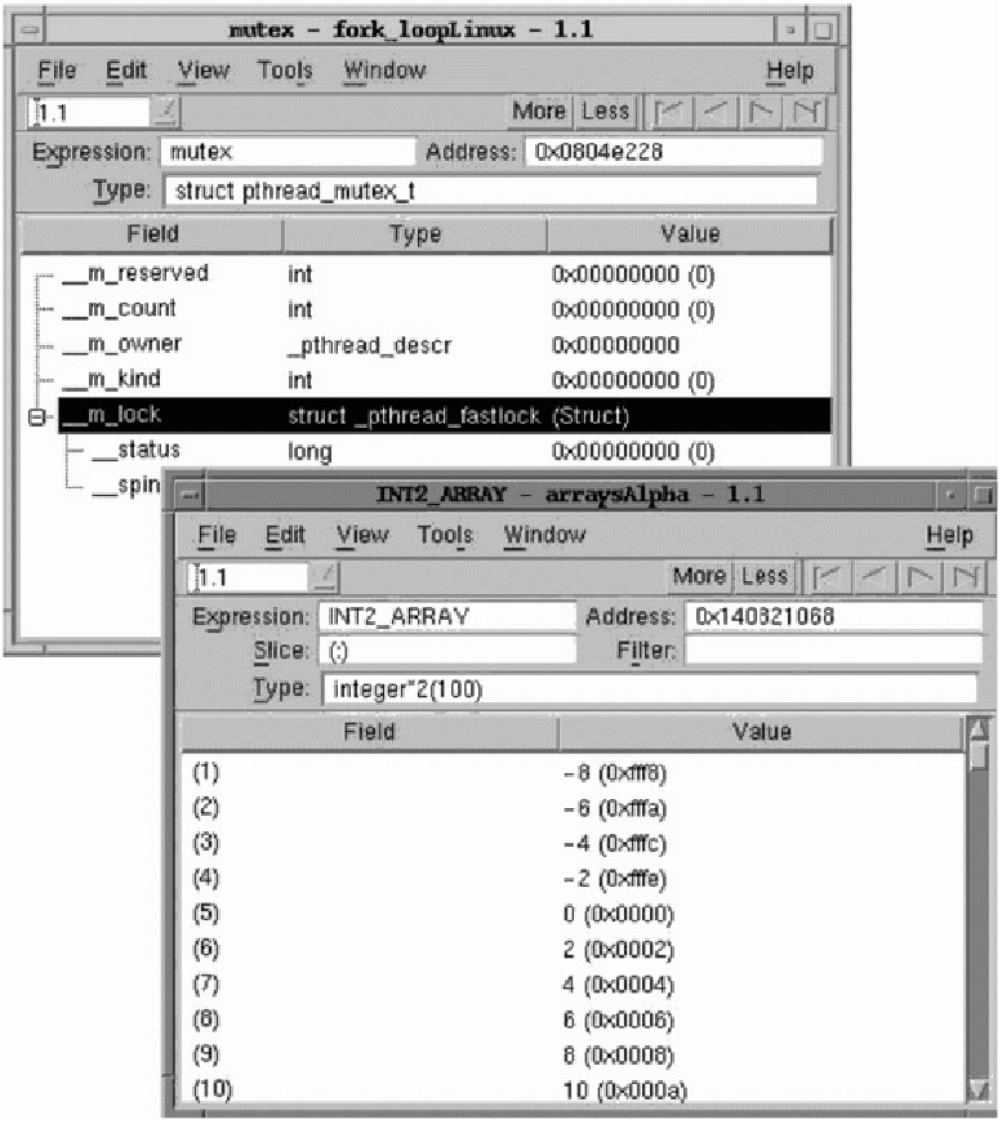



Рис. 5.11. Диалог просмотра значений переменных в TotalView

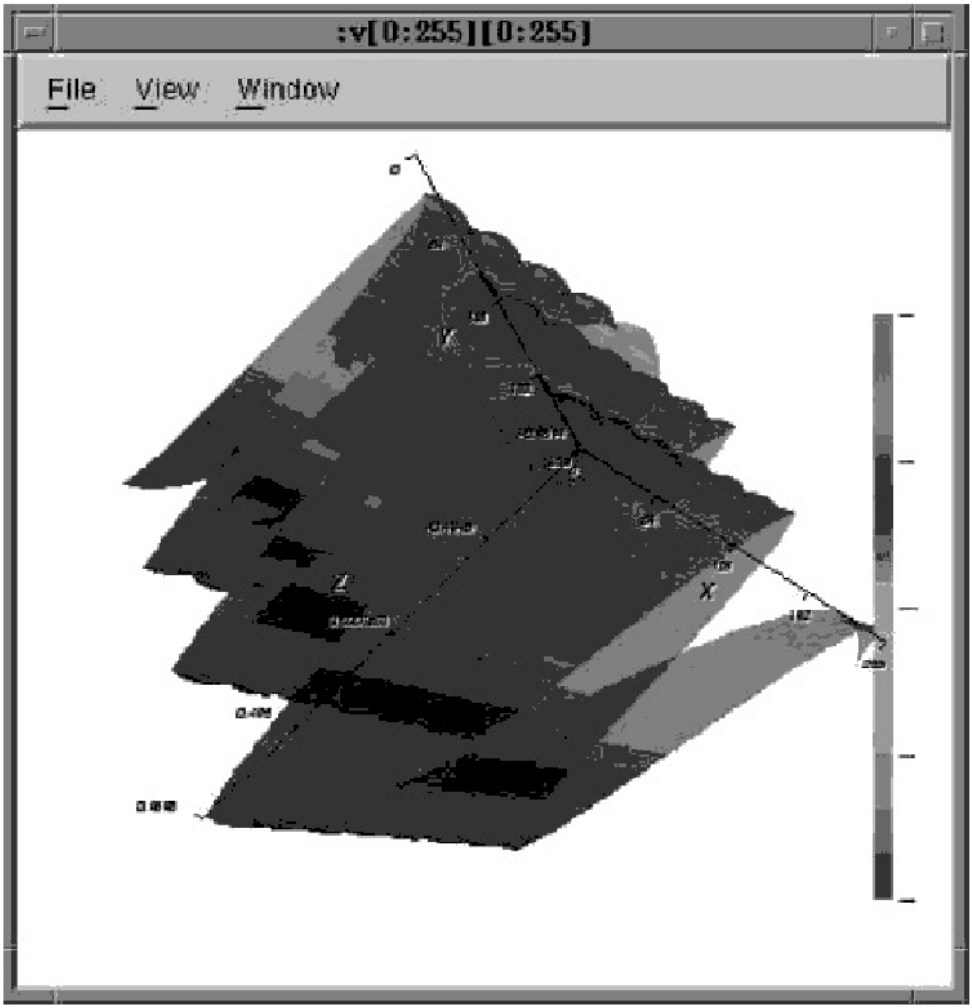



Рис. 5.12. Просмотр значений элементов массива в графическом виде

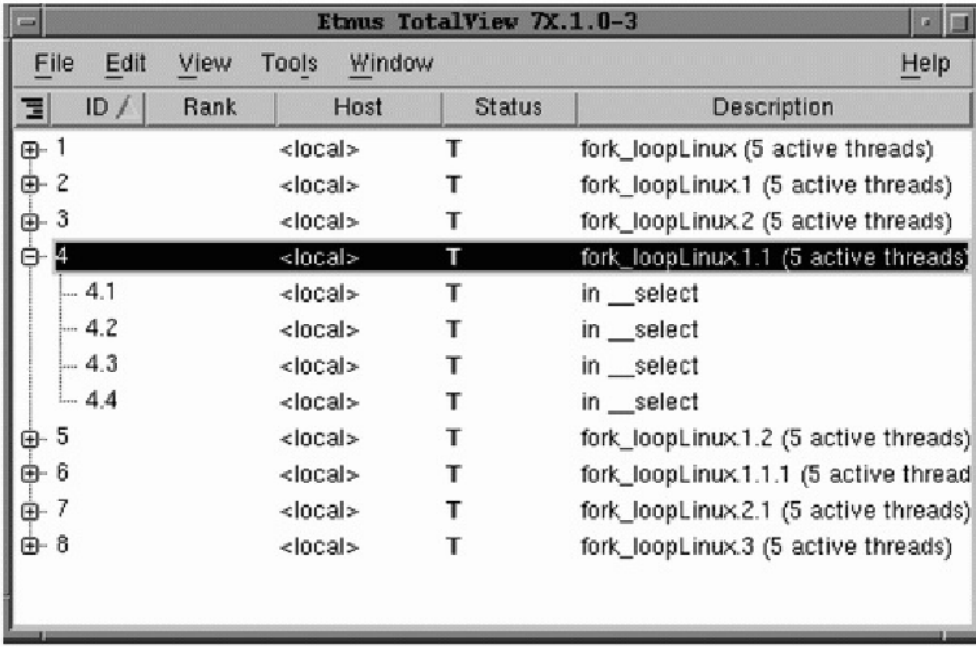



Рис. 5.13. Просмотр и выбор процессов в корневом окне

Отметим, что просматривать переменные в TotalView можно не только в числовом виде, как показано на рис.5.11, но и в графическом. Пример просмотра элементов массива в графическом виде изображен на рис.5.12.

При запуске многопоточной программы под управлением отладчика TotalView на экране появляется корневое окно (Root Window) с иерархической информацией по всем запущенным и запускаемым в процессе работы процессам. Вид этого окна представлен на рис.5.13. Все процессы можно разделить на две большие группы: локальные (local) и удаленные (remote). Локальные процессы - это процессы, запущенные на том же процессоре, на котором запущен и сам отладчик. Удаленные процессы запущены на других процессорах. Для отладчика, в принципе, безразлично, какие процессы просматриваются: удаленные или локальные. Программа в основном окне просмотра выводит информацию о них в единообразном виде. Достаточно просто выбрать процесс, как показано на рис.5.13.

В заключение отметим, что более подробно с возможностями отладчика

TotalView можно познакомиться, воспользовавшись его документацией.

# Настройка и ускорение программ в OpenMP

Настоящая лекция посвящена рассмотрению вопросов настройки и ускорения программ, разработанных с использованием OpenMP.

В процессе разработки программ этап настройки и ускорения работы программ занимает важное место. До этого этапа программа представляет собой еще сырой материал, поскольку, как правило, ее характеристики еще далеки от характеристик, запланированных перед началом разработки. Иногда ситуация оказывается еще хуже: характеристики программы перед этапом настройки вообще существенно ниже, чем те, которые ожидались. Поэтому на этапе настройки и ускорения программ, проанализировав с разных сторон созданную программу, нужно по возможности довести ее характеристики до запланированного уровня, а еще лучше - превзойти их. Важно отметить, что обычно доводка программы сопряжена с модификацией ее текста и очень часто - с изменением алгоритмов. Поэтому процесс настройки и ускорения тесно связан с предыдущими этапами разработки: компиляцией и тестированием. В самом деле, ведь в процессе настройки в текст разрабатываемой программы вносятся изменения, и поэтому программа должна быть вновь откомпилирована и протестирована.

Процесс настройки и ускорения программ, разрабатываемых с использованием OpenMP, тесно связан с анализом параллельных алгоритмов. Поэтому далее в настоящей лекции этому вопросу будет уделено значительное внимание. Также будет затронут вопрос автоматизации процесса настройки и ускорения.

## Основные принципы настройки и ускорения программ в OpenMP

В этом разделе остановимся на основных стратегиях настройки и ускорения программ с использованием OpenMP.

В первую очередь следует отметить, что при настройке программ в OpenMP по возможности следует применять средства автоматизированного распараллеливания программ. В настоящее время

все основные компиляторы Fortran и C/C++, предназначенные для разработки параллельных программ с использованием OpenMP, имеют возможности автоматического распараллеливания. Более подробно эти возможности будут рассмотрены в следующей лекции.

Чтобы найти и локализовать наиболее трудоемкие участки программы, можно воспользоваться возможностью профилирования (profiling) программы. В настоящее время этот процесс также в значительной степени автоматизирован. Существуют различные сервисные программы, позволяющие проводить профилирование разрабатываемых параллельных программ. Такие сервисные программы созданы различными производителями системного программного обеспечения, в том числе и компанией Intel. Как уже говорилось в предыдущей лекции, в состав набора программ Intel Threading Tools входит программа Intel Thread Profiler. Отметим также, что в составе программы Intel VTune Performance Analyzer имеются и другие средства профилирования программ. При профилировании программы важно выделить ее критический путь. Критический путь в многопоточной программе - это наиболее протяженный путь на диаграмме выполнения потоков. Для его определения необходимо провести анализ диаграммы выполнения потоков в параллельной программе. Пример такой диаграммы приведен на рис.6.1.

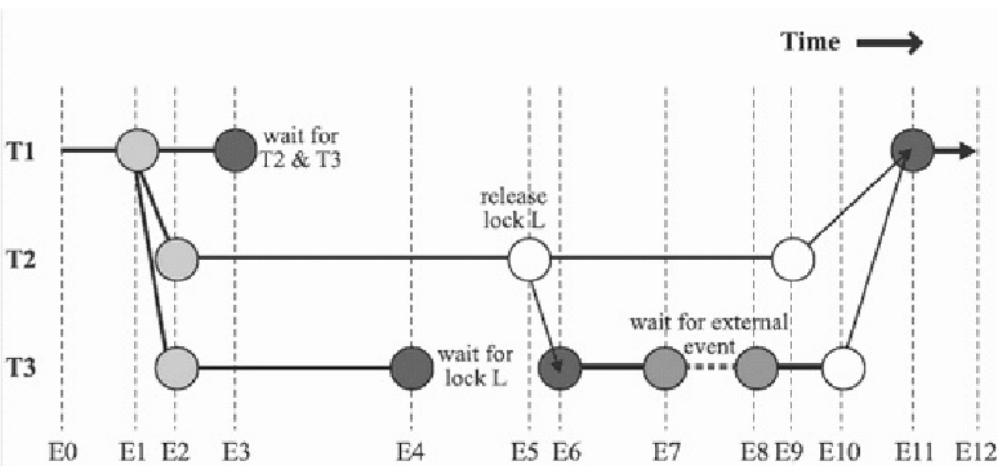



Рис. 6.1. Диаграмма потоков в многопоточной программе

На этом рисунке через `T1`, `T2` и `T3` обозначены потоки в программе, а

через `E1`, `E2`, ..., `E12` - события в программе. Длина отрезков на диаграмме соответствует времени выполнения потоков. Обратите внимание, что образование параллельных потоков требует определенных временных затрат, что и отражено на диаграмме.

После профилирования программы и анализа результатов в настраиваемую программу рекомендуется добавить инструкции OpenMP для распараллеливания наиболее затратных участков. В первую очередь это касается потоков, составляющих критический путь в программе.

В случае недостаточно эффективного распараллеливания программы с использованием OpenMP следует обратить самое пристальное внимание:

- на распараллеливание конструкций `do/for`. Надо обязательно учитывать высокую трудоемкость инициализации параллельных потоков;
- на неэффективность распараллеливания небольших циклов;
- на несбалансированность потоков;
- на недопустимость многочисленных ссылок к переменным в общей памяти;
- на ограниченный объем кэш-памяти;
- на высокую стоимость операции синхронизации;
- на значительные задержки доступа к удаленной общей памяти (на NUMA-компьютерах).

При распараллеливании вложенных циклов следует сначала распараллеливать внешние петли. Также следует иметь в виду, что петли циклов по объему вычислений могут быть зачастую "треугольными" и порождать несбалансированные параллельные потоки. Чтобы избежать несбалансированности при работе программы, следует правильно использовать возможности директивы OpenMP `schedule`.

## Иерархия памяти

Современные параллельные системы могут иметь весьма сложную *иерархию памяти*. Ниже дана примерная классификация памяти

современных высокопроизводительных систем по мере возрастания времени доступа к памяти:

- регистры;
- кэш-память 1-го уровня;
- кэш-память 2-го уровня;
- кэш-память 3-го уровня;
- локальная память;
- удаленная память (с доступом через интерфейс межузлового соединения Interconnect).

Время доступа к памяти существенно возрастает при движении по иерархии сверху вниз - порой даже в несколько раз. В связи с этим становится весьма актуальной задача эффективной загрузки кэш-памяти и регистров, а также минимизация доступа к удаленной памяти.

## Настройка кэш-памяти

Для эффективной загрузки кэш-памяти необходимо в первую очередь принимать специальные меры по выравниванию строк или столбцов массивов. В программах, написанных на алгоритмических языках C/C++, следует выравнивать строки массивов, а в программах, написанных на алгоритмическом языке Fortran, - столбцы.

Для эффективного применения кэш-памяти рекомендуется использовать многомерные массивы в одномерном виде.

Вложенные циклы следует модифицировать так, чтобы обеспечить последовательный быстрый доступ к элементам массива без скачков по индексам.

Рассмотрим пример фрагмента параллельной программы, представленный в примере 6.1. В этом примере элементы массива `prss`, используемые в петлях цикла по `k`, разнесены между собой со значительными скачками. Для преодоления указанной трудности и улучшения загрузки кэш-памяти можно применить перестановку циклов, как показано во фрагменте программы, приведенном в примере 6.2.

```
!$omp parallel do
!$omp& private (r1, r2, k, j)
  do j = jlow, jup
    do k = 2, kmax - 1
      r1 = prss(jminu(j),k) + prss(jplus(j),k) - 2. * prss(j,k)
      r2 = prss(jminu(j),k) + prss(jplus(j),k) - 2. * prss(j,k)
      coef (j,k) = ABC (r1/r2)
    enddo
  enddo
!$omp end parallel
```

**Пример 6.1. Пример фрагмента программы с циклом, петли которого содержат обращения к элементам массива со значительными скачками по индексам**

```
!$omp parallel do
!$omp& private (r1, r2, k, j)
  do k = 2, kmax - 1
    do j = jlow, jup
      r1 = prss(jminu(j),k) + prss(jplus(j),k) - 2. * prss(j,k)
      r2 = prss(jminu(j),k) + prss(jplus(j),k) - 2. * prss(j,k)
      coef (j,k) = ABC (r1/r2)
    enddo
  enddo
!$omp end parallel
```

**Пример 6.2. Пример модификации фрагмента программы с перестановкой циклов, оптимизирующей загрузку кэш-памяти**

Очень часто загрузку кэш-памяти можно улучшить с помощью транспонирования элементов массивов в петлях вложенных циклов. Рассмотрим такой пример. В примере 6.3 показаны исходный и модифицированный Исходный вариант:

Исходный вариант:

```
    real rx (jdim, idim)
!$omp parallel do
    do i = 2, n -1
      do j = 2, n
        rx(i, j) = rx(i, j -1) + …
```

```
      enddo
    enddo
```

Модифицированный вариант:

```
    real rx (idim, jdim)
!$omp parallel do
    do i = 2, n - 1
      do j = 2, n
        rx(j, I) = rx(j -1, i) + …
      enddo
    enddo
```

**Пример 6.3. Пример модификации программы с переопределением массива для оптимизации загрузки кэш-памяти**

фрагменты параллельной программы с вложенными циклами, в которых осуществляется обращение к элементам массивов `rx`. Для оптимизации загрузки кэш-памяти в этом примере массив `rx` транспонирован.

На компьютерах серии NUMALINK (non-*uniform memory access*) с неоднородным по времени доступом к памяти важно учитывать следующие особенности:

- надо хорошо представлять, где запущены потоки;
- надо четко понимать, какие данные загружены в локальную память;
- необходимо знать стоимость доступа к удаленной памяти. Для систем с NUMALINK все вышеперечисленные особенности очень сильно зависят от их архитектуры.

В программах, написанных с использованием OpenMP, существуют следующие возможности управления потоками:

- можно запускать поток на определенном процессоре;
- можно размещать данные в определенном месте памяти.

Эти задачи решаются с помощью системных инструкций, которые могут

быть различными на различных платформах и зависеть от используемого системного программного обеспечения.

## Зависимости по данным

При распараллеливании программ, написанных с использованием OpenMP, надо стремиться к максимально большей независимости параллельных потоков друг от друга. При этом следует избегать ситуаций, когда одни параллельные потоки используют данные из других. Если таких ситуаций нет, то говорят, что параллельные потоки независимы по данным.

Таким образом, при разработке параллельных программ с применением OpenMP важно, анализируя параллельные алгоритмы, находить в них зависимости по данным и стараться устранять их по мере возможности. Часто зависимости по данным между петлями циклов могут быть исключены с помощью модификации алгоритмов. Ниже представлен пример цикла с зависимостью по данным между петлями цикла.

```
DO I = 2 , 5
  A (I) = C * A (I -1)
ENDDO
```

Далее представлено элементарное распараллеливание этого цикла:

```
c$omp parallel sections
c$omp section
  A (2) = C * A (1)
c$omp section
  A (3) = C * C * A (1)
c$omp section
  A (4) = C * C * C *A (1)
c$omp section
  A (5) = C * C * C * C * A (1)
c$omp end sections nowait
```

Рассмотрим еще один пример программы с циклом, петли которого содержат зависимости по данным. Исходный фрагмент этой программы приведен в примере 6.4.

В примере 6.5 представлен пример распараллеленной версии программы. При распараллеливании вычисления индексов массивов были устранены операторы вычисления индексов по рекуррентным формулам.

Еще один характерный пример фрагмента программы с зависимостью по данным приведен в примере 6.6.

Распараллеливание этого фрагмента можно осуществить путем применения директивы OpenMP `reduction` к циклу `do`.

```
i1 = 0
  i2 = 0
do i =1, n
    i1 = i1 + 1
    B(i1) = …

    i2 = i2 + i
    A(i2) = …
enddo
```

**Пример 6.4. Пример фрагмента программы с циклом, петли которого содержат зависимости по данным**

```
c$omp parallel do
  do i =1, n
    B( i ) = …

    A((i**2 +i)/2) = …
  enddo
```

**Пример 6.5. Пример фрагмента программы с зависимостью по данным**

```
do i = 1, n
  xsum = xsum + a (i)
  xmu1 = xmu1 * a (i)
  xmax = max (xmax, a (i))
  xmin = min (xmin, a (i))
enddo
```

**Пример 6.6. Пример фрагмента программы с зависимостью по данным**

Из сказанного выше следует, что для эффективного распараллеливания циклов нужно по возможности обеспечить независимость петель вложенных циклов. При этом надо иметь в виду, что циклы с условными выходами, пример которых показан во фрагменте программы (пример 6.7), не распараллеливаются.

При распараллеливании вложенных циклов надо стремиться к вынесению независимых по данным петель циклов на верхний уровень и проводить распараллеливание только на этом уровне. Пример такого цикла приведен во фрагменте программы (пример 6.8). Распараллеливание такого цикла возможно только на самом верхнем уровне, т. е. по индексу `k`.

```
do i = 1, n
      a (i) = b (i) + c (i)
      if (a(i) .GT. amax) then
         a(i) = amax
         goto 100
      endif
   enddo
100 continue
```

**Пример 6.7. Пример нераспараллеливаемого цикла с условным переходом**

```
do k = 1, n
  do j = 1, n
    do i = 1, n
      a (i, j) = a (i, j) + b (i, k) * c (k, j)
    enddo
  enddo
enddo
```

**Пример 6.8. Пример цикла с независимостью по данным, вынесенной на верхний уровень**

## Эффективность параллельных программ и

# масштабируемость

После создания работающей параллельной программы обычно возникает проблема оценки ее эффективности. Под эффективностью параллельной программы понимают оценку ускорения, которое удается получить при переходе от последовательной версии программы к параллельной. Понятно, что такая оценка будет зависеть от числа параллельных процессоров.

Под масштабируемостью параллельной программы понимают изменение этой оценки в зависимости от числа параллельных процессоров.

Впервые теоретическая оценка эффективности параллельных программ предложена в работе [6.9]. В настоящее время эта оценка известна как закон Амдала, согласно которому максимальное ускорение `A` на параллельной вычислительной системе, содержащей `p` процессоров, удовлетворяет следующему неравенству

$$A \le \frac{1}{s + (1-s)p^{-1}}$$

Здесь `s` - суммарная доля последовательных блоков в параллельной программе. Понятно, что `0<s<1`.

Из закона Амдала следует, что при числе параллельных процессоров, стремящемся к бесконечности, максимальное ускорение `A` не превысит величины `1/s` вне зависимости от качества реализации параллельной части программы. При этом, если половина программы представляет из себя последовательную часть `s=1/2`, то ускорить такую программу более чем в два раза не удастся при любом числе параллельных процессоров.

Еще одна интересная оценка, следующая из закона Амдала: для того чтобы ускорить программу на 8 параллельных процессорах, например в четыре раза, необходимо обеспечить, чтобы последовательная часть программы не превышала 1/7 части программы.

Таким образом, из закона Амдала видно, что при наличии в

параллельных программах даже весьма незначительных последовательных частей происходит весьма существенное снижение быстродействия таких программ.

# Средства автоматизированного распараллеливания программ

В этой лекции будут рассмотрены средства автоматизированного распараллеливания программ.

Как уже отмечалось в предыдущих разделах, средства автоматизированного распараллеливания следует использовать на первом этапе разработки параллельной программы, если у нее уже существует работающий последовательный аналог. В этом случае на первом этапе разработки параллельной программы рекомендуется применить средства автоматизированного распараллеливания к исходной последовательной программе.

В настоящее время все основные компиляторы Fortran и C/C++, предназначенные для разработки параллельных программ с использованием OpenMP, имеют возможности автоматического распараллеливания. Кроме того, эти компиляторы допускают установку различных уровней автоматического распараллеливания, а также генерируют отчеты по результатам распараллеливания.

После автоматического распараллеливания рекомендуется провести профилирование программы, и если окажется, что наиболее трудоемкие места хорошо распараллелены, то работу по созданию параллельной версии программы можно закончить. В противном случае рекомендуется воспользоваться средствами OpenMP для дальнейшего распараллеливания наиболее трудоемких участков программы. Обычно так и происходит, поскольку для сложных программ с помощью автоматического распараллеливания редко удается получить максимальный эффект. Чтобы четко разобраться в этом вопросе, далее рассмотрим некоторые основные средства автоматизированного распараллеливания и принципы их работы.

# Основные средства автоматизированного распараллеливания и принципы их работы

Основным объектом автоматизированного распараллеливания с помощью специальных настроек компиляторов являются циклы. При

распараллеливании циклов компилятор, во-первых, выделяет независимые по данным петли циклов. Во-вторых, он оценивает приблизительный эффект от распараллеливания цикла. Если оценка этого эффекта соответствует заданному уровню, то компилятор определяет локальные переменные. И, наконец, только затем производится само распараллеливание цикла с помощью библиотек процессов. По результатам проведенной или не проведенной работы выдается сообщение в файл отчета о распараллеливании с объяснением причин.

Как правило, других возможностей автоматизированного распараллеливания большинство компиляторов не имеет. Все это в полной мере относится к компиляторам Fortran и C/C++ компании Intel и некоторым другим. Как видим, вышеперечисленные возможности автоматизированного распараллеливания сравнительно невелики.

Отметим, что значительно более широкими возможностями в области автоматического распараллеливания обладает компилятор Fortran 95 для операционной системы Linux компании Lahey, известный под названием Lahey/Fujitsu Fortran 95 for Linux. В этом компиляторе значительно улучшен процесс оптимизации программ, а также обеспечивается поддержка аппаратных функций новых процессоров. Кроме того, поддерживается автоматическое распараллеливание и дополнительные инструкции *SSE2* для процессоров Intel Xeon и Pentium IV. Компилятор Fortran Lahey/Fujitsu занимает одну из лидирующих позиций в областях, где необходимы большие объемы математических вычислений и осуществляется работа с большими массивами. Этот компилятор осуществляет автоматическое распараллеливание программ, в том числе и с использованием OpenMP версии 2.0. Оптимизация программ осуществляется для процессоров как компании Intel, так и компании AMD, при этом поддерживается оптимизация конвейера упреждающей выборки. Кроме того, для отладки параллельных программ компилятор совместим с параллельным отладчиком TotalView.

Кроме компиляторов существуют и более продвинутые программные системы в области автоматизированного распараллеливания.

В качестве примера отметим программный продукт Bert77, разрабатываемый компанией Paralogic. Программа Bert77 автоматически

распараллеливает Fortran-программы в средах *PVM* или MPI. Распараллеливание осуществляется с использованием механизма обмена сообщениями. Программа Bert77 - это коммерческий продукт, однако существует и его упрощенная бесплатная версия (lite-версия).

В подразделении компании Intel - KAI Software Lab (бывшая Kuck & Associates Inc.) разработаны и продолжают совершенствоваться:

- KAP/Pro Toolset - набор программных средств для распараллеливания больших расчетных программ на *параллельных вычислительных системах* с общей памятью;
- KAI C++ - компилятор C/C++ с широкими возможностями оптимизации и распараллеливания;
- Visual KAP - программа автоматического распараллеливания Fortran-программ в режиме визуального диалога;
- Visual KAP для OpenMP - инструмент визуальной генерации программ с использованием OpenMP.

Компания Applied Parallel Research Inc. предлагает программу *spf*, автоматически распараллеливающую программы для параллельных вычислительных систем с общей памятью с применением OpenMP или POSIX.

Компания Pacific-Sierra Research разработала интересные и очень эффективные средства:

- VAST/Parallel - программу автоматического распараллеливания программ на языках Fortran и C/C++ для параллельных вычислительных систем с общей памятью с использованием OpenMP;
- VAST/toOpenMP - программу, автоматически распознающую параллелизм и генерирующую директивы OpenMP в программах на языке Fortran.

Отметим также разработанную этой компанией среду проектирования, отладки и анализа производительности параллельных программ в системах с распределенной и общей памятью DEPP (Development Environment for *Parallel Programs*), реализующую директивы OpenMP.

Существует и еще целый ряд других средств автоматизированного распараллеливания, однако здесь были перечислены лишь те средства, которые осуществляют автоматическое распараллеливание с использованием OpenMP.

## Автоматическое распараллеливание программ с помощью компиляторов Intel

Рассмотрим возможности автоматического распараллеливания программ с применением современных компиляторов компании Intel. Отметим, что в компиляторах Intel реализована только одна принципиальная возможность распараллеливания: это создание многопоточных приложений распараллеливанием цикла с помощью библиотек процессов.

Далее рассмотрим подробнее настройки режима автоматического распараллеливания, имеющиеся в компиляторах Fortran и C/C++ компании Intel:

- `-parallel` - эта настройка позволяет компилятору автоматически создавать многопоточные версии программ с безопасным распараллеливанием циклов (подчеркнем, что кроме циклов в этом режиме больше ничего не распараллеливается);
- `-par_report{0|1|2|3}` - эта настройка позволяет создавать отчеты различного уровня - 0, 1, 2 или 3 (наиболее подробный отчет имеет уровень 3) по результатам автоматического распараллеливания. Рекомендуется тщательно анализировать такие отчеты, чтобы четко понять, какие еще места программы могли бы быть распараллелены и почему это не было сделано. Возможно, компилятору не хватило информации для принятия решения о распараллеливании такого участка программы. В этом случае это можно сделать вручную;
- `-par_threshold[n]` - эта настройка передает компилятору целое число `n` в диапазоне от `0` до `100`. `n` является оценкой эффективности распараллеливания циклов в процентах. Эту оценку компилятор производит самостоятельно.

# Настройки компиляторов Intel для распараллеливания программ с использованием OpenMP

В предыдущем параграфе были рассмотрены возможности автоматического распараллеливания программ для параллельных вычислительных систем с общей памятью. Однако, как правило, в автоматическом режиме нельзя выбрать все имеющиеся возможности для эффективного распараллеливания программ. Поэтому для параллельных вычислительных систем с общей памятью используют средства OpenMP для более глубокого распараллеливания программ.

Современные компиляторы Intel содержат следующие специальные настройки для создания параллельных программ с применением средств OpenMP для параллельных вычислительных систем с общей памятью:

- `-openmp` - эта настройка позволяет компилятору автоматически создавать многопоточные версии программ с использованием директив OpenMP;
- `-openmp_profile` - эта настройка добавляет в создаваемую программу средства профилирования программы для последующего анализа с помощью программы VTune Performance Analyzer;
- `-openmp_stubs` - эта настройка позволяет компилировать OpenMP программы в последовательном режиме. При этом предложения OpenMP игнорируются, а библиотека OpenMP используется редактором связей в последовательном режиме;
- `-openmp_report{0|1|2}` - эта настройка позволяет создавать отчеты различного уровня - 0, 1 или 2 (наиболее подробный отчет имеет уровень 2) по результатам автоматического распараллеливания с использованием OpenMP. Рекомендуется тщательно анализировать такие отчеты, чтобы четко понять, какие еще места программы могли бы быть распараллелены и почему это не было сделано. Возможно, компилятору не хватило информации для принятия решения о распараллеливании того или иного участка программы. В этом случае это можно сделать вручную.

Итак, вышеперечисленные настройки компиляторов позволяют создавать параллельные версии программ как просто с многопоточным распараллеливанием циклов, так и с помощью директив OpenMP. Возможно также сочетание обоих этих режимов распараллеливания.

В заключение для полноты изложения приведем простейшие команды компиляции программ `prog.c` и `prog.f` с помощью компиляторов Intel:

```
icc -openmp prog.c
```

и

```
ifort -openmp prog.f
```

Здесь `prog.c` и `prog.f` - соответственно тексты программ, написанных на алгоритмических языках C/C++ и Fortran с применением директив OpenMP. Для программ, написанных на Fortran, компилятор распознает также расширения файлов с текстами программ `for` и `f90` (для текстов программ, написанных на алгоритмическом языке Fortran 90). В результате компиляции в среде операционной системы Linux создается выполняемый файл `a.out`. Отметим также, что для создания выполняемых файлов с именем `prog` следует использовать следующие команды компиляции для программ, написанных на алгоритмических языках C/C++ и Fortran соответственно:

```
icc -openmp prog.c -o prog
```

и

```
ifort -openmp prog.f -o prog
```

## Настройки компиляторов Intel для распараллеливания программ для кластеров с распределенной памятью

В предыдущем параграфе были рассмотрены возможности автоматического распараллеливания программ, эффективные для параллельных вычислительных систем с общей памятью. При создании программ для параллельных вычислительных систем с распределенной

памятью, как было отмечено ранее, применяются методы распараллеливания с использованием MPI и *PVM*. Однако в новой версии компиляторов Intel, начиная с версии 9.1, реализовано расширение OpenMP, предназначенное для распараллеливания программ для вычислительных систем с распределенной памятью. Это расширение известно под названием Cluster OpenMP. В нем имеется возможность объявлять области данных доступными для всех узлов кластера. Эта возможность реализуется с помощью предложения `sharable`, которое будет рассмотрено далее. Пока же отметим, что это предложение реализует неявную передачу данных между узлами кластера по протоколу Lazy Release Consistency и избавляет программистов от утомительного процесса анализа посылаемых и принимаемых сообщений, необходимого при использовании MPI. В результате процесс параллельного программирования для параллельных вычислительных систем с распределенной памятью существенно упрощается.

Компиляторы Intel, начиная с версии 9.1.x.xx, получили следующие дополнительные настройки, предназначенные для создания параллельных программ для параллельных вычислительных систем с распределенной памятью (кластеров):

- `-cluster-openmp` - эта настройка позволяет компилятору создавать многопоточные версии программ с использованием директив расширенной версии OpenMP - Cluster OpenMP;
- `-cluster-openmp-profile` - эта настройка добавляет в создаваемую программу с применением Cluster OpenMP средства профилирования программы для последующего анализа с помощью программы VTune Performance Analyzer;
- `-[no-]clomp-sharable-propagation` - эта настройка позволяет создать отчет со списком переменных, которые должны быть объявлены программистом общедоступными для всех узлов кластера;
- `-[no-]clomp-sharable-info` - эта настройка позволяет создать отчет со списком переменных, которые были автоматически объявлены компилятором общедоступными для всех узлов кластера. Далее рассмотрим, как применить эти настройки для создания параллельных программ для вычислительных систем с распределенной памятью.

Во-первых, отметим, что для компиляции программ `prog.c` и `prog.f`, написанных соответственно на языках C/C++ и Fortran с использованием Cluster OpenMP, можно воспользоваться соответственно следующими командами:

```
icc -cluster-openmp prog.c -o prog
```

и

```
ifort -cluster-openmp prog.f -o prog
```

В результате будут созданы выполняемые файлы с именами `prog`. Однако чтобы запустить программу `prog` на выполнение с помощью команды

```
./prog
```

необходимо, чтобы в директории, откуда запускается программа, содержался файл `kmp_cluster.ini`. Если же в этой директории этот файл отсутствует, то операционная система ищет файл . `kmp_cluster` в директории с именем, определенным значением переменной окружения `KMP_CLUSTER_PATH`. Если же эта переменная окружения отсутствует или не определена, то операционная система ищет . `kmp_cluster` файл в домашнем каталоге пользователя. В файле `kmp_cluster.ini` и его аналогах перечисляются узлы кластера, на которых выполняется задание, и количество параллельных потоков на узлах. В следующем примере 7.1 представлена распечатка простейшего `kmp_cluster.ini` -файла.

```
[tt1@hpc1 clomp_samples] $ cat kmp_cluster.ini
--hostlist = hpc1,hpc2,hpc3,hpc4 --process_threads=2
[tt1@hpc1 clomp_samples]$
```

**Пример 7.1. Распечатка простейшего kmp_cluster.ini-файла**

В этом файле после ключевого слова `hostlist` следует список узлов кластера, а после ключевого слова `process_threads` - число физических параллельных потоков в узле (оно равно числу процессоров в узле кластера).

Теперь рассмотрим вопрос использования третьей из вышеперечисленных четырех настроек для перенастройки OpenMP-программ под Cluster OpenMP.

Перед тем как начать перенастройку, следует убедиться, что программа корректно работает под OpenMP. Затем следует оттранслировать и собрать программу с настройкой -cluster-openmp. Если после этого программа будет работать корректно, то можно считать, что перенастройка успешно завершена. В противном случае следует продолжить процесс перенастройки следующим образом.

Во-первых, надо определить переменные программы, которые необходимо сделать общими для всех узлов кластера. Эти общие переменные имеют тип `sharable`. Для их описания в Cluster OpenMP имеются специальные предложения следующего вида:

```
#pragma intel omp sharable(var1, var2, ..., varN)
```

в программах, написанных на языке C/C++, и

```
!dir$ omp sharable(var1, var2, ..., varN)
```

в программах, написанных на языке Fortran. Здесь `var1, var2, ..., varN` - список переменных, которые объявлены общими для всех узлов кластера. В программах на языке Fortran в качестве переменных могут быть использованы имена common-блоков. При этом при описании с помощью предложения Cluster OpenMP `sharable` имена common-блоков должны быть заключены в слэши, как показано ниже:

```
!dir$ omp sharable( /cblock1/, /cblock2/, ..., /cblockN/ )
```

Важно отметить, что переменные, которые применяются в описаниях `EQUIVALENCE` в программах, написанных на языке Fortran, не могут быть объявлены как `sharable`.

Для определения `sharable` -переменных в программе, написанной с OpenMP, можно воспользоваться следующей настройкой компиляторов Intel:

```
-clomp-sharable-propagation
```

Эту настройку следует использовать обязательно вместе с настройкой -*ipo*. В результате компиляции с такой настройкой OMP-программ могут появиться сообщения о необходимости определения `sharable` -переменных. Пример такого сообщения показан в примере 7.2.

```
fortcom: Warning: Sharable directive should be inserted by user
as '!dir$ omp sharable(s)' in file p2.f, line 2, column 18
```

**Пример 7.2. Пример сообщения о необходимости определения sharable-переменных**

Из этого примера видно, что сообщение компилятора указывает конкретное место в программе, где должны быть определены общие для всех узлов кластера переменные, а также Cluster OpenMP-директиву, которую следует использовать для определения имен переменных В программах на языке C++ наряду с предложениями `sharable` необходимо добавить включение ссылки на файл, в котором описаны переменные `sharable`. Эта ссылка имеет следующий вид:

```
#include <kmp_sharable.h>
```

На втором этапе перенастройки программы под Cluster OpenMP следует в файлах программы расставить необходимые директивы для определения общих для всех узлов кластера переменных, после чего необходимо перекомпилировать и пересобрать программу и убедиться в ее корректной работе.

Отметим, что предложение `sharable` - единственное новое дополнительное предложение, отличающее Cluster OpenMP от стандартного OpenMP. Однако в Cluster OpenMP имеется значительное число дополнительных специфических настроек, таких, как `kmp_cluster.ini` -файл и т. п., которые и рассмотрим далее. Начнем с более подробного изучения настроек в `kmp_cluster.ini` -файле.

## Дополнительные настройки в kmp_cluster.ini-файле

В этом разделе рассмотрим более подробно настройки, которые можно задавать в `kmp_cluster.ini` -файлах. Настройки `--hostlist` и `--process_threads` уже были рассмотрены ранее. Отметим лишь, что по умолчанию значение `process_threads` равно `1`, а список `hostlist` по умолчанию тождественен узлу кластера, на котором в данный момент осуществляется работа.

Настройка `--omp_num_threads` в файле `kmp_cluster.ini` позволяет определить число потоков OpenMP, т. е. значение переменной окружения `OMP_NUM_THREADS`. По умолчанию значение `omp_num_threads` равно произведению значений настроек

```
processes * process_threads
```

Настройка `--processes` позволяет задавать число параллельных процессов. По умолчанию оно принимает значение, либо равное числу узлов в списке `hostlist`, либо, если определено значение настройки `omp_num_threads`, - равное частному

```
omp_num_threads / process_threads
```

Отметим, что если значение `omp_num_threads` не равно значению произведения `processes * process_threads`, то следует переопределить настройку `process_threads` как `omp_num_threads / processes`.

Настройка `--transport` определяет протокол связи между узлами кластера. Эта настройка по умолчанию принимает значение `--transport=tcp`, что соответствует протоколу TCP IP. Кроме того, допустимо значение `--transport=dapl`. В этом случае настройка `--adapter` должна быть определена следующим образом:

```
adapter=Openib-ib0
```

В последнем случае в команду компиляции программы должны быть добавлены настройки примерно следующего вида: `-L/usr/local/ibgd/lib` и `-ldat`, первая из которых определяет путь к библиотеке, а вторая - саму библиотеку `dat`.

Настройка `--launch` определяет метод запуска Cluster OpenMP-программы на удаленных узлах кластера. Она может принимать два возможных значения - `rsh` или `ssh` ; первое значение является тем значением, которое эта настройка принимает по умолчанию.

Настройка `--sharable_heap` определяет число страниц, выделяемых под общую для всех узлов кластера память. По умолчанию ее значение равно `16384`.

Настройка `--`*`suffix`* позволяет определить суффикс, который будет добавлен слева к именам узлов кластера. Эта настройка полезна в случае наличия различных типов межузловых соединений в кластере. По умолчанию *`suffix`* имеет значение `null`.

Для задания задержки при запуске удаленных процессов можно воспользоваться настройкой `--startup_timeout`. Значение этой настройки задается в секундах, и по умолчанию оно равно `30`. Если процесс не успевает стартовать за заданное время, то выполнение программы прерывается.

Настройка `--IO={system | debug | files}` позволяет переопределять выходные потоки `stderr` и `stdout` следующим образом. Настройка по умолчанию `system` определяет пути к файлам `stderr` и `stdout` в соответствии с правилами оболочки `shell`. В случае `debug` происходит переадресация потоков `stderr` и `stdout` с удаленных узлов на рабочий узел кластера, при этом для идентификации файлов, соответствующих *удаленным процессам*, в их имена добавляется префикс `Process x`:, где `x` - номер *удаленного процесса*. В случае `files` осуществляется переадресация файлов `stderr` и `stdout` в файлы `clomp-<process_id>-stdout` и `clomp-<process_id>-stderr` соответственно, где `process_id` есть идентификатор процесса.

Настройка `--[no]`*`heartbeat`* позволяет включить или выключить механизм контроля того, что все запущенные процессы функционируют нормально. По умолчанию устанавливается настройка `--`*`heartbeat`*.

Для задания имени директории, в которой размещается файл обмена страниц динамической, общей для всех процессоров памяти, в

`kmp_cluster.ini` -файле имеется специальная настройка `--back_store= <имя директории обмена страниц памяти>`. По умолчанию обмен страниц памяти осуществляется в директории `/tmp`.

Наконец, последняя настройка `--[no-]divert_twins` предназначена для резервирования двух страниц в директории обмена страниц для файла обмена страниц памяти. По умолчанию эта настройка имеет значение `--no-divert_twins`, соответствующее отключению режима резервирования.

В заключение отметим, что в `kmp_cluster.ini` -файле строго запрещается использовать переменные `PATH`, `SHELL` и `LD_LIBRARY_PATH`.

## Дополнительные настройки компилятора Fortran в Cluster OpenMP

В программах, написанных на языке Fortran, имеются дополнительные настройки компилятора для работы и анализа переменных, размещенных в общей для всех узлов кластера памяти. Отметим, что компиляторы Fortran компании Intel, начиная с версии 9.1.xx, по умолчанию определяют как `sharable` переменные программы, содержащиеся в common-блоках, а также переменные программных модулей, локальные переменные типа `SAVE` и временные переменные для вычисления выражений в вызовах функций и подпрограмм. Для изменения этого режима определения по умолчанию в Intel Fortran компиляторе имеются четыре дополнительные настройки.

Настройка `-[no]-clomp-sharable-common` позволяет отключить или включить режим определения по умолчанию переменных common-блоков как `sharable`.

Для отключения или включения режима определения по умолчанию локальных переменных типа `SAVE` как `sharable` следует использовать настройку `-[no]-clomp-sharable-localsaves`.

Настройка `-[no]-clomp-sharable-modvars` позволяет отключить или включить режим определения по умолчанию переменных программных модулей как `sharable`.

Для отключения или включения режима определения по умолчанию временных переменных для вычисления выражений в вызовах функций и подпрограмм как `sharable` следует использовать настройку `-[no]-clompsharable-argexprs`.

В заключение этого раздела отметим, что по умолчанию все четыре вышеперечисленные настройки устанавливаются с префиксом `no`, т. е. в выключенном режиме.

## Переменные окружения Cluster OpenMP

При программировании с использованием Cluster OpenMP можно определить ряд переменных окружения, позволяющих контролировать и изменять процессы выполнения программ.

Для задания размера стеков потоков в Cluster OpenMP можно воспользоваться переменной окружения `KMP_STACKSIZE`. Размер стека задается в килобайтах (K) или мегабайтах (M). По умолчанию задается `KMP_STACKSIZE=1M`. Определять размер стека можно не только для отдельных параллельных потоков, но и для общих переменных типа `sharable`. В последнем случае следует определить переменную окружения `KMP_SHARABLE_STACKSIZE`. По умолчанию значение этой переменной также равно `1M`.

Переменная окружения `KMP_STATSFILE` предназначена для определения имени файла, в который будет выводиться статистическая информация при профилировании программы, созданной с настройкой `-cluster-openmp-profile`. По умолчанию `KMP_STATSFILE=guide.gvs`.

При отладке Cluster OpenMP программ необходимо задать имя отладчика. Для этого имеется специальная переменная окружения `KMP_CLUSTER_DEBUGGER`. Эта переменная может принимать значения `idb` (отладчик Intel), `gdb` (отладчик GNU) или `totalview`

(отладчик TotalView). Значение по умолчанию для этой переменной окружения отсутствует.

Переменная окружения `KMP_WARNINGS` позволяет включить ( `1` или `on` ) или выключить ( `0` или `off` ) режим вывода предупреждающих сообщений библиотеки выполнения ( `runtime library` ) Cluster OpenMP. По умолчанию задается `KMP_WARNINGS=on`.

Для задания *режима вывода* предупреждающих сообщений об общих для всех узлов кластера переменных в Cluster OpenMP имеется переменная окружения `KMP_SHARABLE_WARNINGS`. Она также может принимать значения "включить" ( `1` или `on` ) или "выключить" ( `0` или `off` ) режим вывода предупреждающих сообщений. Но по умолчанию задается режим `KMP_SHARABLE_WARNINGS=off`.

Переменная окружения `KMP_CLUSTER_SETTINGS` позволяет организовать вывод на экран всех переменных окружения, в том числе определенные в `kmp_cluster.ini` -файле, а также настройки, заданные в этом же файле. Для этой переменной отсутствует значение по умолчанию.

Переменная окружения `KMP_CLUSTER_PATH` определяет путь к `kmp_cluster.ini` -файлу, если этот файл отсутствует в разделе, из которого запускается программа. По умолчанию это путь к домашнему разделу каталога пользователя.

Для вывода информации об использовании настроек в `kmp_cluster.ini` -файле следует задать переменную окружения `KMP_CLUSTER_HELP`. Еще одна сервисная переменная окружения `KMP_VERSION` позволяет выводить информацию о версии в процессе работы библиотеки выполнения (run-*time library*).

Наконец, последняя переменная окружения `KMP_DISJOINT_HEAP` задает размер несимметрично распределенной по процессам динамической памяти в килобайтах (K) или мегабайтах (M). Минимальный размер несимметрично распределенной по процессам динамической памяти равен `2K`. Значение по умолчанию этой переменной не определено.

# Особенности реализации переменных окружения OpenMP в Cluster OpenMP

В этом разделе рассматриваются особенности реализации переменных окружения OpenMP в среде Cluster OpenMP.

В Cluster OpenMP максимальное число параллельных потоков в параллельной области программы определяется заданием настройки `--omp_num_threads`. Однако число параллельных потоков может быть и меньше максимального, если оно определено с помощью задания переменной окружения `OMP_NUM_THREADS` или с помощью вызова функции `omp_set_num_threads()` из библиотеки выполнения (runtime library).

Число процессоров, которое возвращает функция `omp_get_num_procs()`,

- это сумма процессоров, вычисленная по всем узлам вычислительной системы. Cluster OpenMP не поддерживает вложенный параллелизм. Поэтому вложенные параллельные области выполняются как последовательность потоков, причем на том же процессоре, на котором выполняется их родительский поток. Таким образом, вызов функции `omp_set_nested` или задание переменной окружения `OMP_NESTED` не имеют никакого влияния.

При задании переменных окружения, определяющих загрузку вычислительной системы, следует иметь в виду, что если переменная окружения `OMP_SCHEDULE` не определена, то это означает, что по умолчанию задается статический ( `static` ) режим загрузки параллельных процессов.

# Специальные функции Cluster OpenMP

В Cluster OpenMP имеется целый ряд специальных функций для задания некоторых переменных окружения в процессе выполнения программы, а также для выделения и освобождения памяти в процессе работы программы и передачи сигналов. Рассмотрение этих функций начнем с

рассмотрения функций задания переменных окружения и функций, информирующих об окружающей среде.

Функции

```
void kmp_set_warnings_on ( void )
```

и

```
void kmp_set_warnings_off ( void )
```

предназначены для задания *режима вывода* предупреждающих сообщений в процессе работы библиотеки выполнения. Первая функция устанавливает переменную окружения `KMP_WARNINGS=on`, а вторая - `KMP_WARNINGS=off` отменяет ее.

Следующая функция

```
omp_int_t kmp_get_process_num ( void )
```

возвращает номер текущего выполняемого процесса. Функция

```
omp_int_t kmp_num_processes ( void )
```

возвращает общее число потоков в текущий момент времени. Следующая функция

```
omp_int_t kmp_get_process_thread_num ( void )
```

возвращает номер текущего потока в выполняемом процессе.

Теперь перейдем к рассмотрению функций передачи сигналов в Cluster OpenMP. Всего таких функций шесть.

Первая функция

```
void kmp_lock_cond_wait ( omp_lock_t *lock )
```

предназначена для *приостановки выполнения процесса*, содержащего данный оператор, до момента разблокировки объекта, блокировка которого установлена переменной `lock`.

Вторая функция

```
void kmp_lock_cond_signal ( omp_lock_t *lock )
```

посылает сигнал о блокировке объекта, установленной с помощью переменной `lock`.

Третья функция

```
void kmp_lock_cond_broadcast ( omp_lock_t *lock )
```

рассылает всем процессам сигнал о блокировке объекта, установленной с помощью переменной `lock`.

Четвертая функция

```
void kmp_nest_lock_cond_wait ( omp_nest_lock_t *lock )
```

предназначена для *приостановки выполнения процесса*, содержащего данный оператор, до момента разблокировки объекта, вложенная блокировка которого установлена переменной `lock`.

Пятая функция

```
void kmp_nest_lock_cond_signal ( omp_nest_lock_t *lock )
```

посылает сигнал о вложенной блокировке объекта, установленной с помощью переменной `lock`.

Наконец, шестая функция

```
void kmp_nest_lock_cond_broadcast ( omp_nest_lock_t *lock )
```

рассылает всем процессам сигнал о вложенной блокировке объекта, установленной с помощью переменной `lock`.

Для работы с памятью в Cluster OpenMP имеется девять специальных функций. Далее рассмотрим эти функции.

Функция

```
void *kmp_sharable_malloc ( size_t size )
```

предназначена для выделения динамической, общей для всех узлов кластера памяти. Объем выделяемой памяти определяется значением переменной `size`.

Следующая функция

```
void *kmp_aligned_sharable_malloc ( size_t size )
```

предназначена для выделения динамической, общей для всех узлов кластера памяти. Это выделение осуществляется с выравниванием по границе страницы памяти. Объем выделяемой памяти определяется значением переменной `size`.

Функция

```
void *kmp_sharable_calloc ( size_t n, size_t size )
```

предназначена для выделения в динамической, общей для всех узлов кластера памяти места под массив из `n` элементов размера `size`. При этом все элементы этого массива заполняются нулями.

Еще одна функция

```
void *kmp_sharable_recalloc ( void *ptr, size_t size )
```

предназначена для переинициализации выделения места в динамической, общей для всех узлов кластера памяти под массив размера `size`. Адрес этого места определяется указателем `*ptr`. При этом все элементы этого массива заполняются нулями.

Функция

```
void *kmp_sharable_free ( void *ptr )
```

предназначена для освобождения в динамической, общей для всех узлов кластера памяти раздела, определенного указателем `*ptr`.

Следующие функции

```
void *kmp_private_mmap (char *file,size_t *len,void **addr)
```

и

```
void *kmp_private_munmap ( void *start )
```

предназначены для отображения файлов в области индивидуальной адресной памяти процессов, начиная с одного и того же адреса. Первая функция `kmp_private_mmap` отображает `file` длиной `len` байт в память, начиная с адреса `addr`. Вторая функция `kmp_private_munmap` удаляет все отображения из заданной области памяти с начальным адресом `start`, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти". Отображение удаляется автоматически при завершении процесса. Однако закрытие файла не приводит к снятию отображения. Эти функции возвращают `0` в случае успешного завершения и `-1` в противном случае. Отметим, что их можно применять только в последовательной области программы. В параллельной области они не поддерживаются. Кроме того, эти две функции имеют тип `read-only`, т.е. после удаления отображения с помощью второй функции содержимое файла не обновляется содержимым области памяти.

Функции

```
void *kmp_sharable_mmap ( char *file, size_t *len, void **addr )
```

и

```
void *kmp_sharable_munmap ( void *start )
```

предназначены для отображения файлов в области адресной памяти главного процесса, начиная с одного и того же адреса. Первая функция `kmp_sharable_mmap` отображает `file` длиной `len` байт в память, начиная с адреса `addr`. Вторая функция `kmp_sharable_munmap` удаляет все отображения из заданной области памяти с начальным адресом `start`, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти". Отображение удаляется автоматически при завершении процесса. Функции возвращают `0` в случае успешного завершения и `-1` в противном

случае. Эти две функции имеют тип `read-write`, т. е. после удаления отображения с помощью второй функции содержимое файла обновляется содержимым области памяти. Отметим также, что их можно применять только в последовательной области программы. В параллельной области их использовать нельзя.

## Загрузка данных в общую для всех узлов кластера память в Cluster OpenMP

При рассмотрении проблемы загрузки данных в общую ( `sharable` ) для всех узлов кластера память прежде всего следует иметь в виду, что наименьший размер элементов, с которыми оперирует Cluster OpenMP, равен 4 байтам. Если же размер элементов по каким-то причинам оказывается меньше 4 байт, то операции с ними не производятся. Таким образом, если в программе, например, определены массивы строковых переменных

```
char a[100], b[100]
```

а затем эти переменные определены как `sharable`, то следующий параллельный блок в программе выполняться не будет:

```
#pragma omp parallel for
for( i=0; i<N; i++ )
 {
   a[i] = b[i];
 }
```

Для динамической *загрузки переменных* в общую ( `sharable` ) для всех узлов кластера память в программах, написанных на языке C/C++, можно воспользоваться одной из двух следующих функций:

```
void * kmp_sharable_malloc ( int size )
```

или

```
void * kmp_aligned_sharable_malloc ( int size )
```

Обе эти функции определяют в динамическом режиме в общей для всех узлов кластера памяти области заданного размера и возвращают адреса этих областей. При этом следует иметь в виду, что вторая из этих функций осуществляет выравнивание выделяемой области памяти по границе страницы и тем самым существенно ускоряет процесс обращения к памяти. Для освобождения выделенной памяти следует использовать функцию

```
void kmp_sharable_free ( *ptr )
```

Здесь `ptr` - указатель на освобождаемую область памяти.

*Динамическое выделение памяти* в общей ( `sharable` ) для всех узлов кластера памяти в программах, написанных на языке Fortran, осуществляется следующим образом:

```
integer, allocatable :: x( : )
!dir$ omp sharable (x)
   allocate( x( 500 ) )
```

В этом примере в общей для всех узлов кластера памяти динамически размещается целочисленный массив, состоящий из 500 элементов.

В программах, написанных на языке C++, при динамическом размещении переменных и объектов в общей для всех узлов кластера памяти необходимо добавить включение ссылки на файл, в котором описаны переменные `sharable`. Эта ссылка имеет следующий вид:

```
#include <kmp_sharable.h>
```

Теперь рассмотрим вопрос создания динамических объектов в общей для всех узлов кластера памяти. Пусть в исходном тексте программы динамически определяется объект класса `zoo`, например, следующим образом:

```
zoo * fp = new zoo (10);
```

В Cluster OpenMP это определение модифицируется так:

```
zoo * fp = new kmp_sharable zoo (10);
```

Теперь рассмотрим вопрос создания класса объектов, динамически размещаемых в общей для всех узлов кластера памяти. Пусть в исходной программе этот класс создавался, например, следующим образом:

```
class zoo : public zoo_base
 {
  // …содержание класса zoo
 };
```

В программе, написанной с использованием Cluster OpenMP, создание этого класса осуществляется следующим образом:

```
class zoo : public zoo_base, public kmp_sharable_base
 {
  // …содержание класса zoo
 };
```

При динамическом создании *STL* (Standard Template Library) контейнеров в общей для всех узлов кластера памяти следует иметь в виду, что, во-первых, необходимо описать динамическое размещение контейнера как объекта и, кроме того, необходимо описать динамическое размещение его содержания. Если в исходной программе контейнер создавался, например, следующим образом:

```
std::vec<int> * vecp = new std :: vec<int>;
```

то в программе, написанной с использованием ClusterOpenMP, создание такого контейнера должно быть описано, например, так:

```
std::vec<int,kmp_sharable_allocator<int>>*vecp=new
kmp_sharable std :: vec<int, kmp_sharable_allocator<int>>;
```

## Настройки отладчиков в Cluster OpenMP

Для отладки многопоточных параллельных программ в Cluster OpenMP можно использовать как бесплатный отладчик GNU `gdb`, так и лицензионные отладчики `idb` компании Intel и TotalView компании Etnus. Однако для того, чтобы корректно пользоваться этими отладчиками, необходимо сделать специальные настройки в системных

файлах. Далее рассмотрим подробно эти настройки применительно ко всем вышеперечисленным отладчикам. Предварительно отметим, что для корректной работы всех этих отладчиков необходимо отключить прерывание по сигналу ошибки адресации SIGSEGV (Signal Segmentation *Violation*). Причина в том, что этот сигнал используется для внутренних нужд Cluster OpenMP, поэтому он не является аварийным и не несет информации об ошибках доступа к памяти (в Cluster OpenMP имеется свой собственный обработчик этого сигнала). Отключение этого сигнала производится по-разному для рассматриваемых отладчиков и будет рассмотрено далее. Для всех отладчиков в `kmp_cluster.ini`-файле необходимо использовать настройку `--no-heartbeat`, отключающую режим аварийного завершения параллельных потоков при отсутствии в течение определенного промежутка времени ответных сигналов от параллельных потоков. Такими сигналами в Cluster OpenMP постоянно обмениваются все параллельные потоки. Если в течение определенного промежутка времени ответный сигнал от какого-либо потока отсутствует, то это считается свидетельством его аварийного завершения. В этом случае происходит аварийное завершение всех потоков рассматриваемой задачи.

В бесплатном отладчике GNU `gdb` для отключения сигнала аварийного доступа к памяти, во-первых, необходимо в системном файле `.gdbinit`, находящемся в домашнем разделе пользователя, добавить строку

```
handle SIGSEGV nostop noprint
```

Во-вторых, необходимо определить отладчик с помощью переменной окружения `KMP_CLUSTER_DEBUGGER`, например следующим образом:

```
export KMP_CLUSTER_DEBUGGER=gdb
```

В-третьих, необходимо определить переменную окружения `DISPLAY`, указывающую на IP-адрес компьютера, с которого стартует X Windows. Эту переменную можно определить либо в `kmp_cluster.ini` -файле, например так:

```
DISPLAY= 10.0.6.191:0
```

либо с помощью команды

```
export DISPLAY= 10.0.6.191:0
```

которую следует поместить в системный файл, откуда считываются переменные окружения при старте программы `xterm`. После запуска отладчика `gdb` следует набрать команду

```
break __itmk_segv_break
```

для перехвата ошибок адресации памяти.

В отладчике `idb` компании Intel для отключения сигнала аварийного доступа к памяти необходимо в системном файле `.dbxinit`, находящемся в домашнем разделе пользователя, добавить строку

```
ignore segv
```

Во-вторых, необходимо определить отладчик с помощью переменной окружения `KMP_CLUSTER_DEBUGGER` следующим образом:

```
export KMP_CLUSTER_DEBUGGER=idb
```

Переменная окружения `DISPLAY` для этого отладчика задается так же, как и для отладчика `gdb`. После запуска отладчика `idb` следует набрать команду

```
stop in __itmk_segv_break
```

для перехвата ошибок адресации памяти.

В отладчике TotalView компании Etnus для отключения сигнала аварийного доступа к памяти необходимо в системном файле `.tvdrc`, находящемся в домашнем разделе пользователя, добавить строку

```
dset TV::signal_handling_mode { Resend=SIGSEGV }
```

После запуска отладчика TotalView следует набрать команду

```
breakpoint in __itmk_segv_break
```

для перехвата ошибок адресации памяти.

## Настройка динамической памяти при отладке в Cluster OpenMP

При использовании динамической памяти в программах с использованием Cluster OpenMP возможна следующая ошибочная ситуация: некоторые данные одного типа в параллельных потоках размещены в локальной памяти с одними и теми же адресами (т. е. некоторые данные из параллельных потоков симметрично размещены в локальной памяти). При этом указатель на эти данные в одном из параллельных потоков ошибочно передается в другой поток, и из этого последнего потока осуществляется доступ к данным по переданному адресу. В таком случае ошибки доступа к данным может и не возникнуть, т. е. сигнал SIGSEGV не будет инициирован. Однако данные, которые будут извлечены из другого потока, будут неверными для исходного потока. Для

того чтобы выявить такие ошибочные ситуации, в Cluster OpenMP реализован

механизм несимметричного выделения динамической памяти (Disjoint Heap).

Для включения механизма несимметричного выделения динамической памяти в параллельных потоках следует задать переменную окружения `KMP_DISJOINT_HEAPSIZE`, например, следующим образом:

```
export KMP_DISJOINT_HEAPSIZE 128M
```

Здесь была задана область для несимметричного выделения динамической памяти, равная `128` мегабайтам. При выделении размера этой области следует позаботиться о том, чтобы ее размер был не меньше, чем произведение размера области динамической памяти, выделяемой для одного параллельного потока, на число параллельных потоков. В этом случае области динамической памяти в параллельных потоках будут размещены в адресном пространстве без пересечений и наложений. Тогда обращение в каком-либо из параллельных потоков к

области динамической памяти по указателю, переданному из другого потока, приведет к ошибке доступа к данным и возникновению сигнала SIGSEGV. Это и будет побудительной причиной более тщательного анализа работы с данными в параллельных потоках для устранения таких некорректных ситуаций.

В заключение отметим, что по умолчанию значение переменной окружения `KMP_DISJOINT_HEAPSIZE` задается равным `2` мегабайтам. При задании меньшей величины размер `KMP_DISJOINT_HEAPSIZE` автоматически увеличивается до `2` мегабайт.

Подчеркнем еще раз, что задание переменной окружения `KMP_DISJOINT_HEAPSIZE` необходимо только на этапе отладки программы для поиска ошибок некорректного обращения к динамической памяти в параллельных потоках. После выявления всех таких ошибок и создания окончательной оптимизированной версии программы эту переменную окружения следует удалить.

## Сводка переменных окружения Cluster OpenMP

В предыдущих разделах был уже рассмотрен ряд специфических переменных окружения Cluster OpenMP. Ниже перечислены все переменные окружения, допустимые в Cluster OpenMP, и описаны способы их применения.

- Переменная окружения `KMP_STACKSIZE` задает объем стека в параллельных потоках Custer OpenMP. Величина этой переменной задается в килобайтах (K) или мегабайтах (M). По умолчанию значение этой переменной равно `1M`.
- Переменная окружения `KMP_SHARABLE_STACKSIZE` задает объем стека для размещения `sharable` (общих для всех узлов кластера) данных для OpenMP потоков. Величина этой переменной задается в килобайтах (K) или мегабайтах (M). По умолчанию значение этой переменной равно `1M`.
- Переменная окружения `KMP_STATSFILE` задает имя файла статистики при использовании настройки `-cluster-openmp-`

`profile`. По умолчанию имя этого файла `guide.gvs`.

- Задание значения переменной окружения `KMP_WARNINGS` выключает ( `0` или `off` ) или включает ( `1` или `on` ) выдачу сообщений библиотеки выполнения (run-*time library*) Cluster OpenMP. По умолчанию значение этой переменной равно `1` или `on`.
- Задание значения переменной окружения `KMP_WARNINGS` выключает ( `0` или `off` ) или включает ( `1` или `on` ) выдачу предупреждающих сообщений о переменных в параллельных потоках, которые должны быть общими для всех узлов кластера ( `sharable` ), но не являются таковыми. По умолчанию значение этой переменной равно `0` или `off`.
- Переменная окружения `KMP_CLUSTER_SETTINGS` задает режим вывода переменных окружения и настроек, определенных в `kmp_cluster.ini` файле. Значения по умолчанию для этой переменной нет.
- Переменная окружения `KMP_CLUSTER_PATH` задает путь к `kmp_cluster.ini` -файлу, если он отсутствует в текущем разделе. По умолчанию в качестве этого пути задается путь к домашнему разделу пользователя.
- Задание переменной окружения `KMP_CLUSTER_HELP` позволяет выводить описание настроек, допустимых в `kmp_cluster.ini` -файле. Значения по умолчанию для этой переменной нет.
- Переменная окружения `KMP_CLUSTER_DEBUGGER` задает имя исполняемого файла отладчика (например, `idb` для отладчика Intel). Значения по умолчанию для этой переменной нет.
- Переменная окружения `KMP_CLUSTER_VERSION` задает режим вывода сообщений о версии системы. Значения по умолчанию для этой переменной нет.
- Переменная окружения `KMP_DISJOINT_HEAPSIZE` задает значение размера динамической памяти в параллельных потоках и включает механизм Disjoint Heap. Величина этой переменной задается в килобайтах (K) или мегабайтах (M). По умолчанию значение этой переменной равно `2M`. Если заданное значение меньше `2M`, то оно автоматически увеличивается до `2M`.

На этом завершается знакомство с Cluster OpenMP. Отметим, что для изучения дополнительных материалов, оставшихся за рамками данного

изложения, следует обратиться к руководству [7.9].

Список литературы

1. Воеводин Вл.В., Капитонова А.П, *Методы описания и классификации вычислительных систем*, М.: Издательство МГУ, 1994, 79 с

2. Королев Л.Н, *Архитектура ЭВМ*, М.: Научный мир, 2005, 272 с

3. Бройдо В.Л., Ильина О.П, *Архитектура ЭВМ и систем*, СПб.:Питер, 2005, 720 с

4. *Технологии грид. Том. 1*, М.: ИПМ им. М.В. Келдыша, 2006. 380 с

5. *Технологии грид. Том. 2*, М.: ИПМ им. М.В. Келдыша, 2006. 196 с

6. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, URL: http://www.openmp.org/drupal/node/view/8 , October 1997, 16 p

7. *OpenMP: Fortran Application Program Interface Version 1.0*, URL: http://www.openmp.org/drupal/node/ view/8 , October 1997, 55 p

8. *OpenMP: C and C++ Application Program Interface Version 1.0, October 1998, Document Number 004 – 2229 – 001, 77 p*, URL: http://www.openmp.org/drupal/node/view/8 , October 1998, Document Number 004 – 2229 – 001, 77 p.

9. *OpenMP: Fortran Application Program Interface Version 2.0*, URL: http://www.openmp.org/drupal/ node/view /8 , November 2000, 124 p

10. *OpenMP: C and C++ Application Program Interface Version 2.0*, URL: http://www.openmp.org/drupal/ node/view/8 , March 2002, 106 p

11. *OpenMP: Application Program Interface Version 2.5*, URL: http://www.openmp.org/drupal/node/view/8) , May 2005, 250 p

12. Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001, 248 p

13. Quinn M.J, *Parallel Programming in C with MPI and OpenMP*, McGrawHill, 2004, 544 p

14. Satoh S.S., Kusano K., and Tanaka Y, *Design of OpenMP compiler for an SMP cluster*, EWOMP'99, Lund, Sweden, October, 1999

15. Teresco J.D., Devine K.D., and Flaherty J.E, *Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations*, Numerical Solution of Partial Differential Equations on Parallel Computers, Editors: Are Magnus Bruaset, Petter Bjorsatad, Aslak Tveito. Springer-Verlag, 2005

16. Biswas R., Das S.K., Harvey D., and Oliker L, *Portable Parallel Programming for the Dynamic Load Balancing of Unstructured Grid Applications*, URL: http://crd.lbl.gov/~oliker/papers/ipps99.pdf , 13th International Parallel Processing Symposium, 1999, 5 p

17. Wilson L.F. and Shen W, *Experiments in Load Migration and Dynamic Load Balancing in Speeds*, Proceedings of the 1998 Winter Simulation Conference, 1998, pp. 483-490, Editors: D.J. Medeiros, E.F. Watson, J.S. Carson, and M.S.Manivannan

18. Reddy H, *Performance Evaluation of Static and Dynamic Load Balancing Schemes for a Parallel Computational Fluid Dynamics Software (CFD) Application (FLUENT) Distributed across Clusters of Heterogeneous Symmetric Multiprocessor Systems*, URL: http://www.redbooks.ibm.com/abstracts/redp3930.html , IBM RED Paper-3930, 2004, 22 p

19. Devine K.D., Boman E.G., Heaphy R,T., Hendrickson B.A., Teresco J.D., Faik J., Flaherty J.E., Gervasio L.G, *New challenges in dynamic load balancing*, Previously published as Williams College Department of Computer Science Technical Report CS-04-02, and Sandia Report SAND 2004-1496J, Sandia National Laboratories, 2004. Appl. Numer. Maths. Vol. 52,

Issues 2-3, 2005, pp. 133-152

20. Flaherty J.E., Loy R.M, Ozturan C., Shephard M.S., Szymanski B.K.,Teresco J.D., Ziantz L.H, *Parallel Structures and Dynamic Load Balancing for Adaptive Finite Element Computation*, Appl. Numer. Maths., Vol. 26, 1998, pp. 241-263.

21. Shekhar S., Ravada S., Turner G., Chubb D., and Kumar V, *Load Balancing in High Performance GIS: Partitioning Polygonal Maps*, Proc. Int. Symp. on Large Spatial Databases, Springer Verlag (Lecture Notes in Computer Science), (1995)

22. Karypis G., and Kumar V, *A fast and high quality multilevel scheme for partitioning irregular graphs*, TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995

23. Малышкин В.Э, *Основы параллельных вычислений: Учебное пособие / Часть 2*, URL: http://ssga.ru/metodich/paral2/contents.html , Новосибирск: Изд-во НГТУ, 1999. 51 с

24. Chetverushkin B.N., Iakobovsky M.V., Kornilina M.A., and Sukov S.A, *Methane combustion simulation on multiprocessor computer systems. In: Mathematical Modeling. Problems, methods, applications*, Proc. Of the Fourth International Mathematical Modeling Conference, June 27 July 1 2000, Moscow, Russia (Editors: L.A. Uvarova, A.V. Latyshev), Kluwer Academic, Plenum Publishers, New York, Boston, Dordrecht, London, Moscow, 2001, pp. 53-59

25. Fagg G.E., London K., and Dongarra J.J, *Taskers and General Resource Manager: PVM supporting DCE Process Management*, Proceeding of the Third EuroPVM Group Meeting, Munich, Springer Verlag, October 1996

26. Fagg G.E., and Williams S.A, *Improved Program Performance using a cluster of Workstations*, Parallel Algorithms and Applications, Vol. 7, 1995

27. Fagg G.E., Dongarra J.J., and Geist A, *Heterogeneous MPI Application Interoperation and Process Management under PVM-MPI*, University of Tennessee Computer Science Technical Report, CS-97, June 1997, 8 p

28. Amdahl G, *Validate of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, AFIPS Conference Proceedings, Vol. 30, 1967, p. 483

29. *Cluster OpenMP User's Guide*, Intel Corporation, Document Number: 309076-002, February 2006

# Содержание