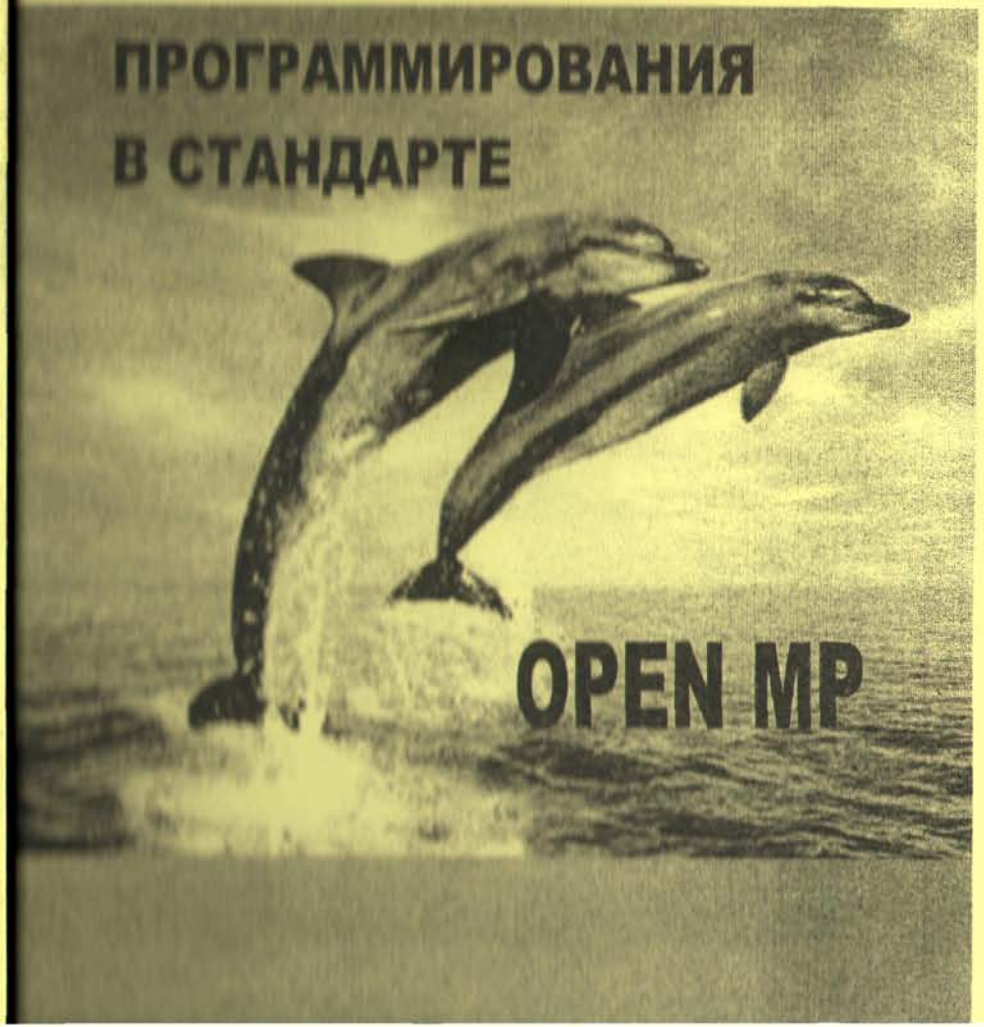
Н.Н. Тунгатаров
Г.М. Даирбаева

ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СТАНДАРТЕ OPEN MP



КАЗАХСКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ имени АЛЬ-ФАРАБИ

Н.Н. Тунгатаров, Г.М. Даирбаева

ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СТАНДАРТЕ OPEN MP

Учебное пособие

Алматы
«Қазақ университеті»
2008

ББК 32. 97
Т 88

Рекомендовано к изданию
Ученым советом механико-математического факультета
и РИСО КазНУ им. аль-Фараби

Р е ц е н з е н т ы:

доктор физико-математических наук, профессор ***У.С. Абдибеков;***
кандидат физико-математических наук, профессор ***М.Р. Елешев***

Тунгатаров Н.Н., Даирбаева Г.М.

Т 88 Основы параллельного программирования в стандарте OPEN MP: Учебное пособие. – Алматы: Қазақ университеті, 2008. – 82 с.

ISBN 9965-30-699-0

В данном пособии рассматриваются основы параллельного программирования в стандарте OPEN MP, который является новейшей технологией параллельного программирования для систем с общей памятью.

Пособие, предназначенное для студентов старших курсов, магистрантов, аспирантов и научных работников, позволит освоить параллельное программирование как в процессе плановой учебной работы, так и в процессе самостоятельной работы.

Т $\frac{\text{1602070000-265}}{\text{460(05)-08}}$ **036-07** **ББК 32. 97**

ISBN 9965-30-699-0

Предисловие

В последние годы технологии компьютерной архитектуры и индустрии значительно усилилось. Это является следствием его постоянного повышенного спроса для решения сложных фундаментальных и прикладных задач в производстве, экономике и науке. Одним из приоритетных спектров его развития является создание и совершенствование высокопроизводительных вычислительных систем, суперкомпьютеров. Наряду с ним развилась область параллельных вычислений, которая стала новым направлением в программировании. Сфера применения параллельных вычислений делает его перспективной и актуальной в компьютерных и вычислительных технологиях.

Данное учебное пособие явилось результатом чтения лекции по параллельному программированию для студентов механико-математического факультета КазНУ имени аль-Фараби. Большинство литературы по параллельным вычислениям написаны на английском языке, трудоемки по содержанию и громоздки по объёму. Многие из них ориентированы на научных сотрудников и опытных программистов. В частности в них рассматриваются вопросы архитектуры вычислительных систем и общие вопросы параллельных вычислений. Многие зарубежные и российские ученные значительно уделили вопросам технологии параллельного программирования в стандарте MPI (The Message Passing Interface) для систем с распределенной памятью, которые требуют специализированных вычислительных систем в виде кластеров. И сравнительно мало уделено вопросам стандарта OpenMP (Open specifications for MultiProcessing) для систем с разделенной памятью. Кроме этих технологии параллельного программирования имеются стандарты POSIX Pthread, HPF (High Performance Fortran), библиотека PVM (Parallel Virtual Machine), EXPRESS, система DVM (Distributed Virtual Machine), система Linda, система FLOWer и другие технологии, которые реализованы на уровне операционных систем, библиотек, специализированных языках, например, Concurrent C++, Fortran M, Норма, Fortran-GNS, Fortran-DVM, C-DVM, mpC, T-система.

Настоящее учебное пособие предназначено для ознакомления и обучения основам параллельного программирования в стандарте Open MP.

Стандарт OpenMP является новой технологией параллельного программирования для систем с общей памятью, в частности, персональные компьютеры. Стандарт OpenMP реализован для языков

Fortran и C/C++. Технология OpenMP состоит из спецификации набора директив компилятора, подпрограмм и переменных среды.

Учебное пособие построено в виде лекции, которые разделены по темам. Каждая лекция рассчитана на 50 минут занятий. Приводятся примеры на языках Fortran и C/C++.

Пособие предназначено для студентов старших курсов, магистрантов, аспирантов и научных работников. Пособие позволяет научиться параллельному программированию, как в процессе плановой учебной работы, так и в процессе самостоятельной работы.

Лекция №1. Введение в параллельное программирование и вычисление

Введение в параллельное программирование

В наше время круг задач, требующих для своего решения применения мощных вычислительных ресурсов, еще более расширился. Это связано с тем, что произошли фундаментальные изменения в самой организации научных исследований. Вследствие широкого внедрения вычислительной техники значительно усилилось направление численного моделирования и численного эксперимента. Стало возможным моделировать в реальном времени процессы интенсивных физико-химических и ядерных реакций, глобальные атмосферные процессы, процессы экономического и промышленного развития регионов и т.д. Очевидно, что решение таких масштабных задач требует значительных вычислительных ресурсов.

Вычислительное направление применения компьютеров всегда оставалось основным двигателем прогресса в компьютерных технологиях. В качестве основной характеристики компьютеров используется такой показатель, как *производительность* - величина, показывающая, какое количество арифметических операций он может выполнить за единицу времени. Именно этот показатель с наибольшей очевидностью демонстрирует масштабы прогресса, достигнутого в компьютерных технологиях. Производительность компьютеров улучшилось с увеличением скорости работы микропроцессоров и максимально широким распараллеливанием обработки данных.

Впервые идею параллельной обработки данных как мощного резерва увеличения производительности вычислительных аппаратов была высказана Чарльзом Бэббиджем, однако уровень развития технологий середины 19-го века не позволил ему реализовать эту идею.

Параллельный компьютер – это набор процессоров, которые могут работать совместно для решения общей задачи.

Развитие компьютерной архитектуры и сетевых технологий, а также появление все более новых, требующих громадной массы вычислений, научных и прикладных задач показало актуальность и перспективность *параллельных вычислений*, выдвинув её на одно из

центральных мест в программировании и вычислительных технологиях.

Применение параллельных вычислительных систем (ПВС) является стратегическим направлением развития вычислительной техники. Это обстоятельство вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным существованием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно.

Параллельные вычисления используются в таких направлениях, как

- моделирование климата,
- генная инженерия,
- проектирование интегральных схем,
- анализ загрязнения окружающей среды,
- создание лекарственных препаратов,
- нейронные и клеточно-нейронные сети,
- механика сплошных сред и физика плазмы,
- автомобилестроение,
- нефте- и газодобыча,
- фармакология.

Обзор технологии параллельного программирования

За годы существования параллельного программирования появились разные разработки и технологии её реализации. Рассмотрим наиболее популярные средства:

1. Стандарт **MPI** (The Message Passing Interface) является библиотекой функций обмена данными между процессами, реализованная для языков C/C++ и Fortran. Головной организацией проекта MPI является Аргоннская национальная лаборатория США. После появления первой версии стандарта MPI в мае 1994 года MPI получил широкое распространение. В настоящее время стандарт MPI адаптирован для большинства суперЭВМ и кластеров. Благодаря простоте технической реализации кластеров на базе локальных сетей сотни университетов используют MPI для учебных и научных целей.

2. Стандарт **OpenMP** (Open specifications for MultiProcessing, открытая спецификация мультиобработки данных) совместно разработан в октябре 1997 года компаниями Silicon Graphics, Kuck & Associates, Digital, IBM и Intel, поддерживает платформы Unix и

Windows NT. В настоящее время OpenMP стал одним из наиболее популярных средств программирования компьютеров с общей памятью. Стандарт OpenMP базируется на языках программирования Fortran, C/C++. За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется набор директив, процедур и переменных окружения, которые вставляет в виде специальных комментариев. Процесс выполнения OpenMP-программы представляют цепочка регионов:

3. Стандарт **HPF** (High Performance Fortran, высокопроизводительный Fortran) разработан в 1993 году компаниями Convex, DEC, IBM, Sun Microsystems основывается на параллелизме данных и модели обработки данных архитектуры SIMD. Стандарт HPF реализует идею инкрементного распараллеливания и модель общей памяти на системах с распределенной памятью. HPF разработан для языка Fortran в виде комментариев, объединяет набор директив разделения данных, инструкции параллелизма и подпрограммы распределения данных.

4. Коммуникационная библиотека **PVM** (Parallel Virtual Machine) разработана лабораторией Oak Ridge National Laboratory для систем с распределенной памятью. Библиотека PVM реализуется на механизме передачи сообщений.

5. Система **DVM** (Distributed Virtual Machine) создано Институтом прикладной математики им. М.В. Келдыша РАН для систем с

распределенной памятью. Система **DVM** поддерживает языки Фортран, C/C++.

6. Система **Linda**. В системе Linda параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно последовательной программе. Все процессы имеют доступ к общей памяти (пространство кортежей), единицей хранения данных является кортеж (упорядоченная последовательность значений). Все процессы работают с пространством кортежей по принципу: поместить кортеж, захватить, скопировать. В отличие от традиционной памяти, если процесс захватывает кортеж из пространства кортежей, то он становится недоступным другим процессам. Если кортеж поместить в пространство кортежей с именем, совпадающим с другим кортежем, то не происходит обновления значения переменной - в пространстве кортежей окажется два кортежа с одинаковыми именами. В отличие от традиционной памяти, изменить кортеж прямо в пространстве нельзя. Для этого нужно сначала этот кортеж захватить, затем изменить его значение и вновь обратно поместить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение происходит через пространство кортежей.

7. Система **FLOWer** используется для систем с распределенной памятью (MPP). Параллельная программа создается с помощью графа потока данных (ГПД) записанная на специальном языке DGL (Dataflow Graph Language). Процессы записываются в виде отдельных функций, которые считывают значения параметров из входных дуг ГПД и передают результат в выходные. Использование модели управления потоком данных позволяет избежать сложностей, связанных с синхронизацией, но это снижает эффективность алгоритма.

Лекция №2. Классификация параллельных вычислительных систем

Основные определения:

Архитектура компьютера - описание компонент компьютера и их взаимодействия.

Организация компьютера - описание конкретной аппаратной реализации архитектуры, ее воплощение "в железе".

Схема компьютера - детальное описание электронных компонент компьютера, их соединений, устройств питания, охлаждения и пр.

flops - единица измерения производительности вычислительных систем - количество операций с плавающей точкой, выполняемых за 1 секунду, например 1 Tflops равен одному триллиону операций в секунду.

Классификация Шора

Классификация Дж. Шора (начало 70-х гг.) базируется на выделении типичных способов компоновки вычислительных систем (ВС) на основе фиксированного числа базисных блоков: устройства управления (***УУ***), арифметико-логического устройства (***АЛУ***), памяти команд (***ПК***) и памяти данных (***ПД***). Предполагается, что выборка из памяти данных может осуществляться словами (выбираются все разряды одного слова – "горизонтальная выборка"), и/или битовым слоем (по одному разряду из одной и той же позиции каждого слова – "вертикальная выборка").

Машина I (рисунок 1) – вычислительная система, содержащая УУ, АЛУ, ПК и ПД с пословной выборкой. Считывание данных осуществляется выборкой всех разрядов некоторого слова для их параллельной обработки в арифметико-логическом устройстве, т.е. выполняется последовательная обработка слов при параллельной обработке разрядов. Состав АЛУ может состоять из нескольких функциональных устройств, в т.ч. конвейерного типа. Представители: классические последовательные машины (IBM 701, PDP-11), конвейерные скалярные (CDC 7600) и векторно-конвейерные (CRAY-1).

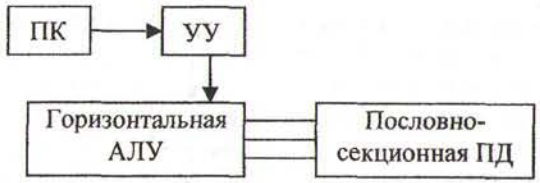


Рисунок 1. Машина I.

Машина II (рисунок 2) – осуществляется выборка содержимого одного разряда из всех слов, т.е. последовательная обработка битовых слоев при параллельной обработке множества слов. Компьютеры на основе структуры машины II, имеют множество сравнительно простых

АЛУ поразрядной обработки. Примеры: центральный процессор машины STARAN, матричная система ICL DAP.

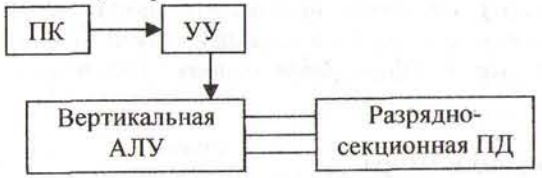


Рисунок 2. Машина II.

Машина III (рисунок 3) – объединяет принципы построения машин I и II, т.е. имеет два АЛУ и модифицированную память данных, которая обеспечивает доступ, как к словам, так и к битовым слоям. Представитель – вычислительная система OMEN-60.

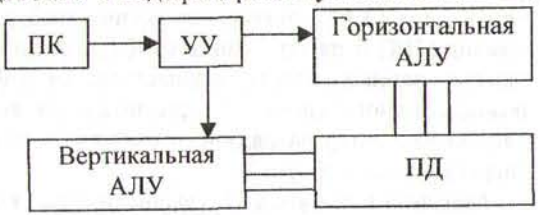


Рисунок 3. Машина III.

Машина IV (рисунок 4) имеет несколько пар процессорных элементов АЛУ-ПД. Единственное устройство управления выдает команду сразу всем процессорным элементам. Преимущество – простота увеличения числа процессорных элементов за счет отсутствия соединений между ними. Недостаток – ограниченность применимости таких машин. Представитель – вычислительная система PEPE.

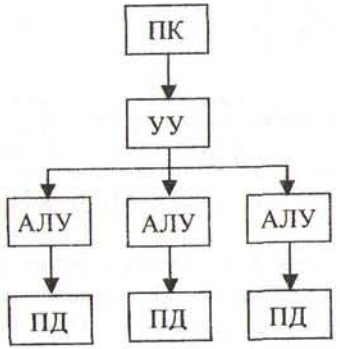


Рисунок 4. Машина IV.

Машина V (рисунок 5) – имеет непосредственные линейные связи между соседними процессорными элементами машины, например, в виде матричной конфигурации. Любой процессорный элемент может обращаться к данным, как в своей памяти, так и в памяти непосредственных соседей. Пример – классический матричный компьютер ILLIAC IV.

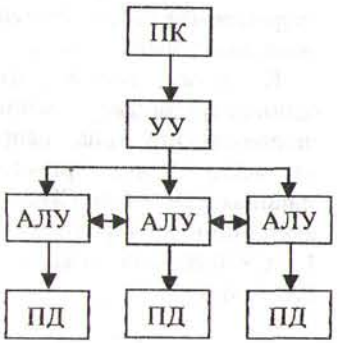


Рисунок 5. Машина V.

Все машины с I-й по V-ю предполагают наличие шины данных или какого-либо коммутирующего элемента между ПД и АЛУ.

Машина VI (рисунок 6) - матрица с функциональной памятью или память со встроенной логикой, подразумевает распределение логики процессора по всему запоминающему устройству. Примерами могут служить как простые ассоциативные запоминающие устройства, так и сложные ассоциативные процессоры.

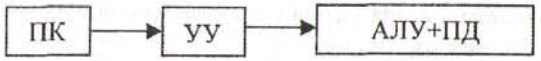


Рисунок 6. Машина VI.

Классификация Флинна

Классификация Майкла Флинна является одной из самых ранних и наиболее известных классификаций архитектур ВС, предложенная в 1966 году. Классификация базируется на понятии *потока*, под которым понимается последовательность команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных выделяется четыре класса архитектур: **SISD, MISD, SIMD, MIMD**.

Обозначим: ПР – один или несколько процессорных элементов, УУ – устройство управления, ПД – память данных.

SISD (Single Instruction stream – Single Data stream) – **ОКОД** (одиночный поток команд и одиночный поток данных) (рисунок 7). В таких машинах в каждый момент времени выполняется лишь одна

операция над одним элементом данных. Все команды обрабатываются последовательно.

К этому классу относятся классические последовательные однопроцессорные машины фон-неймановского типа, например, PDP-11, VAX 11/780, персональные ЭВМ, машины CDC 6600, CDC 7600, а также векторно-конвейерные машины CRAY-1, CYBER 205, машины семейства FACOM VP.

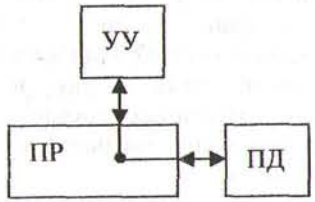


Рисунок 7. Архитектура SISD.

SIMD (Single Instruction stream – Multiple Data stream) – **ОКМД** (одиночный поток команд – множественный поток данных) (рис. 8). Поток команд включает векторные команды, что позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора.

SIMD компьютеры состоят из одного командного процессора (контроллера) и нескольких процессорных элементов. Контроллер принимает, анализирует и выполняет команды. Если в команде встречаются данные, контроллер рассылает команду на все процессорные элементы и она выполняется на нескольких или на всех процессорных элементах. Одно из преимуществ данной архитектуры – более эффективно реализована логика вычислений. В SIMD компьютере управление выполняется контроллером, а "арифметика" отдана процессорным элементам.

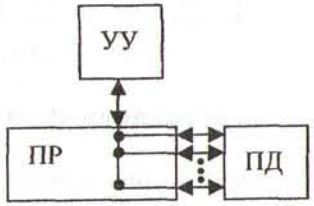


Рисунок 8. Архитектура SIMD.

MISD (Multiple Instruction stream – Single Data stream) – МКОД (множественный поток команд – одиночный поток данных) (рисунок 9). Класс подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Вычислительных машин такого класса практически нет и трудно привести пример их успешной реализации.

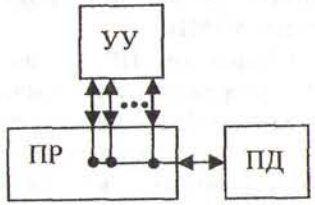


Рисунок 9. Архитектура MISD.

MIMD (Multiple Instruction stream – Multiple Data stream) – МКМД (множественный поток команд и множественный поток данных) (рисунок 10). Класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

Класс MIMD чрезвычайно широк, в него попадают симметричные параллельные вычислительные системы, рабочие станции с несколькими процессорами, кластеры рабочих станций, всевозможные мультипроцессорные системы: Cm*, CRAY T3D, Intel Paragon и многие другие. В начале 90-х годов MIMD компьютеры выходят в лидеры на рынке высокопроизводительных вычислительных систем.

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера.

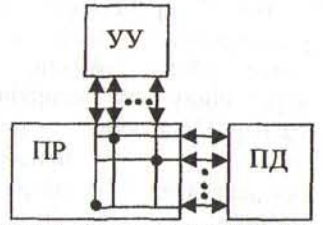


Рисунок 10. Архитектура MIMD.

Классификация Хокни

Р. Хокни разработал иерархическую структуру для систематизации архитектур вычислительных систем (рисунок 11), относящихся, по классификации Флинна, к классу MIMD.

Рисунок 11.
Классификация Хокни

Классификация по способу взаимодействия процессоров с оперативной памятью

Выделяют три основные группы архитектур:

1. Системы с общей (разделяемой) памятью (**Shared Memory Systems - SMP**) или симметричные мультипроцессорные системы (***Symmetric Multi Processors - SMP***). Современные системы SMP архитектуры состоят из нескольких однородных микропроцессоров и массива общей памяти (рисунок 12).

Все процессоры имеют равноправный доступ к любой точке общей памяти, обычно через шину или иерархию шин. Распараллеливание процессов между процессорами осуществляет операционная система.

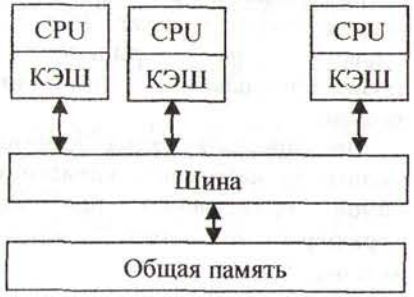


Рисунок 12. Архитектура SMP.

Достоинства: простота взаимодействия процессоров между собой за счет общей памяти.

Недостатки:

- Проблема конфликтов при обращении к общей шине памяти;
- Проблема, связанная с иерархической структурой организации памяти современных компьютеров. Дело в том, что самым узким местом в современных компьютерах является оперативная память, скорость работы которой значительно отстала от скорости работы процессора.

2. Системы с распределенной памятью или массивно параллельные процессоры (**Massively Parallel Processors – MPP**). Каждый процессор снабжается собственной локальной памятью. Прямой доступ к памяти других процессоров невозможен.

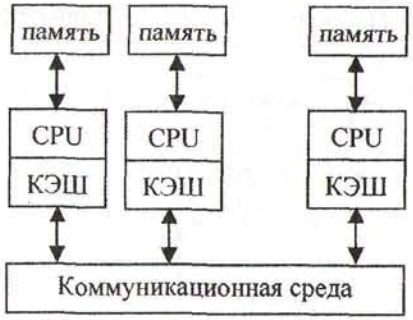


Рисунок 13. Архитектура MPP.

Компьютеры этого типа представляют собой многопроцессорные системы с распределенной памятью, в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы (рисунок 13). Каждый узел может включать несколько процессоров по архитектуре SMP.

3. Системы с распределенно–разделяемой памятью или компьютеры с неоднородным доступом к памяти (***Non Uniform Memory Access - NUMA***). NUMA-архитектуры представляют собой нечто среднее между SMP и MPP: в них память физически распределена, но логически общедоступна.

Главное различие между МКМД-ЭВМ с общей и индивидуальной памятью в характере адресной системы. В машинах с разделяемой памятью адресное пространство всех процессоров является единым, следовательно, если в программах нескольких процессоров встречается одна и та же переменная *X*, то эти процессоры будут обращаться в одну и ту же физическую ячейку общей памяти. Это вызовет как положительные, так и отрицательные последствия.

Наличие общей памяти не требует физического перемещения данных между взаимодействующими программами, которые параллельно выполняются в разных процессорах. Это упрощает программирование и исключает затраты времени на межпроцессорный обмен.

Однако одновременное обращение нескольких процессоров к общим данным может привести к получению неверных результатов.

Рассмотрим пример. Пусть имеется система с разделяемой памятью и двумя процессорами, каждый с одним регистром. Пусть в первом процессоре выполняется процесс *L1*, во втором – *L2*:

L1: ... X:=X+1; ...
L2: ... X:=X+1; ...

Процессы выполняются асинхронно, используя общую переменную *X*. При выполнении процессов возможно различное их взаиморасположение во времени, например, возможны две ситуации:

L1: R1:=X; R1:= R1+1; X:= R1; ...
L2: R2:=X; R2:= R2+1; X:= R2; ... *(А)*

L1: R1:=X; R1:= R1+1; X:= R1; ...
L2: R2:=X; R2:= R2+1; X:= R2; ... *(Б)*

Пусть в начальный момент *X=V*. Тогда в случае (А) второй процессор производит чтение *X* до завершения всех операций в первом процессоре, поэтому *X=V+1*.

В случае (Б) второй процессор читает X после завершения всех операций в первом процессоре, поэтому $X=V+2$.

Таким образом, результат зависит от взаиморасположения процессоров во времени, что для асинхронных процессов определяется случайным образом. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов, например, семафоры.

Поскольку при выполнении каждой команды процессорам необходимо обращаться в общую память, то требования к пропускной способности коммутатора этой памяти высоки, что ограничивает число процессоров в системах величиной 10…20.

В системах с индивидуальной памятью каждый процессор имеет независимое адресное пространство, и наличие одной и той же переменной X в программах разных процессоров приводит к обращению в физически разные ячейки памяти. Это вызывает физическое перемещение данных между взаимодействующими программами в разных процессорах. Однако, основная часть обращений производится каждым процессором в собственную память, то требования к коммутатору уменьшаются, и число процессоров в системах разделенной памятью и коммутатором типа гиперкуб может достигать нескольких сотен и даже тысяч.

Лекция № 3. Введение в технологию программирования OpenMP

OpenMP (Open specifications for MultiProcessing, открытая спецификация мультиобработки данных) – программный интерфейс (API) для программирования компьютеров с разделяемой памятью (SMP/NUMA). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения. Спецификации для языка Fortran появились в октябре 1997 года, а для C/C++ в октябре 1998 года.

OpenMP предоставляет следующие преимущества разработчику:

1. *Инкрементальное распараллеливание* – обеспечивает разработчикам быстро распараллелить вычислительные программы с

большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

2. *Гибкий механизм* – предоставляет разработчику большие возможности контроля над поведением параллельного приложения.

3. *Последовательность программы* – OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.

4. *Оторванные директивы* – достоинство OpenMP его разработчики считают поддержку так называемых "orphan" (оторванных) директив, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

Понятие 1. **Нить** (Thread, light-weight process – легковесный процесс, поток управления) – это независимый поток управления с собственным счетчиком команд, выполняемый в контексте некоторого процесса.

Понятие 2. **Семафоры** – это переменная S, связанная с некоторым ресурсом и принимающая два состояния: 0 (запрещено обращение) и 1 (разрешено обращение). Над S определены две операции: X и Y. Операция X изменяет значение S семафора на значение S+1. Действие операции Y таково:

1. если S≠0, то Y уменьшает значение на единицу.
2. если S=0, то Y не изменяет значение S и не завершается до тех пор, пока некоторый другой процесс не изменит значение S с помощью операции X.

Операции X и Y считаются неделимыми, т.е. не могут исполняться одновременно.

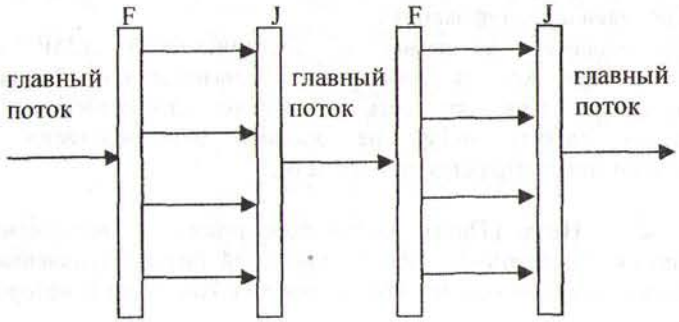
Технология OpenMP опирается на понятие общей памяти ориентированные на SMP-компьютеры, которые эффективно поддерживают нити.

Fork-Join модель. OpenMP использует Fork-Join модель параллельного выполнения. Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свои уникальный номер, причем нить-мастер

всегда имеет номер 0. Все нити исполняют один и тот же код, параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

Fork (Ветвление): главный поток создает группу параллельных потоков, все они параллельно выполняют инструкции в программе, которые включены в область параллельной конструкцией.

Join (Объединение): все потоки группы завершают инструкции в области параллельной конструкции, они синхронизируются и закрываются, оставляя только главный поток.



Явный параллелизм. OpenMP имеет явную (не автоматический) модель программирования, предлагая программисту полное управление по распараллеливанию.

Базовый потоковый параллелизм. OpenMP основан на существовании множественных потоков в общедоступной памяти, программирующей парадигму. Общедоступный процесс памяти может состоять из множественных потоков, несколько нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити.

Кратко о переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

- SHARED(*список*) – общие переменные, под именем А все нити видят одну переменную;
- PRIVATE(*список*) – приватные переменные, под именем А каждая нить видит свою переменную (локальную),

где *список* – переменные, разделенные запятыми.

Кроме них, используются REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN, которые определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла.

По умолчанию, все COMMON-блоки, а также переменные, порожденные вне параллельной области, при входе в эту область остаются общими (SHARED). Исключение составляют переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (PRIVATE). Явно назначить класс переменных по умолчанию можно с помощью клаузы DEFAULT.

Кратко о директивах

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов:

!$OMP – свободный и фиксированный формат,

C$OMP – фиксированный формат,

*$OMP – фиксированный формат.

Все директивы OpenMP можно разделить на три основные категории:

- директива определения параллельной секции;
- директивы разделения работы;
- директивы синхронизации.

Директивы могут сопровождаться одним или несколькими дополнительными атрибутами - *клауз*. Клаузы определяют поведение переменных в параллельных конструкциях.

Рассмотрим только два директив PARALLEL и DO

Директива PARALLEL – определяет параллельную область программы. При входе в эту область главная нить порождает N-1 новых нитей, образуется "команда" из N нитей. Главная нить получает номер 0 и становится основной нитью (master-ом) команды. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Форма записи директивы:

!$OMP PARALLEL [список клауз]

замкнутый блок параллельной секции

!$OMP END PARALLEL

Директива **DO** определяет параллельный цикл и имеет следующую форму записи:

!$OMP DO [список клауз]
замкнутый цикл
!$OMP [ENDDO]

Если внутри PARALLEL содержится только одна конструкция DO, то можно использовать укороченную запись: *!$OMP PARALLEL DO*.

В OpenMP с помощью функций OMP_GET_THREAD_NUM() и OMP_GET_NUM_THREADS нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера.

Спецификация OpenMP для языков C/C++

Спецификация OpenMP для C/C++ содержит в основном аналогичную функциональность как в Fortran.

Отличие заключается в следующих моментах:

1) Вместо спецкомментариев используются директивы компилятора *#pragma omp*.

2) Компилятор с поддержкой OpenMP определяет макрос "_OPENMP", который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы.

3) Распараллеливание применяется к for-циклам, для этого используется директива *#pragma omp for*. В параллельных циклах запрещается использовать оператор *break*.

4) Статические (*static*) переменные, определенные в параллельной области программы, являются общими (*shared*).

5) Память, выделенная с помощью *malloc()*, является общей, но указатель на нее может быть как общим, так и приватным.

6) Типы и функции OpenMP определены во включаемом файле *<omp.h>*.

7) Кроме обычных, возможны также "вложенные" (*nested*) замки - вместо логических переменных используются целые числа, и нить, уже захватившая замок, при повторном захвате может увеличить это число.

Фрагмент распараллеливания DO-цикла на языке FORTRAN

```
!$OMP PARALLEL DO PRIVATE(i)
!$OMP& SHARED(x,y,n) REDUCTION(+:a,b)
DO i=1,n
a=a+x(i)
b=b+y(i)
END DO
```

Аналогично к этому for-цикла на языке C

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
   a = a + x[i];
   b = b + y[i];
}
```

Пример программы

Скалярное параллельное произведение элементов двух векторов. Суммирование осуществляется в цикле. В программе применяется комбинация параллельного цикла **PARALLEL DO** и редуцированной операции **REDUCTION** по всем процессам.

```
PROGRAM MAIN
PARAMETER (N=100)
DOUBLE PRECISION A(N),B(N),S
INTEGER I
! Инициализация элементов векторов
DO I=1,N
A(I)=2.0*I
B(I)=A(I)-1
END DO
S=0.0
! Создание множества параллельных процессов и
! распараллеливание цикла по виткам. При выходе из цикла все
! значения переменной S суммируются по всем процессам.
!$OMP PARALLEL DO REDUCTION(+:S)
DO I=1,N
S=S+A(I)*B(I)
END DO
```

```
! Главный процесс выводит на экран значение S
PRINT *, 'S=',S
STOP
END
```

На языке С программа будет иметь вид:

```
#include <omp.h>
#include <iostream.h>
#define N 100
void main()
{
int i;
double A[N], B[N], S;

// Инициализация элементов векторов
for (i=0;i<N;i++)
  {
   A[i]= i*2.0; B[i]=A[i]-1;
  }
S=0.0;

/* Создание множества параллельных процессов и
* распараллеливание цикла по виткам. При выходе из цикла все
* значения переменной S суммируются по всем процессам. */

#pragma omp parallel for reduction(+:S)
  for (i=0;i<N;i++)
    S += (A[i]*B[i]);

// Главный процесс выводит на экран значение S
cout << "S=" << S << "\n";
}
```

Лекция № 4. Директивы OpenMP

Общая форма записи директив в OpenMP, выглядит:

стартовый_знак имя_директивы [список клауз]
замкнутый_блок_кода
стартовый_знак END имя_директивы [модификатор завершения]

Форма *стартового знака* для языка FORTRAN !OMP, cOMP, *$OMP, для языка C/C++ #pragma omp. Кроме того, для языка C/C++, необходимо включить библиотеку omp.h.

1. Директива определения параллельной секции *parallel*

Синтаксис для языка Fortran:

!$OMP PARALLEL [список клауз]
структурный_блок
!$OMP END PARALLEL

Клаузами для директивы *PARALLEL* могут быть следующие директивы:

- IF(скалярное_логическое_выражение),
- PRIVATE(список),
- FIRSTPRIVATE(список),
- DEFAULT(PRIVATE | SHARED |NONE),
- SHARED(список),
- COPYIN(список),
- REDUCTION({оператор | имя_встроенной_процедуры}:список),
- NUM_THREADS(выражение_целого_типа).

Синтаксис для языка C/C++:

#pragma omp parallel [список клауз]
структурный_блок

Клаузами для директивы *parallel* могут быть следующие директивы:

- if(скалярное_выражение),
- private(список),
- firstprivate(список),
- default(shared | none),

- shared(список),
- copyin(список),
- reduction(оператор : список),
- num_threads(целое_выражение)

Директива *parallel* используется, когда для распараллеливания секции применяются несколько директив формирующих процессы или выполняющих другие действия.

2. Директива разделения работы в параллельных циклах do / for

Синтаксис для языка Fortran:

!$OMP DO [список клауз]
замкнутый_цикл
!$OMP END DO [NOWAIT]

Клаузами для директивы *DO* могут быть следующие директивы:

- PRIVATE(список),
- FIRSTPRIVATE(список),
- LASTPRIVATE(список),
- REDUCTION({оператор | имя_встроенной_процедуры}:список),
- ORDERED
- SHEDULE(тип [, размер_порции]).

Директива *DO* используется к циклам, которые не имеют право досрочного выхода за блок директивы. Переменная цикла автоматически становится локальной и не определяется в списке клаузы PRIVATE. Модификатор завершения *NOWAIT* отменяет синхронизацию подпроцессов группы исполнения на выходе из блока директивы *DO.*

Синтаксис для языка C/C++:

#pragma omp for [список клауз]
замкнутый_цикл

Клаузами для директивы *for* могут быть следующие директивы:

- private(список),
- firstprivate(список),
- lastprivate(список),
- reduction(оператор : список),

- ordered,
- shedule(тип [, размер_порции]),
- nowait.

Директива do/for формирует процесс, содержащие копии ассоциированного с директивой цикла типа do/for. Каждая копия будет выполнять свою часть от общего числа итераций, описанных в блоке цикла.

Кроме того, имеется сокращенная форма записи директивы do/for. Используется, когда параллельная секция содержит единственный оператор цикла do/for.

Синтаксис на языке Fortran имеет следующий вид:

!$OMP PARALLEL DO [список клауз]
замкнутый_цикл

Синтаксис на языке C/C++:

#pragma omp parallel for [список клауз]
замкнутый_цикл

3. Директива разделения работы в параллельных секциях *sections*

Синтаксис для языка Fortran:

!$OMP SECTIONS [список клауз]
[!$OMP SECTION]
структурный_блок
[!$OMP SECTION
структурный_блок]
...
!$OMP END SECTIONS [NOWAIT]

Клаузами для директивы *SECTIONS* могут быть следующие директивы:

- PRIVATE(список),
- FIRSTPRIVATE(список),
- LASTPRIVATE(список),
- REDUCTION({оператор | имя_встроенной_процедуры}:список).

Синтаксис для языка C/C++:

#pragma omp sections [список клауз]
{

[#pragma omp section]
структурный_блок
[#pragma omp section
структурный_блок]
...
}

Клаузами для директивы *sections* могут быть следующие директивы:

- private(список),
- firstprivate(список),
- lastprivate(список),
- reduction(оператор:список),
- nowait.

Директива *sections* формирует параллельную секцию из блоков, расположенных в исходном тексте программы последовательно друг за другом. Перед каждым преобразуемым блоком указывается директива *section*.

Section – вспомогательная директива, используется только в области действия директивы *sections* для формирования нитей из ассоциированных блоков.

Кроме того, предусмотрена сокращенная форма записи директивы *sections*, которая используется для создания параллельной секции, содержащего единственную директиву *sections*.

Синтаксис на языке Fortran имеет следующий вид:

!$OMP PARALLEL SECTIONS [список клауз]
замкнутый_цикл

Синтаксис на языке C/C++:

#pragma omp parallel sections [список клауз]
замкнутый_цикл

4. Директива разделения работы для исполнения одной нитью *single*

Синтаксис для языка Fortran:

!$OMP SINGLE [список клауз]
структурный_блок
!$OMP END SINGLE [список_концов_клауз]

Клаузами для директивы *SINGLE* могут быть директивы: PRIVATE(список) и FIRSTPRIVARE(список).

Клаузами концов могут быть директивы: COPYPRIVATE(список) и NOWAIT.

Синтаксис для языка C/C++:

#pragma omp single [список клауз]
структурный_блок

Клаузами для директивы *single* могут быть следующие директивы:

- private(список),
- firstprivate(список),
- copyprivate(список),
- nowait.

Директива *single* указывает на то, что в секции должна выполняться только одна нить, содержащая ассоциированный с директивой блок. Такая нить может, например, изменять значения частных переменных, используемых другими нитями секции.

5.Директива WORKSHARE

Директива применяется к блоку операторов, которые могут быть распределены по подпроцессам автоматически, в соответствии с определенными правилами.

Синтаксис директивы имеет вид:

!$OMP WORKSHARE
структурный_блок
!$OMP END WORKSHARE [NOWAIT]

Для разделения по подпроцесам, структурный блок должен содержать завершенные конструкции: присваивания массивов и скаляров, операторы и конструкции WHERE и FORALL, директивы ATOMIC, CRINICAL, PARALLEL.

Операторы структурного блока будут разделены на элементы исполнения по следующим правилам:

1) В выражениях с массивами в каждом операторе, включая обращения к встроенным функциям редукции массивов, элементами считаются:

- вычисление каждого элемента массивного выражения;
- вычисление встроенных преобразующих функции для массивов может быть разделено на элементы произвольным образом.

2) В присваиваниях массивов каждое присваивание считается отдельным элементом.

3) В обращениях к элементным функциям вычисление значения функции каждого элемента любого аргумента-массива рассматривается как элемент исполнения.

4) В операторе и конструкции WHERE, каждое вычисление маски и каждое присваивание под управлением маски считаются элементами исполнения.

5) В операторе и конструкции FORALL, каждое вычисление маски, вычисление индексных диапазонов и управляемые назначения рассматриваются как элементы исполнения.

6) Все операторы блока директивы ATOMIC рассматриваются как элементы исполнения.

7) Все операторы блока директивы CRINICAL рассматриваются как элементы исполнения.

8) Все конструкции PARALLEL, заключенные в блоке операторов директивы *WORKSHARE*, рассматриваются как элементы исполнения и каждая такая область будет исполняться новой группой подпроцессов исполнения.

К встроенным преобразующим функциям для массивов относятся: MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, SPREAD, PACK, UNPACK, RESHAPE, TRANSPOSE, EOSHIFT, CSHIFT, MINLOG и MAXLOG.

Кроме того, предусмотрена сокращенная форма записи директивы WORKSHARE, которая используется для создания параллельной секции, содержащего единственный блок операторов.

Синтаксис на языке Fortran имеет следующий вид:

!$OMP PARALLEL WORKSHARE [список клауз]
структурный_блок
!$OMP END PARALLEL WORKSHARE

Клаузами могут быть любые клаузы принятые в директиве *PARALLEL* с идентичными значениями и ограничениями. Кроме того, NOWAIT не может, определено в директиве *END PARALLEL WORKSHARE.*

Лекция №5. Директивы синхронизации OpenMP

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.

Один из типов — неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.

Неявная барьерная синхронизация выполняется также в конце каждого блока:

В языке Fortran	*В языке C/C++*
!$OMP DO …!$OMP END DO	#pragma omp for
!$OMP SINGLE …!$OMP END SINGLE	#pragma omp single
!$OMP SECTIONS … !$OMP END SECTIONS	#pragma omp sections

Чтобы отключить неявную барьерную синхронизацию в каком-либо из этих трех блоков разделения работы, укажите раздел nowait:

На языке Fortran	*На языке C/C++*

!$OMP PARALLEL
!$OMP DO
do i=1,size
X(i)=(Y(i-1)+Y(i+1))/2
End do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
	#pragma omp parallel
	{
	#pragma omp for nowait
	for(int i = 1; i < size; ++i)
	x[i] = (y[i-1] + y[i+1])/2;
	}

Как видите, этот раздел директивы распараллеливания говорит о том, что синхронизировать потоки в конце цикла do/for не надо, хотя в конце параллельного региона они все же будут синхронизированы.

Второй тип — явная барьерная синхронизация.

К директивам синхронизации в OpenMP относятся: master, critical, barrier, atomic, flush, ordered.

1. Директива master

Эта директива используется тогда, когда требуется выполнение блока кода только основным потоком, т.е. чтобы какую-то задачу выполнил именно нулевая нить. Синтаксис директивы master:

На языке Fortran	*На языке C/C++*
!$OMP MASTER *структурный блок* !$OMP END MASTER	#pragma omp master *структурный блок*

Остальные процессы пропускают этот блок, и начинают выполнять программу дальше.

Фрагмент программы, вывод номера итерации:

На языке Fortran	*На языке C/C++*
!$OMP MASTER k=k+1 print *, 'Iteration number =', k !$OMP END MASTER	#pragma omp master { ++k cout << "Iteration number =" << k }

2. Директива critical

Эта директива используется тогда, когда требуется разграничить доступ к общему ресурсу, например, к памяти. Директива critical *определяет критическую секцию*, то есть блок кода, который не должен выполняться одновременно двумя или более нитями, а выполнять этот блок кода каждой нитью поочередно. Синтаксис директивы critical:

На языке Fortran	*На языке C/C++*
!$OMP CRITICAL *[(имя)]* *структурный блок* !$OMP END CRITICAL *[(имя)]*	#pragma omp critical *[(имя)]* *структурный блок*

Здесь *имя* –это уникальное имя данной секции.

Все нити параллельного региона ждут завершения выполнения критической секции. Если есть несколько критических секций, то им надо присвоить уникальные имена.

Фрагмент программы, поочередный вывод значений для каждой нити.

```
!$OMP  PARALLEL SHARED(x, y, z) PRIVATE(rank,i)
rank=OMP_GET_THREAD_NUM()
!$OMP DO
do i=1,N
```



```
x(i)=z(i)*z(i)
y(i)=x(i)-z(i)
end do
!$OMP CRITICAL (for_x)
print *, 'rank=', rank, ' i=', i, ' x=', x
!$OMP END CRITICAL (for_x)
!$OMP CRITICAL (for_y)
print *, 'rank=', rank, ' i=', i, ' y=', y
!$OMP END CRITICAL (for_y)
!$OMP END DO
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(x, y, z) private(rank, i)
{    rank=omp_get_thread_num();
     #pragma omp for
{    x[i] = z[i]*z[i];  y[i] = x[i]-z[i];
#pragma omp critical for_x
{ cout << "thread num=" << rank << " i=" << i << " x=" << x[i] }
#pragma omp critical for_y
{ cout << "thread num=" << rank << " i=" << i << " y=" << y[i] }
} // завершение параллельного цикла
} // завершение параллельного региона
```

3. Директива barrier

Эта директива определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных нитей в группе. Синтаксис директивы barrier:

На языке Fortran	*На языке C/C++*
!$OMP BARRIER	#pragma omp barrier

Все нити будут приостановлены в месте директивы, и возобновятся, после того как последняя нить этой группы достигнет её.

4. Директива atomic

Эта директива определяет переменную в левой части оператора "атомарного" присваивания, которая должна корректно обновляться несколькими нитями. Директива обеспечивает порядок последовательного обновления значения переменной в памяти, и не

допускает конкурирующего обращения нитей к записи в одну и туже ячейку памяти.

Синтаксис директивы atomic на языке *Fortran*:

!$OMP ATOMIC

утверждение

Здесь *утверждение* имеет одно из следующих форм:

x = x operator выражение, x = выражение operator x,

x= имя_процедуры(x, список_выражений),

x= имя_процедуры(список_выражений, x),

где *x* - скалярная переменная, *выражение* - скалярное выражение, *список_выражений* – не пустой список скалярных выражений, разделенными запятыми, не ссылающихся на *x*, *operator* – одна из операции +, *, -, /, .AND., .OR., .EQV., .NEQV., *имя_процедуры* – конкретное имя существующей процедуры.

Синтаксис директивы atomic на языке *C/C++*:

#pragma omp atomic

выражение-утверждение

Здесь *выражение-утверждение* имеет одно из следующих форм:

x *binor= выражение*, x++, ++x, x--, --x,

где x – переменная скалярного типа,

binor – один из операции: +, *, -, /, &, ^, |, <<, >>,

выражение - выражение скалярного типа.

5. Директива flush

Эта директива явно определяет точку, в которой реализация должна обеспечить одинаковый вид памяти для всех нитей. Иначе говоря, директива используется, чтобы завершить все незавершенные операции над памятью перед началом следующей операции.

Синтаксис директивы flush:

На языке Fortran	*На языке C/C++*
!$OMP FLUSH *[(список)]*	#pragma omp flush *[(список)]*

Список – перечень разделенных запятыми переменных, требующих подтверждения консистентности. Если список отсутствует, то flush будет применяться ко всем видимым нитью глобальным переменным.

К видимым нитью глобальным переменным относятся следующие категории переменных:

- глобальные переменные (из COMMON-блоков и модулей);
- переменные, ассоциированные через носитель;
- локальные переменные с атрибутом SAVE;

- переменные, ассоциированные в памяти (EQUIVALENCE) с видимыми нити переменными;
- локальные переменные с захваченным адресом;
- локальные переменные без атрибута SAVE, но объявленные разделяемыми во внешнем блоке параллелизации;
- формальные параметры;
- все разыменования ссылок.

Директива flush обеспечивает соответствие между памятью исполняющего потока и глобальной памятью. Чтобы достичь соответствия для всех потоков, необходимо, чтобы все потоки выполнили операцию flush.

Неявно FLUSH присутствует в следующих директивах: BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.

6. Директива ordered

Директива определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле. Может использоваться для упорядочения вывода от параллельных нитей. Синтаксис директивы ordered:

На языке Fortran	*На языке C/C++*
!$OMP ORDERED *структурный_ блок* !$OMP END ORDERED	#pragma omp ordered *структурный блок*

Может использоваться для упорядочения вывода от параллельных нитей.

В целях синхронизации можно также пользоваться механизмом замков (locks).

Примеры программ

Программа на Fortran для директивы *barrier*

```
SUBROUTINE  WORK(N)
INTEGER N
END SUBROUTINE WORK

SUBROUTINE SUB3(N)
INTEGER N
CALL WORK(N)
!$OMP   BARRIER
CALL WORK(N)
```

```
END SUBROUTINE SUB3

SUBROUTINE SUB2(K)
INTEGER K
!$OMP  PARALLEL SHARED(K)
CALL SUB3(K)
!$OMP  END PARALLEL
END SUBROUTINE SUB2

SUBROUTINE SUB1(N)
INTEGER N
INTEGER I
!$OMP PARALLEL PRIVATE(I)
!$OMP DO
DO I = 1, N
CALL SUB2(I)
END DO
!$OMP END PARALLEL
END SUBROUTINE SUB1

PROGRAM A15
CALL SUB1(2)
CALL SUB2(2)
CALL SUB3(2)
END PROGRAM A15
```

Лекция №6. Управление окружающей средой данных OpenMP

Для управления переменными в параллельных областях в OpenMP используется одна директива threadprivate и несколько клауз разделения данных.

1. Директива threadprivate

Директива threadprivate применяется к переменным описанные во внешнем блоке (COMMON-блокам), которые необходимо сделать приватными. Директива должна применяться после каждой декларации внешнего блока. Синтаксис директивы следующее:

В языке Fortran	!$OMP THREADPRIVATE(*список*)
В языке C/C++	#pragma omp threadprivate(*список*)

Здесь *список* - разделенные запятыми переменные, описанные во внешнем блоке.

Внешние блоки, перечисленные в списке параметров директивы threadprivate, тиражируются для каждой нити. Нить не может сослаться на копию другой нити, т.е. переменные внешних блоков, оставаясь глобальными в рамках одной нити, будут невидимы из других нитей.

Директива threadprivate должна размещаться в области последовательного блока, вне блоков действия директив параллелизации. Так как параметрами этой директивы являются имена внешних блоков, она должна находиться в области их видимости.

Пример. При входе в первую параллельную область переменная внешнего блока k копируется по нитям и принимает значение номера нити. Во второй параллельной области значения внешнего блока для каждой нити сохраняется.

В языке Fortran	*В языке C/C++*

integer, save :: k
!$OMP THREADPRIVATE(k)
…
!$OMP PARALLEL
k=OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
…
!$OMP PARALLEL
…
!$OMP END PARALLEL
	int k;
	#pragma omp threadprivate(k)
	…
	#pragma omp parallel
	{
	k=omp_get_thread_num();
	}
	…
	#pragma omp parallel
	{…}

2. Клауза default

Клауза default позволяет пользователю управлять типами переменных в директиве parallel по умолчанию, т.е. определять их неявно.

Синтаксис клаузы default следующее:

Fortran:	DEFAULT(PRIVATE | SHARED | NONE)
C/C++:	default (shared | none)

Все переменные, которые не описаны явно в списках переменных клауз **private** и shared будут относиться к списку переменных клаузы default,

В случаях использования:

- default (shared) – все переменные, которые не определены в параллельной области становятся общими – shared;
- default (none) – все переменные параллельной области должны быть определены в private и shared или они будут недоступными;
- default (private) – все переменные, которые не определены в параллельной области становятся приватными – private.

Примечание. Клауза default *применяется с директивой* parallel.

Пример 1. В параллельной области переменная x является общей, а все остальные приватные по умолчанию.

Fortran:	!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(x)
C/C++:	#pragma omp parallel default (shared) shared(x)

Пример 2. В параллельной области определены переменные x, y и I, а все остальные являются невидимыми, их следует определить.

Fortran:	!$OMP PARALLEL DEFAULT(NONE) SHARED(x,y) PRIVATE(i)
C/C++:	#pragma omp parallel default (none) shared(x, y) private(i)

3. Клауза shared

Клауза shared применяется к переменным, которые нужно сделать глобальными (общими) среди всех нитей в команде.

Синтаксис клаузы shared следующее:

Fortran:	SHARED(список)
C/C++:	shared(список)

Другими словами, нити будут работать не с копиями, а с оригиналами переменных.

Все нити в пределах команды получают доступ к той же самой области хранения для каждого объекта shared.

Примечание. Клауза shared *применяется с директивой* parallel.

Пример. В параллельной области переменные x и y являются общими.

Fortran:	!$OMP PARALLEL SHARED(x, y)
C/C++:	#pragma omp parallel shared(x, y)

4. Клауза private

Клауза private применяется к переменным, которые нужно сделать локальными (приватными).

Синтаксис клаузы private следующее:

Fortran:	PRIVATE(список)
C/C++:	private (список)

При входе в параллельную область для каждой нити создаётся отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

Примечание. Клауза private *применяется с директивами* parallel, do/for, sections, single.

Пример. В параллельном цикле переменные i и j являются приватными.

Fortran:	!$OMP PARALLEL DO PRIVATE (i, j)
C/C++:	#pragma omp parallel for private (i, j)

5. Клауза firstprivate

Клауза firstprivate объявляет, что один или несколько приватных копии переменных при входе в параллельную область инициализируются значением оригинальной переменной.

Синтаксис клаузы firstprivate следующее:

Fortran:	FIRSTPRIVATE(список)
C/C++:	firstprivate (список)

Предполагается, что копии private-переменных должны быть созданы до начала блока параллелизации.

Примечание. Клауза firstprivate *применяется с директивами* parallel, for, sections, single.

Пример. При входе в параллельную область переменная x равна 5.

Fortran:	x=5 i=1 !$OMP PARALLEL PRIVATE (i) FIRSTPRIVATE(x)
C/C++:	x=5; i=1; #pragma omp parallel private (i) firstprivate(x)

6. Клауза lastprivate

Клауза lastprivate объявляет, что один или несколько приватных копии переменных по окончании параллельно цикла do/for или блока параллельных секций section, нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение оригинальной переменной.

Синтаксис клаузы firstprivate следующее:

Fortran:	LASTPRIVATE(список)

C/C++:	lastprivate (список)

Отличается от параметра private тем, что после выхода из цикла или из последней секции конечные значения перечисленных в списке переменных будут доступны для использования.

При совмещении lastprivate с модификатором завершения nowait переменные остающегося экземпляра будут находиться в состоянии неопределенности до первого барьера (директива barrier).

Примечание. Клауза lastprivate *применяется с директивами* for, sections.

Пример. При выходе из параллельного цикла переменная x принимает значение последней завершенной нити в группе.

Fortran	!$OMP DO PRIVATE (i) LASTPRIVATE(x) do i=1,1000 x=i end do !$OMP END DO

C/C++	#pragma omp for private (i) lastprivate(x)
	for(i=0; i<100: i++)
	{
	x=i;
	}

7. Клауза reduction

Клауза reduction определяет оператор и один или несколько переменных списка и используется для исполнения некоторой формы параллельного вычисления. Для вычисления применяются математические ассоциативные и коммутативные операторы. Для каждой оригинальной переменной создаются приватные копии на каждой нити. Значения оригиналов корректируются после выполнения каждой нити.

Синтаксис клаузы reduction следующее:

Fortran:	REDUCTION({оператор | имя_встроенной_процедуры} : список)
C/C++:	reduction(оператор : список)

Все нити переменных из списка будут обработаны указанной операцией, переменные остающей нити получают значения, равные результату обработки. Данная операция производится над копиями переменной во всех нитях. Операция обработки задаётся либо оператором, либо встроенной функций (в Fortran). При выходе из цикла результат присваивается оригинальной переменной.

Операторы для C/C++:

Оператор	Действие	Значение инициализации
+	Сложение	0
*	Умножение	1
-	Вычитание	0
&	Логическое И	~0 (каждый бит установлен)
|	Логическое ИЛИ	0
^	Побитовое исключающее ИЛИ	0
&&	Побитовое логическое И	1
||	Побитовое логическое ИЛИ	0

Операторы для Fortran:

Оператор	Действие	Значение инициализации
+	Сложение	0
*	Умножение	1
-	Вычитание	0
.and.	Логическое И	.true.
.or.	Логическое ИЛИ	.false.
.eqv.	Логическое эквивалентность	.true.
.neqv.	Логическое неэквивалентность	.false.

Имя встроенной процедуры для Fortran:

MAX	Выбор максимального элемента	Наибольшее отрицательное число в редукционном переменном типе
MIN	Выбор минимального элемента	Наибольшее положительное число в редукционном переменном типе
IAND	Побитовое логическое И	Все биты включены
IOR	Побитовое логическое ИЛИ	0
IEOR	Побитовое исключающее ИЛИ	0

Формы для определения переменных в параллельной области:

ФОРТРАН	C/C ++
x = x *оператор выражение* x = *выражение оператор* x (кроме	x = x *оператор выражение* x *бинор = выражение*

вычитания), где *оператор* - +, *,-, .and.. или.. eqv., .neqv., *выражение* - скалярное выражение, не включающее x. Формы для функций в reduction: = *имя_функций* (x, *список_выражений*) = *имя_функций* (*список_выражений*, x), где *имя_функций* - MAX, MIN, IAND, IOR, IEOR и *список_выражений* – разделенными запятыми список скалярных выражений, не включающих x.	x = *выражение оператор* x (кроме вычитания) x- - x, где *выражение* – скалярное выражение, не включающее x, *оператор* - не перегруженный оператор, но один из +, *,-, &, ^, |, &&, | |, *бинор* - не перегруженный оператор, но один из +, *, -, &, ^, |.

Примечание. Клауза reduction *применяется с директивами* parallel, for, sections.

Пример1. Параллельное скалярное произведение двух векторов a и b. При выходе из цикла все значения переменной s суммируются по всем процессам.

Fortran	S=0.0 !$OMP PARALLEL DO SHARED(a, b) REDUCTION(+:s) do i=1,N s=s+a(i)*b(i) end do

C/C++	s=0.0;
	#pragma omp parallel for shared(a, b) reduction(+:s)
	for(i=0; i<n: i++)
	{
	s += (a[i]*b[i]);
	}

Пример2. Параллельное произведение:

$$\prod_{i=2}^{n}\left(1-\frac{1}{i^2}\right)$$

При выходе из цикла все значения переменной p умножаются по всем процессам.

Fortran	program main double precision p integer i,n print*, 'Input n' read*, n p=1.0 !$OMP PARALLEL DO REDUCTION(*:p) do i=2,n p=p*(1-1/(i*i)) end do print*, 'p=', p stop end

C/C++	#include <iostream.h>
	int main()
	{
	double p;
	int i,n;
	cout << "Input n=";
	cin >> n;
	p=1.0;
	#pragma omp parallel for reduction(*:p)
	for (i=2;i<n;i++)
	p *=(1-1/(i*i));
	cout << "p=" << p << endl;
	return 0;
	}

Лекция №7. Другие клаузы OpenMP

Лекция посвящена изложению следующих клауз:

- клаузы копирования данных,
- клауза условия,
- разделения итерации по нитям.

1. Клауза копирования данных copyin

Для каждой нити параллельного региона создаются копии переменных, описанных в директиве threadprivate. При входе в

параллельную область приватные копии этих данных инициализируются оригинальными значениями. Имена, указанные в списках директивы threadprivate и параметра copyin, должны совпадать.

Синтаксис клаузы copyin следующее:

Fortran	COPYIN(список)
C/C++	copyin(список)

Клауза позволяет тиражировать не полные переменные из внешнего блока, а только нужные из них.

Примечание. Клауза copyin *применяется с директивой* parallel.

Пример.

Fortran	*C/C++*

!$OMP THREADPRIVATE(k)
!$OMP PARALLEL
k=OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
…
!$OMP PARALLEL
…
!$OMP END PARALLEL
 	
!$OMP PARALLEL COPYIN(k)
…
!$OMP END PARALLEL
	#pragma omp threadprivate(k)
	#pragma omp parallel
	{
	k=omp_get_thread_num();
	}
	…
	#pragma omp parallel
	{…}
	
	#pragma omp parallel copyin(k)
	{…}

2. Клауза копирования данных copyprivate

После выполнения нити, содержащей конструкцию single, новые значения переменных списка copyprivate будут доступны всем одноименным частным переменным (private и firstprivate), описанным в начале параллельного региона и используемым всеми его нитями.

Синтаксис клаузы copyprivate следующее:

Fortran	COPYPRIVATE(список)
C/C++	copyprivate(список)

Примечание. Клауза copyprivate *применяется с директивой* single.

Пример.

Fortran	SUBROUTINE Init_ab(a,b) real a, b COMMON /cd/ c,d !$OMP THREADPRIVATE(/cd/) !$OMP SINGLE read (1,*) a, b, c, d !$OMP END SINGLE COPYPRIVATE(a,b, /cd/) END SUBROUTINE Init_ab

C/C++	#include <omp.h>
	#include <stdio.h>
	float c,d
	#pragma omp threadprivate(c,d d)
	
	void Init_ab(a,b)
	{
	#pragma omp single copyprivate(a, b, c,d)
	{
	scanf("%f %f %f %f", & a, &b, &c, &d);
	}
	}

3. Клауза if

Если условие выполнено (true), то регион распараллеливается, в противном случае (false) все его блоки выполняются одним процессом (процессором) в естественном порядке.

Синтаксис клаузы if следующее:

Fortran	IF(скалярное_логическое_выражение)
C/C++	if(скалярное_выражение)

Результатом скалярного выражения является логическое значение в Fortran .TRUE. или .FALSE., а в C/C++ скалярное значение 0 или 1.

Скалярное выражение в Fortran строятся с помощью знаков отношения (.gt., .lt., .ge., .le., .eq., .ne. или >, <, >=,<=, ==, /=) и логических операции (.NOT., .AND., .OR., .EQV., .NEQV.).

Скалярное выражение в C/C++ строятся с помощью знаков отношения (>, <, >=,<=, ==, !=) и логических операции (!, &&, ||)

Примечание. Клауза if *применяется с директивой* parallel.

Пример. Если значение k при входе в параллельную область задано больше чем 100, то код в области выполняется параллельно

несколькими нитями. Если значение k меньше или равно 100, то код выполняется последовательно одной нитью.

Fortran	!$OMP PARALLEL IF(k>100) !$OMP DO do i=1,k … end do !$OMP END DO !$OMP END PARALLEL

C/C++	#pragma omp parallel if(k>100)
	{
	#pragma omp for
	for(i=1;i<k;i++)
	{
	…
	}
	}

4. Клауза schedule

По умолчанию в OpenMP для планирования параллельного выполнения циклов do/for используется алгоритм, называемый статическим планированием. Это означает, что все нити из группы выполняют равное число итераций цикла. Если n — число итераций цикла, а k — число нитей в группе, то каждая нить в среднем выполнит n/k итераций. Кроме того, OpenMP поддерживает и другие механизмы планирования, наилучше используемые в разных ситуациях: динамическое планирование, планирование в период выполнения и управляемое планирование.

Для того, чтобы устанавливать такие механизмы планирования в OpenMP применяется клауза schedule для директивы do/for. Клауза schedule определяет правила распределения итерации цикла по нитям.

Синтаксис клаузы schedule имеет следующий вид:

Fortran	SHEDULE(тип [, порция])
C/C++	schedule(тип [, порция])

Здесь *тип* – обязательный параметр, определяет алгоритм планирования работы в цикле, *порция* – число итераций в алгоритме планирования работы. Параметр *тип* может быть одним из следующих:

- static (статический);

- dynamic (динамический);
- guided (управляемый);
- runtime (во времени выполнения).

СТАТИЧЕСКИЙ: Итерации в цикле распределяются по нитям в равном количестве, установленной порцией, и продолжается в течение выполнения итерационного процесса.

По умолчанию, число работы равно числу нитей в команде, и все итерации приблизительно равны в размере. В этом случае, число порции определяется делением числа итерации в цикле do/for на число нитей. Суммарный проход всех этих итерации по нитям равно полному итеративному процессу в цикле do/for.

Если порция определена, то размер частей установлен к количеству нитей. В этом случае части работы могут иметь различный размер, чем *порция*.

Пример. Рассмотрим фрагмент параллельного цикла с тремя нитями со статическим планированием.

Fortran:	!$OMP DO SCHEDULE(STATIC, *chunk*) do i=1,600 …. end do !$OMP END DO

C/C++:	#pragma omp for schedule(static, *chunk*)
	for (i=1;i<=600;i++)
	{
	…
	}

Пусть при n единицы времени выполняется 50 итерации. Тогда при случаях:

1) chunk=150, четыре блока работы необходимы, чтобы охватить полный итеративный процесс. Так как общее количество повторений - точно четыре, то все они равны по размеру. Заканчивающаяся параллель, управляющая правилам не очень эффективна, так как работает только 0 нить, в то время как все другие нити в состоянии ожидании. Полное время равно 6n.

2) chunk=250, число блоков равняется нитям, является более-менее хорошим случаем. Но и здесь размеры блоков не равны между собой. Является лучшим чем в предыдущем случае, здесь время ожидания меньше. Полное время равно 5n.

3) chunk=300, число блоков меньше чем число доступных нитей. Это означает, что 2-я нить не учувствует. Заканчивающаяся эффективность хуже тогда в предыдущем случае и равна первому случаю. Полное время равно 6n.

4) chunk не задан, OpenMP создает множество равных по размеру блоков, и число блоков равны числу доступных нитей. Тогда создаются три блока , каждая выполняет 200 итерации. Вообще этот выбор ведет к лучшей работе, если все повторения требуют того же самого вычислительного времени. Полное время равно 4n.

<table>
<tr><th>Итерации</th><th>chunk=150</th><th>chunk=250</th><th>chunk=300</th><th>chunk=default</th></tr>
<tr><td>0…50</td><td rowspan="3">0 нить</td><td rowspan="4">0 нить</td><td rowspan="5">0 нить</td><td rowspan="3">0 нить</td></tr>
<tr><td>50…100</td></tr>
<tr><td>101…150</td></tr>
<tr><td>151…200</td><td rowspan="3">1 нить</td></tr>
<tr><td>201…250</td><td rowspan="3">1 нить</td></tr>
<tr><td>251…300</td><td rowspan="4">1 нить</td></tr>
<tr><td>301…350</td><td rowspan="3">2 нить</td><td rowspan="4">1 нить</td></tr>
<tr><td>351…400</td><td rowspan="3">2 нить</td></tr>
<tr><td>401…450</td></tr>
<tr><td>451…500</td><td rowspan="3">0 нить</td><td rowspan="3">2 нить</td></tr>
<tr><td>501…550</td></tr>
<tr><td>551…600</td></tr>
</table>

ДИНАМИЧЕСКИЙ: Итеративный процесс разделено на части работы с размером равные заданной порции. Каждая нить берет на выполнение первый еще не взятый блок итерации. Если число порции не задано, то по умолчанию он равен 1-ой итерации. После того как нить завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный.

Пример. Рассмотрим фрагмент параллельного цикла с 4 нитями с динамическим планированием.

Fortran:	!$OMP DO SCHEDULE(DYNAMIC, 15) do i=1,100 …. end do !$OMP END DO
C/C++:	#pragma omp for schedule(dynamic,15) for (i=1;i<=100;i++) { … }

Тогда возможен следующий вариант выполнения:

Нить	Действие	Итерации
0	получает право на выполнение	1-15
1	получает право на выполнение	16-30
2	получает право на выполнение	31-45
3	получает право на выполнение	46-60
2	завершает выполнение итераций	
2	получает право на выполнение	61-75
3	завершает выполнение итераций	
3	получает право на выполнение	76-90
0	завершает выполнение итераций	
0	получает право на выполнение	91-100

УПРАВЛЯЕМЫЙ: Размер блока итерации уменьшается экспоненциально до величины порции. Число итераций, выполняемых каждой нитью, определяется по следующей формуле:

n =max(m/k,chunk),

где n – число выполняемых нитью итераций, m – число нераспределенных итераций, k – число равное значению функций omp_get_num_threads(), chunk – размер порции.

Начальный размер порции зависит от реализации. Завершив выполнение назначенных итераций, поток запрашивает выполнение другого набора итераций, число которых определяется по только что приведенной формуле. Таким образом, число итераций, назначаемых каждому потоку, со временем уменьшается. Последний набор итераций может быть меньше, чем значение, вычисленное по формуле.

Если порция не задана, то она равна 1.

Динамическое и управляемое планирование хорошо подходят, если при каждой итерации выполняются разные объемы работы или если одни процессоры более производительны, чем другие.

При статическом планировании нет никакого способа, позволяющего сбалансировать нагрузку на разные потоки.

При динамическом и управляемом планировании нагрузка распределяется автоматически – такова сама суть этих подходов. Как правило, при управляемом планировании код выполняется быстрее, чем при динамическом, вследствие меньших издержек на планирование.

Любой из предыдущих трех методов планирования должен быть установлен во время компилирования исходного кода. Планирование в период выполнения – это способ динамического выбора одного из трех описанных алгоритмов.

ВО ВРЕМЯ ВЫПОЛНЕНИЯ: Планирование в период выполнения дает определенную гибкость в выборе типа планирования, при этом по умолчанию применяется статическое планирование.

Fortran:	!$OMP DO SCHEDULE(RUNTIME)
C/C++:	#pragma omp for schedule(runtime)

При указанном параметре runtime, исполняющая среда OpenMP использует алгоритм планирования, заданный для конкретного цикла do/for при помощи переменной окружения OMP_SCHEDULE. Она имеет формат «тип[,число итераций]», например:

set OMP_SCHEDULE=dynamic,8

Лекция № 8. Библиотека процедур среды выполнения

В OpenMP встроена библиотека процедур и функций обеспечивающие интерфейс программы с многопоточным окружением исполнения. Библиотека подразделяется на следующие категории:

- процедуры и функции для контроля/запроса параметров среды исполнения (библиотека runtime);
- процедуры и функции переменной среды;
- функции для синхронизации на базе замков.

Процедуры и функции для контроля/запроса параметров среды исполнения

Эти процедуры и функции позволяют запрашивать и задавать различные параметры операционной среды OpenMP. Они позволяют в процессе выполнения задания определить и изменить количество нитей, используемых в параллельном регионе, определить номер конкретной нити и количество доступных процессоров в системе.

Процедуры, имена которых начинаются на omp_set_, можно вызывать только вне параллельных регионов. Все остальные функции можно использовать внутри и вне параллельных регионов.

Для языков C/C++ процедуры runtime являются внешними функциями подключающие препроцессором omp.h, для FORTRAN они являются процедурами и вызываются оператором вызова вида:

call *имя_процедуры(параметры)*

К runtime процедурам относятся:

- omp_set_num_threads;
- omp_get_num_threads;
- omp_get_max_threads;
- omp_get_thread_num;
- omp_get_num_procs;
- omp_in_parallel;
- omp_set_dynamic;
- omp_get_ dynamic;
- omp_set_nested;
- omp_get_nested.

1. OMP_SET_NUM_THREADS

Процедура, устанавливает число нитей для ближайшей следующей группы исполнения равным значению указанного скалярного целого выражения. Процедура имеет действие только в области последовательного исполнения при условии, динамическое назначение числа нитей внутри программы разрешено. Имеет приоритет перед переменной окружения параллельного OMP_ NUM_THREADS. Внутри области параллелизации или при отсутствии разрешения менять число нитей ее действие игнорируется.

Формат записи:

Fortran	SUBROUTINE OMP_SET_NUM_THREADS(*число_нитей*) INTEGER *число_нитей*
C/C++	void omp_set_num_threads(int *число_нитей*);

Пример.

Fortran: CALL OMP_SET_NUM_THREADS(9)

C/C++: omp_set_num_threads(9);

2. OMP_GET_NUM_THREADS

Функция, возвращает целое значение, равное числу нитей текущей группы исполнения в параллельном регионе. При вызове из области последовательного исполнения возвращает 1.

Формат записи:

Fortran	INTEGER FUNCTION OMP_GET_NUM_THREADS()
C/C++	int omp_get_num_threads(void);

Пример.
Fortran: k=OMP_GET_NUM_THREADS()
C/C++: k=omp_get_num_threads();

3. OMP_GET_MAX_THREADS

Функция, возвращает целое значение, равное максимально возможному числу нитей.

Формат записи:

Fortran	INTEGER FUNCTION OMP_GET_MAX_THREADS()
C/C++	int omp_get_max_threads(void);

Пример.
Fortran: n=OMP_GET_MAX_THREADS()
C/C++: n=omp_get_max_threads();

4. OMP_GET_THREAD_NUM

Функция, возвращает целое значение, равное порядковому номеру нити, которая сделала вызов.

Формат записи:

Fortran	INTEGER FUNCTION OMP_GET_THREAD_NUM()
C/C++	int omp_get_thread_num(void);

Пример.
Fortran: p=OMP_GET_THREAD_NUM()
C/C++: p=omp_get_thread_num();

5. OMP_GET_NUM_PROCS

Функция, возвращает целое значение, равное числу нитей доступных программе.

Формат записи:

Fortran	INTEGER FUNCTION OMP_GET_NUM_PROCS()
C/C++	int omp_get_num_procs(void);

Пример.
Fortran: m=OMP_GET_NUM_PROCS()
C/C++: m=omp_get_num_procs();

6. OMP_IN_PARALLEL

Функция, возвращает логическое значение .TRUE. в *Fortran* или целое значение не равное 0 в *C/C++*, если вызов ее был сделан из параллельной области, и .FALSE. в *Fortran* или 0 в *C/C++*, если вызов был сделан из последовательной области.

Формат записи:

Fortran	LOGICAL FUNCTION OMP_IN_PARALLEL()
C/C++	int omp_in_parallel(void);

Пример.
Fortran: q=OMP_IN_PARALLEL()
C/C++: q=omp_ in_parallel();

7. OMP_SET_DYNAMIC

Процедура, управляющая разрешением динамического изменения числа нитей в группе исполнения ближайшей следующей параллельной области. Имеет преимущество перед переменной окружения OMP_DYNAMIC. Устанавливается равным значению указанного скалярного логического выражения. Процедура имеет действие только в последовательного исполнения, в параллельной области игнорируется.

Формат записи:

Fortran	SUBROUTINE OMP_SET_DYNAMIC(скал_лог_выражение) LOGICAL скал_лог_выражение
C/C++	void omp_set_dynamic(int скалярное_выражение);

Пример.
Fortran: CALL OMP_ SET_DYNAMIC(10)
C/C++: omp_set_dynamic(10);

8. OMP_GET_DYNAMIC

Функция, возвращает логическое значение, сообщающее о разрешении/запрещении динамического изменения числа нитей в группе исполнения ближайшей области параллелизации.

Формат записи:

Fortran	LOGICAL FUNCTION OMP_GET_DYNAMIC()
C/C++	int omp_get_dynamic();

Пример.
Fortran: p=OMP_GET_DYNAMIC()
C/C++: p=omp_get_dynamic();

9. OMP_SET_NESTED

Функция, управляющая разрешением вложенного параллелизма. Вложенный параллелизм разрешается, если значение указанного в параметре вызова равно истине. Имеет преимущество перед переменной окружения OMP_NESTED.

Формат записи:

Fortran	SUBROUTINE OMP_SET_NESTED(скал_лог_выражение) LOGICAL скал_лог_выражение
C/C++	void omp_set_nested(int скалярное_выражение);

Примеры вызова.
Fortran: CALL OMP_ SET_NESTED(q)
C/C++: omp_set_nested(q);

10. OMP_GET_NESTED

Функция, возвращает логическое значение, сообщающее о разрешении/запрещении вложенного параллелизма.

Формат записи:

Fortran	LOGICAL FUNCTION OMP_GET_NESTED()
C/C++	int omp_get_nested(void);

Примеры вызова.
Fortran: q=OMP_GET_NESTED(q)
C/C++: q=omp_gt_nested();

Пример программы вложенного параллелизма

```
program main
real x(100),y(100)

! разрешить вложенный параллелизм
   call OMP_SET_NESTED(.true.)

! стартовый параллельный регион.
   !$OMP PARALLEL
```



```
! стартовые параллельные секции для x и y.
   !$OMP SECTIONS
! Секция A. Выполняет работу для x. Начинать с новой группы, чтобы
выполнить эту работу.
   !$OMP SECTION
   !$OMP PARALLEL
   !$OMP DO
   do i = 1, 100
   call do_work_on_x(x,i,10 0)
   enddo
   !$OMP END PARALLEL
! Секция B. Выполняет работу для y. Начинать с новой группы, чтобы
выполнить эту работу
   !$OMP SECTION
   !$OMP PARALLEL
   !$OMP DO
   do i = 1, 100
   call do_work_on_y(y,i,10 0)
   enddo
   !$OMP END PARALLEL
   !$OMP END SECTIONS

! включать как x так и y
   !$OMP DO
   do i = 1, 100
   x(i) = x(i)*y(i)
   enddo

   !$OMP END PARALLEL
   end
```

Пример программы с четырьмя отдельными параллельными областями и двумя вложенными.

```
#include <stdio.h>
#include <omp.h>
int main()
{
  omp_set_dynamic(1);
  omp_set_num_threads(10);
```

```
    #pragma omp parallel       // параллельный регион 1
    {
     #pragma omp single
    printf("Num threads in dynamic region is = %d\n",
 omp_get_num_threads());
    }
   printf("\n");
   omp_set_dynamic(0);
   omp_set_num_threads(10);
   #pragma omp parallel       // параллельный регион 2
   {
     #pragma omp single
     printf("Num threads in non-dynamic region is = %d\n",
 omp_get_num_threads());
   }
   printf("\n");
   omp_set_dynamic(1);
   omp_set_num_threads(10);
   #pragma omp parallel       // параллельный регион 3
   {
     #pragma omp parallel
     {
       #pragma omp single
     printf("Num threads in nesting disabled region is = %d\n",
omp_get_num_threads());
     }
   }
   printf("\n");
   omp_set_nested(1);
   #pragma omp parallel       // параллельный регион 4
   {
     #pragma omp parallel
     {
       #pragma omp single
       printf("Num threads in nested region is = %d\n",
omp_get_num_threads());
     }
   }
}
```



```
program main
integer k
call omp_set_dynamic(1);
call omp_set_num_threads(10);

! параллельный регион 1
!$OMP parallel
!$OMP single
k=omp_get_num_threads()
write(*,*) 'Num threads in dynamic region is =',k
!$OMP END single
!$OMP END parallel
write(*,*)
call omp_set_dynamic(0)
call omp_set_num_threads(10)

! параллельный регион 2
!$OMP parallel
!$OMP single
k=omp_get_num_threads()
write(*,*) 'Num threads in non-dynamic region is = ', k
!$OMP END single
!$OMP END parallel
write(*,*)

call omp_set_dynamic(1)
call omp_set_num_threads(10)
! параллельный регион 3
!$OMP parallel
!$OMP parallel
!$OMP single
k=omp_get_num_threads()
write(*,*) 'Num threads in nesting disabled region is = ', k
!$OMP END single
!$OMP END parallel
!$OMP END parallel
write(*,*)

call omp_set_nested(1)
! параллельный регион 4
!$OMP parallel
```

```
!$OMP parallel
!$OMP single
k=omp_get_num_threads()
write(*,*) 'Num threads in nested region is = ', k
!$OMP END parallel
!$OMP END parallel
stop
end
```

Результат выполнения:

```
Num threads in dynamic region is = 2
Num threads in non-dynamic region is = 10
Num threads in nesting disabled region is = 1
Num threads in nesting disabled region is = 1
Num threads in nested region is = 2
Num threads in nested region is = 2
```

Для 1-го региона мы включили динамическое создание потоков и установили число потоков в 10. По результатам работы программы видно, что при включенном динамическом создании потоков исполняющая среда OpenMP решила создать группу, включающую всего два потока, так как у компьютера два процессора. Для 2-го параллельного региона исполняющая среда OpenMP создала группу из 10 потоков, потому что динамическое создание потоков для этого региона было отключено.

Результаты выполнения 3-го и 4-го параллельных регионов иллюстрируют следствия включения и отключения возможности вложения регионов. В 3-ем параллельном регионе вложение было отключено, поэтому для вложенного параллельного региона не было создано никаких новых потоков — и внешний, и вложенный параллельные регионы выполнялись двумя потоками. В 4-м параллельном регионе, где вложение было включено, для вложенного параллельного региона была создана группа из двух потоков (т. е. в общей сложности этот регион выполнялся четырьмя потоками). Процесс удвоения числа потоков для каждого вложенного параллельного региона может продолжаться, пока вы не исчерпаете пространство в стеке. На практике можно создать несколько сотен потоков, хотя связанные с этим издержки легко перевесят любые преимущества.

Как вы, вероятно, заметили, для 3-го и 4-го параллельных регионов динамическое создание потоков было включено. Посмотрим, что

будет, если выполнить тот же код, отключив динамическое создание потоков:

```
omp_set_dynamic(0);
omp_set_nested(1);
omp_set_num_threads(10);
#pragma omp parallel
{
  #pragma omp parallel
  {
    #pragma omp single
    printf("Num threads in nested region is = %d\n",
      omp_get_num_threads());
  }
}
```

А происходит то, чего и следовало ожидать. Для 1-го параллельного региона создается группа из 10 потоков, затем при входе во вложенный параллельный регион для каждого из этих 10 потоков создается группа также из 10 потоков. В общей сложности вложенный параллельный регион выполняют 100 потоков:

```
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
```

Лекция №9. Библиотека процедур и функций замков OpenMP

Библиотека OpenMP включает набор замков общего назначения, которые могут использоваться для синхронизации кода.

Понятие замка

Замки – это переменные целого типа, которые расширяют возможности управления нитей и используются только процедурами

управления замков. Сначала такая переменная должна быть *инициализирована замком. Любая нить может занять замок, другая* нить предпринявший занять этот замок должен ожидать его освобождения.

Замок OpenMP может быть в одном из следующих трех состояний:

- *неинициализированном;*
- *разблокированном (отпертый);*
- *заблокированном (запертый).*

Если замок находится в отпертом состоянии, нить может *установить* замок, который изменяет его состояние на запертый. Нить, которая устанавливает замок, как говорят, *имеет* замок. Нить, которая имеет замок, может *сбросить* этот замок, возвращая его в отпертое состояние. Нить не может установить или сбросить замок, который принадлежит другой нити.

В OpenMP замки делятся на два вида:

- *простые замки*
- *вкладываемые замки (nestable).*

Простые замки (omp_lock_t) не могут быть установлены более одного раза, даже той же нитью. Вкладываемые замки (omp_nest_lock_t) равносильны простым с тем исключением, что, когда нить пытается установить уже принадлежащую ему вкладываемый замок, он не блокируется. Кроме того, OpenMP ведет учет ссылок на вкладываемые замки и следит за тем, сколько раз они были установлены.

Итак, вкладываемый замок может быть захвачен повторно, а простой замок можно захватить только один раз.

OpenMP предоставляет подпрограммы, выполняющие операции над этими замки. Каждая такая подпрограмма имеет два варианта: для простых и для вкладываемых замков.

Можно выполнить над замком пять следующих действий:

- *инициализировать замок;*
- *установить или захватить замок;*
- *освободить замок;*
- *проверить замок;*
- *уничтожить замок.*

Эти операции похожи на Win32-функции для работы с критическими секциями.

Тип простого замка

В C/C++ **omp_lock_t** – тип объекта, способный к представлению простого замка. У всех простых подпрограмм для работы с замками, переменная замка должна иметь тип **omp_nest_lock_t**. Все простые замки требуют аргумента, который является указателем на переменную типа **omp_lock_t.**

В Fortran для простых замков переменная *svar* должен быть целого числа **kind=omp_lock_kind.**

Тип вкладываемого замка

Для C/C++ **omp_nest_lock_t** – тип объекта, способный к представлению вкладываемого замка nestable. У всех вкладываемых подпрограмм для работы с замками, переменная замка должна иметь тип **omp_nest_lock_t**. Все вкладываемые замки требуют аргумента, который является указателем на переменную типа **omp_nest_lock_t.**

В Fortran для вкладываемых замков переменная *nvar* должен быть целого числа **kind=omp_nest_lock_kind.**

В следующей таблице показаны подпрограммы для работы с замками в OpenMP.

Простой замок	**Вкладываемый замок**
omp_init_lock	omp_init_nest_lock
omp_destroy_lock	omp_destroy_nest_lock
omp_set_lock	omp_set_nest_lock
omp_unset_lock	omp_unset_nest_lock
omp_test_lock	omp_test_nest_lock

Для синхронизации кода можно использовать и подпрограммы исполняющей среды, и директивы синхронизации.

Директивы синхронизации хорошо структурированы, понятны и легко определяются точки входа в синхронизированные регионы и выхода из них.

Преимущество подпрограмм исполняющей среды – это гибкость их использования. Можно передать блокировку в другую функцию и установить/освободить ее в этой функции. При использовании директив это невозможно. Если нет необходимости в гибкости синхронизации, поддерживаемые подпрограммами исполняющей среды, лучше использовать директивы синхронизации.

Подпрограммы для синхронизации на базе замков

1. omp_init_lock и omp_init_nest_lock

Подпрограммы предназначены для инициализации переменного замка простого и вкладываемого соответственно.

Синтаксис имеет вид:

Fortran	SUBROUTINE OMP_INIT_LOCK(svar) INTEGER(kind=omp_lock_kind) svar SUBROUTINE OMP_INIT_NEST_LOCK(nvar) INTEGER(kind=omp_nest_lock_kind) nvar
C/C++	void omp_init_lock (omp_lock_t *lock); void omp_init_nest_lock(omp_nest_lock_t *lock);

Пример:

Fortran	CALL OMP_INIT_LOCK(svar) CALL OMP_INIT_NEST_LOCK(nvar)
C/C++	omp_init_lock (*lock*); omp_init_nest_lock(*lock*);

2.omp_destroy_lock и omp_destroy_nest_lock

Подпрограммы предназначены для уничтожения замков простого и вкладываемого соответственно. Переменная-параметр подпрограмм должна быть инициализированным замком.

Синтаксис имеет вид:

Fortran	SUBROUTINE OMP_DESTROY_LOCK(svar) INTEGER(kind=omp_lock_kind) svar SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar) INTEGER(kind=omp_nest_lock_kind) nvar
C/C++	void omp_destroy_lock (omp_lock_t *lock); void omp_destroy_nest_lock(omp_nest_lock_t *lock);

Пример:

Fortran	CALL OMP_DESTROY _LOCK(svar) CALL OMP_DESTROY_NEST_LOCK(nvar)
C/C++	omp_destroy_lock (*lock*); omp_destroy_nest_lock(*lock*);

3.omp_set_lock и omp_set_nest_lock

Подпрограммы предназначены для установки или захвата замка OpenMP простого и вкладываемого соответственно. Нить, исполняющий подпрограмму, должен ждать, пока замок освободится, и после этого захватить её. Для простых замков захват даётся исполняющему процедуру нити, если замок свободен. Для вкладываемых замков право захвата замка нить получает, если замок свободен или он уже принадлежит этой нити. Переменная-параметр подпрограммы должна быть инициализированным замком.

Синтаксис имеет вид:

Fortran	SUBROUTINE OMP_SET_LOCK(svar) INTEGER(kind=omp_lock_kind) svar SUBROUTINE OMP_SET_NEST_LOCK(nvar) INTEGER(kind=omp_nest_lock_kind) nvar
C/C++	void omp_set_lock (omp_lock_t *lock); void omp_set_nest_lock (omp_nest_lock_t *lock);

Пример:

Fortran	CALL OMP_SET _LOCK(svar) CALL OMP_SET_NEST_LOCK(nvar)
C/C++	omp_set_lock (*lock*); omp_set_nest_lock(*lock*);

4.omp_unset_lock и omp_unset_nest_lock

Подпрограммы предназначены для освобождения замка простого и вкладываемого соответственно. Переменная-параметр подпрограммы должна быть инициализированным замком, принадлежащим нити.

Синтаксис имеет вид:

Fortran	SUBROUTINE OMP_UNSET_LOCK(svar) INTEGER(kind=omp_lock_kind) svar SUBROUTINE OMP_UNSET_NEST_LOCK(nvar) INTEGER(kind=omp_nest_lock_kind) nvar
C/C++	void omp_unset_lock (omp_lock_t *lock); void omp_unset_nest_lock(omp_nest_lock_t *lock);

Пример:

Fortran	CALL OMP_UNSET _LOCK(svar) CALL OMP_UNSET_NEST_LOCK(nvar)
C/C++	omp_unset_lock (*lock*) ; omp_unset_nest_lock(*lock*) ;

5.omp_test_lock и omp_test_nest_lock

Подпрограммы предназначены для проверки замка простого и вкладываемого соответственно. Они тестируют, доступен ли исполняющему подпроцессу указанный параметром замок, и пытаются захватить его, но не блокируют выполнение нити, выполняя подпрограмму. Для простого замка omp_test_lock возвращает ложь (.false. в Fortran, 0 в C/C++), если замок занят, иначе, она захватывает замок и возвращает значение истина (.true. в Fortran, 1 в C/C++).

Для вкладываемого замка, omp_test_nest_lock rountine возвращает новый счет вложения, если замок успешно установлен, иначе, возвращает ноль.

Синтаксис имеет вид:

Fortran	LOGICAL FUNCTION OMP_TEST_LOCK(svar) INTEGER(kind=omp_lock_kind) svar INTEGER FUNCTION OMP_TEST_NEST_LOCK(nvar) (kind=omp_nest_lock_kind) nvar
C/C++	int omp_test_lock (omp_lock_t *lock) ; int omp_test_nest_lock (omp_nest_lock_t *lock) ;

Пример:

Fortran	q=OMP_TEST_LOCK(svar) k=OMP_TEST_NEST_LOCK(nvar)
C/C++	q=omp_unset_lock (*lock*) ; k=omp_unset_nest_lock(*lock*) ;

Пример программы

```
program main
use omp_lib
implicit none
integer(kind=omp_lock_kind) :: lock
integer(kind=omp_integer_kind) :: id
call OMP_INIT_LOCK(lock)
!$OMP PARALLEL SHARED(lock) PRIVATE(id)
  id=OMP_GET_THREAD_NUM()
```



```
  call OMP_SET_LOCK(lock)
  write(*,*) "My thread is ", id
  call OMP_UNSET_LOCK(lock)
!$OMP END PARALLEL
call OMP_DESTROY_LOCK(lock)
end program main
```

```
//Пример программы на C++ с функциями для простого замка
#include <iostream.h>
#include <omp.h>

void skip(int i) {}
void work(int i) {}

int main()
{

    // Объявление переменных lock – замок, id - номер нити
    omp_lock_t lock;
    int id;

    // инициализация простого замка
    omp_init_lock(&lock);

    #pragma omp parallel shared(lock) private(id)
    {
            // определение порядкового номера нити
            id=omp_get_thread_num();

            // установка простого замка
            omp_set_lock(&lock);

            // вывод текущего номера нити
            cout<<"My thread id is"<< id << endl;

            // освобождение простого замка
            omp_unset_lock(&lock);

            // проверка простого замка на незанятость замка
            while(!omp_test_lock(&lock))
            {
```

```
                skip(id);
            }
            work(id);
            // освобождение простого замка
            omp_unset_lock(&lock);
        }
        // уничтожение простого замка
        omp_destroy_lock(&lock);
        return 0;
}
```

```
! Пример программы на Fortran с подпрограммами для простого замка
program main
use omp_lib

! Объявление переменных lock – замок, id - номер нити
integer(omp_lock_kind) lock
integer id

! инициализация простого замка
call OMP_INIT_LOCK(lock)

!$OMP PARALLEL SHARED(lock) PRIVATE(id)

! определение порядкового номера нити
id=OMP_GET_THREAD_NUM()

! установка простого замка
call OMP_SET_LOCK(lock)

! вывод текущего номера нити
print*, 'My thread id is',id

! освобождение простого замка
call OMP_UNSET_LOCK(lock)

! проверка простого замка на незанятость замка
do while(.not.OMP_TEST_LOCK(lock))
call skip(id)
end do
```



```
call work(id)

! освобождение простого замка
call OMP_UNSET__LOCK(lock)

!$OMP EMD PARALLEL

! уничтожение простого замка
call OMP_DESTROY_LOCK(lock)

end program

subroutine skip(id)
end subroutine skip

subroutine work(id)
end subroutine work
```

Лекция №10. Библиотека функций времени и переменные среды OpenMP

Функции для определения времени выполнения среды

В OpenMP имеются две функции, предназначенные для определения текущего времени выполнения программы: omp_get_wtime и omp_get_wtick.

1. Функция omp_get_wtime определяет текущее значение времени в секундах. Формат функции имеет следующий вид:

Fortran	DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
C/C++	double omp_get_wtime(void);

Пример использования функции

На языке Fortran	*На языке C/C++*

a=OMP_GET_WTIME()
…
b= OMP_GET_WTIME()
time=b-a
	a= omp_get_wtime();
	…
	b= omp_get_wtime();
	time=b-a;

2. Функция omp_get_wtick определяет значение времени в секундах между двумя последовательными функциями omp_get_wtime. Формат функции имеет следующий вид:

Fortran	DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
C/C++	double omp_get_wtick(void);

Переменные окружения для среды параллельного выполнения

Стандарт OpenMP, компиляторы Intel Fortran и Intel C++ предполагают наличие четырех переменных окружения. Эти переменные по умолчанию содержат значения, которые могут быть установлены при разработке параллельной области, и использоваться как механизмы управления выполнения кода. Имена переменных должны быть набраны прописными буквами, а аргументы могут быть набраны прописными или строчными буквами. Изменение значений переменных при выполнении задания невозможно. Способ установки значений переменных зависит от конкретной реализации компилятора, например, для Linux/Unix оболочки C++ применяется способ:

setenv OMP_SCHEDULE "dynamic"
setenv OMP_NUM_THREAD 8
setenv OMP_DYNAMIC 1
setenv OMP_NESTED 1

1. OMP_NUM_THREAD

Эта переменная среды определяет количество нитей, которые будут использоваться в течении выполнения параллельного региона в программе OpenMP-parallel.

При установлении количества нитей больше чем число доступных физических процессоров, тогда параллельный прогон программы в общей сложности может оказаться медленным.

Если динамическая установка количества нитей разрешена, то величина в omp_num_threads по умолчанию принимает максимальное число доступных нитей.

Пример

Fortran	export OMP_NUM_THREAD=8
C/C++	set OMP_NUM_THREAD=8

2. OMP_SCHEDULE

Эта переменная определяет способ распределения итерации в цикле do/for, если в директиве DO использована клауза schedule(runtime). Иначе говоря, если параллельный цикл планируется во время выполнения цикла в следующем виде.

В Fortran:
!$OMP PARALLEL DO SHEDULE(RUNTIME)
или
!$OMP DO SHEDULE(RUNTIME)

В C/C++:
#pragma omp parallel for schedule(runtime)
или
#pragma omp for schedule(runtime)

Тогда с помощью переменной omp_shedule можно установить распределение итерации по нитям в следующем виде.

Linux/Unix: setenv OMP_SCHEDULE "тип [,порция]"

В Windows NT/2000/XP: OMP_SCHEDULE "тип [,порция]"

Здесь тип – STATIC (статически блоками по порции), DYNAMIC (динамически блоками по порции), GUIDED (размер блока итерации уменьшается до размера порции), порция – заданное целое число итерации в блоке. По умолчанию переменная принимает тип STATIC.

Пример

Fortran	export OMP_SCHEDULE ="STATIC,10"
C/C++	set OMP_SCHEDULE ="STATIC,10"

3. OMP_DYNAMIC

Эта переменная управляет динамическим регулированием числа нитей в параллельном регионе. Переменная может принимать логическое значение истина или ложь. Если переменная установлена на значение истина, то для оптимизированного использования ресурсов системы разрешается динамическое изменение числа нитей в параллельном регионе. Если переменная установлена на значение ложь, то динамическое изменение числа нитей запрещено.

Если OMP_DYNAMIC не установлена, то по умолчанию принимает значение ложь.

Пример:

Fortran	export OMP_DYNAMIC=TRUE
C/C++	set OMP_DYNAMIC=1

4. OMP_NESTED

Эта переменная управляет вложенным параллелизмом. Переменная может принимать логическое значение истина или ложь. Если переменная установлена на значение истина, то вложенный параллелизм разрешается. Если переменная установлена на значение ложь, то вложенный параллелизм запрещается.

Если OMP_DYNAMIC не установлена, то по умолчанию принимает значение ложь.

Пример.

Fortran	export OMP_NESTED=TRUE
C/C++	set OMP_NESTED=1

Лекция №11. Моделирование и анализ параллельных вычислений

При разработке параллельных алгоритмов решения задач вычислительной математики важным моментом является анализ эффективности использования параллелизма. Анализ включает в себя минимизации времени решения задачи, а в параллельных вычислениях оно носит название как *ускорение* процесса вычисления. В вычислительной математике время решения задачи зависит от выбранного метода решения и от сложности алгоритма. То же самое, для определения оценки ускорения в параллельных вычислениях необходимо в первую очередь оценить выбранный вычислительный алгоритм, т.е. определить эффективность распараллеливания алгоритма, и во вторую очередь оценить выбранный метод для получения решения задачи, т.е. оценить эффективность параллельного способа для решения этой задачи.

Показатели эффективности параллельного алгоритма

Введем обозначение: T_1 - время решения задачи с одним процессом (нулевой нитью), T_n - время решения той же задачи с n процессами (n нитями). Время решения задачи зависит от вычислительной сложности задачи, числа входных данных задачи.

Понятие 1. Ускорение параллельного алгоритма есть величина R_n, определяемая как отношение времени T_1 к времени T_n:

$$R_n = \frac{T_1}{T_n}.$$

Понятие 2. Эффективность параллельного алгоритма есть величина $E_n(k)$, определяемая как отношение ускорения параллельного алгоритма к числу реально используемым процессам:

$$E_n = \frac{T_1}{n \cdot T_n} = \frac{R_n}{n}.$$

Закон Амдала

Введем обозначение: $W_{ск}$ - число скалярных операций в последовательной области алгоритма, $W_{пр}$ - число операции в параллельной области алгоритма, $W = W_{ск} + W_{пр}$ - общее число операции в алгоритме, t - среднее время выполнения одной операции, n - число процессов, $a = W_{ск}/W$ - удельный вес скалярных операций. Тогда получаем известный закон Амдала:

$$R_n = \frac{W \cdot t}{\left(W_{ск} + \frac{W_{пр}}{n}\right) \cdot t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n\to\infty]{} \frac{1}{a}.$$

Закон Амдала заключает важные для параллельных вычислений положения:

1. Ускорение зависит от потенциального параллелизма задачи (величина $1-a$) и числа процессов.

2. Предельное ускорение определяется свойствами задачи.

Например, пусть удельный вес скалярных операций $a = 0{,}2$ (реальная величина), тогда ускорение $R_n \le 5$ при любом числе используемых процессов. Получается, что $\max\limits_{n\to\infty} R_n$ определяется потенциальным параллелизмом задачи.

При идеальном параллелизме, т.е. при $W_{ск} = 0$, то ускорение $R_n = n$ и эффективность $E_n = 1$.

Сетевой закон Амдала

При обмене данными между процессорами и процессами могут быть потери времени, которые повлияют на ускорение параллельного вычисления и замедлить вычисление в целом.

Обозначим W_c - количество передач данных, t_c - среднее время одной передачи. Тогда получаем сетевой закон Амдала:

$$R_c = \frac{W \cdot t}{\left(W_{ск} + \frac{W_{пр}}{n}\right) \cdot t + W_c \cdot t_c} = \frac{1}{a + \frac{1-a}{n} + \frac{W_c \cdot t_c}{W \cdot t}} = \frac{1}{a + \frac{1-a}{n} + c}.$$

Этот закон определяет две особенности:

1. Коэффициент сетевой деградации вычислений, которая определяет общую затрату времени на передачу данных:

$$c = \frac{W_c \cdot t_c}{W \cdot t} = c_A \cdot c_T,$$

где

c_A - коэффициент алгоритма деградации (свойства алгоритма),

c_T - коэффициент технической деградации (быстродействие процессора и передачи данных в сети).

2. При идеальном параллелизме, т.е. при $W_{ск} = 0$ сетевое ускорение определяется величиной:

$$R_c = \frac{1}{\frac{1}{n} + c} = \frac{n}{1 + cn} \xrightarrow{c\to 0} n \cdot$$

Для определения эффективности параллельных вычислений можно использовать коэффициент утилизации:

$$z = \frac{R_c(k)}{n} = \frac{1}{1 + cn} \xrightarrow{c\to 0} 1 \cdot$$

Модель вычислений в виде графа "операции-операнды"

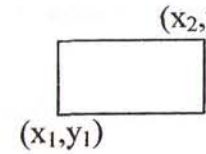
Модель "операции-операнды" позволяет демонстрировать имеющие информационные зависимости в выбранном алгоритме решения задач в виде графа. Предположим, что среднее время для любых вычислительных операции равняется 1, а также не тратится время на передачу данных.

Пусть множество операций в алгоритме решения вычислительной задачи и информационные зависимости между этими операциями представляется в виде ациклического ориентированного графа

$G = (V, D)$,

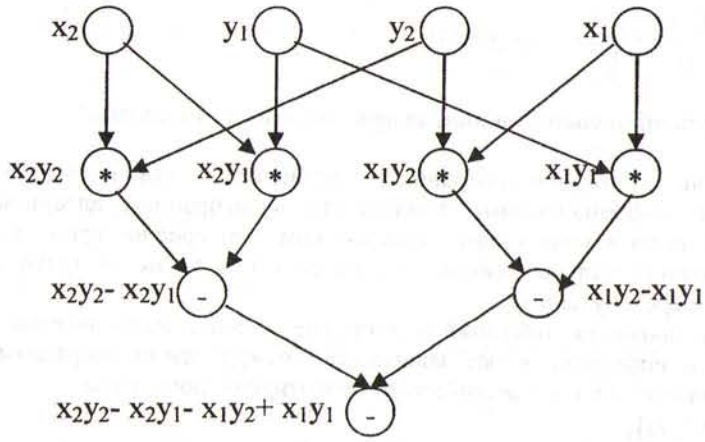
где V – множество пронумерованных вершин графа, представляющие выполняемые операции алгоритма, D – множество дуг графа. Дуга $d_{ij} \in G$, если операция j использует результат выполнения операции i.

Пример. Рассмотрим граф алгоритма вычисления площади прямоугольника, заданного координатами двух углов. Для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений и построены соответственно разные вычислительные модели. Разные схемы вычислений обладают разными возможностями для распараллеливания и, тем самым, при построении модели вычислений может быть поставлена задача выбора наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.



$S=((x_2-x_1)(y_2-y_1)= x_2y_2- x_2y_1- x_1y_2+ x_1y_1$

В этой вычислительной модели алгоритма вершины без входных дуг могут использоваться для операций ввода, а вершины без выходных дуг для операций вывода. Обозначим через $\overline{V}$ – множество вершин графа без вершин ввода, а через $d(G)$ диаметр (длину максимального пути) графа.



Описание схемы параллельного алгоритма

Операции, между которыми нет пути в схеме вычислений, могут быть выполнены параллельно. Для вычислительной схемы на рисунке выше, например, параллельно могут быть выполнены все операции умножения, а затем операции вычитания.

Введем множество расписаний для параллельного выполнения вычислений:

$$H_n = \{(i, n_i, t_i), i \in V\},$$

где для каждой операции $i \in V$ указывается номер используемого процесса n_i и время начала выполнения операции t_i. Для реализуемости расписания, необходимо выполнение следующих требований при задании множества H_n:

1. При времени $t_i = t_j$ для любых $i, j \in V$ должно быть $n_i \neq n_j$, т.е. один и тот же процесс не должен назначаться разным операциям в одно время.

2. Для любых $(i, j) \in D$ должно быть $t_j \geq t_i + 1$, т.е. к назначаемому моменту выполнения операции все необходимые данные должны быть вычислены.

Определение времени выполнения параллельного алгоритма

Время выполнения параллельного алгоритма $A_n(G, H_n)$ с n процессами определяется максимальным значением времени, используемым в расписании H_n:

$$T_n(G, H_n) = \max_{i \in V}(t_i + 1).$$

Для выбранной схемы G вычислений желательно использование расписания H_n, обеспечивающего минимальное время исполнения алгоритма

$$T_n(G) = \min_{H_n} T(G, H_n).$$

Уменьшение времени выполнения может быть обеспечено путем подбора наилучшей вычислительной схемы G:

$$T_n = \min_{G} T_n(G).$$

Оценки $T_p(G, H_p)$, $T_p(G)$ и T_p могут быть использованы в анализе времени выполнения параллельного алгоритма. Для анализа максимально возможной параллельности можно определить оценку наиболее быстрого исполнения алгоритма:

$$T_\infty = \min_{n \geq 1} T_n.$$

Очевидно, что время T_1 выполнения последовательного алгоритма:

$$T_1(G) = |\overline{V}|$$

Примерные темы лабораторных работ

Лабораторная работа №1. Векторы и матрицы.

Задание 1. Суммирование двух векторов.

1.1. Построить алгоритм, написать и отладить параллельную программу суммирования двух векторов с использованием распределения работ для параллельных процессов директивой **parallel**. При этом применить комбинацию параллельного цикла и редуцированной операции по всем процессам.

1.2. Построить алгоритм, написать и отладить параллельную программу суммирование двух векторов как в пункте 1.1. Но суммирование произвести в отдельной подпрограмме.

1.3. Построить алгоритм, написать и отладить параллельную программу суммирование двух векторов как в пункте 1.1 с использованием распределения работ для параллельных процессов директивой **sections**.

Задание 2. Умножение матрицы на вектор

2.1. Построить алгоритм, написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов директивой **parallel**.

2.2. Построить алгоритм, написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов директивой **sections**.

2.3. Построить алгоритм, написать и отладить параллельную программу умножения матрицы на вектор с использованием распределения работ для параллельных процессов с явным (вручную) заданием работы.

Задание 3. Нахождение максимального и минимального элемента матрицы.

3.1. Построить алгоритм, написать и отладить параллельную программу нахождения максимального элемента матрицы с использованием комбинации параллельного цикла и редуцированной операции по всем процессам.

3.2. Построить алгоритм, написать и отладить параллельную программу нахождения минимального элемента матрицы с использованием комбинации параллельного цикла и редуцированной операции по всем процессам.

Задание 4. Умножение матрицы на матрицу.

4.1. Построить алгоритм, написать и отладить параллельную программу умножения матрицы на матрицу с использованием директивы распараллеливания цикла по виткам.

4.2. Построить алгоритм, написать и отладить параллельную программу умножения матрицы на матрицу с использованием распределения работ для параллельных процессов с явным (вручную) заданием работы.

Задание 5. Нахождение определителя матрицы.

5.1. Построить алгоритм, написать и отладить параллельную программу нахождения определителя матрицы с использованием распределения работ для параллельных процессов директивой **sections**.

5.2. Построить алгоритм, написать и отладить параллельную программу нахождения определителя матрицы с использованием распределения работ для параллельных процессов с явным (вручную) заданием работы.

Задание 6. Нахождение обратной матрицы.

6.1. Построить алгоритм, написать и отладить параллельную программу нахождения обратной матрицы с использованием распределения работ для параллельных процессов директив **parallel** и **sections**.

6.2. Построить алгоритм, написать и отладить параллельную программу нахождения определителя матрицы с использованием распределения работ для параллельных процессов с явным (вручную) заданием работы.

Лабораторная работа №2. Функции и ряды.

Задание 1. Вычисление значения функции.

1.1. Построить алгоритм, написать и отладить параллельную программу вычисления значения функции с помощью ряда Тейлора, на интервале от a до b, шагом h , точностью eps.

1.2. Построить алгоритм, написать и отладить параллельную программу вычисления значения функции как в пункте 1.1. Но вычисление значения функции произвести в отдельной подпрограмме.

Задание 2. Вычисление значения суммы степенного ряда.

2.1. Построить алгоритм, написать и отладить параллельную программу вычисления значения суммы степенного ряда.

2.2. Построить алгоритм, написать и отладить параллельную программу вычисления значения суммы степенного ряда как в пункте 2.1. Но вычисление значения степенного ряда произвести в отдельной подпрограмме.

Задание 3. Вычисление значения произведения ряда.

3.1. Построить алгоритм, написать и отладить параллельную программу вычисления значения произведения ряда.

3.2. Построить алгоритм, написать и отладить параллельную программу вычисления значения произведения ряда как в пункте 2.1. Но вычисление значения произведения ряда произвести в отдельной подпрограмме.

Лабораторная работа №3. Решение систем линейных алгебраических уравнений (СЛАУ).

Задание 1. Решение СЛАУ методом Гаусса.

1.1. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений методом Гаусса с использованием распределения работ для параллельных процессов директивой **parallel**.

1.2. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений методом Гаусса. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 1.1.

Задание 2. Решение СЛАУ методом Зейделя.

2.1. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений методом Зейделя с использованием распределения работ для параллельных процессов директивой **parallel**.

2.2. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений методом Зейделя. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 2.1.

Задание3. Решение СЛАУ итерационными методами.

3.1. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений итерационными методами с использованием распределения работ для параллельных процессов директивой **parallel**.

3.2. Построить алгоритм, написать и отладить параллельную программу решения системы линейных алгебраических уравнений итерационными методами. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 3.1.

Лабораторная работа №4. Численное интегрирование с технологией OpenMP.

Задание 1. Численное интегрирование методом прямоугольника.

1.1. Построить алгоритм, написать и отладить параллельную программу решения численного интегрирования методом прямоугольника с использованием распределения работ для параллельных процессов директивой **parallel**.

1.2. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом прямоугольника в отдельной подпрограмме с отличающими параллельными секциями.

1.3. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом прямоугольника с автоматическим выбором шага.

Задание 2. Численное интегрирование методом трапеции.

2.1. Построить алгоритм, написать и отладить параллельную программу решения численного интегрирования методом трапеции с использованием распределения работ для параллельных процессов директивой **parallel**.

2.2. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом трапеции в отдельной подпрограмме с отличающими параллельными секциями.

2.3. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом трапеции с автоматическим выбором шага.

Задание 3. Численное интегрирование методом Симпсона.

3.1. Построить алгоритм, написать и отладить параллельную программу решения численного интегрирования методом Симпсона с использованием распределения работ для параллельных процессов директивой **parallel**.

3.2. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом Симпсона в отдельной подпрограмме с отличающими параллельными секциями.

3.3. Построить алгоритм, написать и отладить параллельную программу численного интегрирования методом Симпсона с автоматическим выбором шага.

Лабораторная работа №5. Интерполирование функций с технологией OpenMP.

Задание 1. Численное интерполирование методом Лагранжа.

1.1. Построить алгоритм, написать и отладить параллельную программу решения численного интерполирования методом Лагранжа с использованием распределения работ для параллельных процессов директивой **parallel**.

1.2. Построить алгоритм, написать и отладить параллельную программу численного интерполирования методом Лагранжа. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 1.1.

Задание 2. Численное интерполирование методом Ньютона.

2.1. Построить алгоритм, написать и отладить параллельную программу решения численного интерполирования методом Ньютона с использованием распределения работ для параллельных процессов директивой **parallel**.

2.2. Построить алгоритм, написать и отладить параллельную программу численного интерполирования методом Ньютона. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 2.1.

Задание 3. Численное интерполирование кубическим сплайном.

3.1. Построить алгоритм, написать и отладить параллельную программу решения численного интерполирования методом Ньютона

с использованием распределения работ для параллельных процессов директивой **parallel**.

3.2. Построить алгоритм, написать и отладить параллельную программу численного интерполирования методом Ньютона. При этом решение СЛАУ произвести в отдельной подпрограмме и соответствующая конструкция параллельной секции должна отличаться от пункта 3.1 .

Лабораторная работа №6. Численное решение дифференциальных уравнений в частных производных с технологией OpenMP.

Задание 1. Метод Якоби для решения задачи Пуассона.

1.1. Построить алгоритм, написать и отладить параллельную программу численного решения задачи Пуассона итерационным методом Якоби с использованием распределения работ для параллельных процессов директивой **parallel**.

1.2. Построить алгоритм, написать и отладить параллельную программу численного решения задачи Пуассона итерационным методом Якоби в отдельной подпрограмме с отличающими параллельными секциями.

Задание 2. Другие итерационные методы для решения задачи Пуассона.

2.1. Построить алгоритм, написать и отладить параллельную программу численного решения задачи Пуассона другими итерационными методами с использованием распределения работ для параллельных процессов директивой **parallel**.

2.2. Построить алгоритм, написать и отладить параллельную программу численного решения задачи Пуассона другими итерационными методами в отдельной подпрограмме с отличающими параллельными секциями.

Лабораторная работа №7. Сортировки и поиск с технологией OpenMP.

Задание 1. Методы сортировок.

1.1. Построить алгоритм, написать и отладить параллельную программу метода сортировки с использованием распределения работ для параллельных процессов директивой **parallel:**

а) Метод сортировки простыми включениями; б) метод сортировки с уменьшающимся расстоянием (метод Шелла); в) метод обменной сортировки; г) метод шейкерной сортировки; д) метод сортировки выбором; е) метод сортировки со слиянием.

1.2. Построить алгоритм, написать и отладить параллельные программы для методов а)-е) пункта 1.1 в отдельной подпрограмме с отличающими параллельными секциями.

Задание 2. Методы поиска.

2.1. Построить алгоритм, написать и отладить параллельную программу метода поиска с использованием распределения работ для параллельных процессов директивой **parallel:**

а) Последовательный поиск; б) Бинарный поиск; в) Интерполяционный поиск; г) Поиск в бинарном дереве поиска; д) Поиск подстрок; е) Алгоритм Хорспула; ж) Алгоритмы хеширования.

2.2. Построить алгоритм, написать и отладить параллельные программы для методов а)-ж) пункта 2.1 в отдельной подпрограмме с отличающими параллельными секциями.

Содержание

Учебное издание

Тунгатаров Нурмат Нургазиевич
Даирбаева Гульлазат Мансуровна

ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В СТАНДАРТЕ OPEN MP

Учебное пособие

ИБ № 4337

Подписано в печать 18. 11. 08. Формат 60x84 1/16. Бумага офсетная.
Печать RISO. Объем 5,125 п.л. Тираж 500 экз. Заказ № 556
Издательство «Қазақ университеті» Казахского национального университета им. аль-Фараби. 050038, г. Алматы, пр. аль-Фараби, 71, КазНУ.

Отпечатано в типографии издательства «Қазақ университеті».