Введение

OpenMP - это спецификация набора директив компилятору, библиотечных функций и переменных среды, которые могут быть использованы для организации многопотокового параллелизма ориентированного на машины с разделяемой памятью в программах на C, C++ и Fortran. OpenMP быстро становится стандартной парадигмой распараллеливания программ. Сравнительно небольшие усилия по написанию программы могут дать масштабируемую производительность выполнения приложения на многопроцессорной системе с разделяемой памятью.

В этом документе приведен обзор вычислительной модели OpenMP, а также описана поддержка OpenMP в компиляторах и инструментах Sun Studio.В дополнение статья сообщает об оценке производительности с помощью SPEC OMP2001 и обрисовывает направления будущей работы.

Содержание

Что такое OpenMP?

OpenMP - это программный интерфейс приложения (Application Programming Interface, API), который позволяет в явном виде определять многопотоковый параллелизм для машин с разделяемой памятью в программах на C, C++ и Fortran`e. Интерфейс OpenMP состоит из трех компонентов:

директив компилятору,
функций библиотеки времени выполнения и
переменных среды

В C/C++ директивы OpenMP определяются при помощи механизма #pragma. В Fortran`e определяются использованием специальных комментариев, которые идентифицируются как директивы OpenMP по уникальным комбинациям (в файлах исходных кодов стандартной формы распознаются сочетания !omp, comp и *$omp).

Спецификация OpenMP - определяющий документ-справочник по OpenMP (см. [1]). Последняя версия 2.5 является комбинированным справочником для C/C++ и Fortran`a. Спецификация принадлежит и управляется OpenMP Architecture Review Board (ARB) [2] - некоммерческой организацией, учрежденной в 1997 году.

Членство в ARB могут свободно получать корпорации, исследовательские организации и научные учреждения. Sun Microsystems является членом OpenMP ARB и играет важную роль в развитии данного программного интерфейса. Sun Microsystems активно участвует в еженедельных собраниях лингвистического комитета ARB, на которых обсуждается и модернизируется Спецификация. Также один из инженеров Sun Microsystems состоит в правлении ARB и является председателем.

Основными мотивами использования OpenMP являются производительность, масштабируемость, переносимость и стандартизация. С ростом технических требований приложений и объемом данных требуется все больше вычислительных мощностей. Большая производительность может быть достигнута использованием многих процессоров вместе для выполнения одного приложения. OpenMP обеспечивает широко поддерживаемый программный интерфейс для программирования машин с разделяемой памятью. Сравнительно небольшие усилия по написанию программы могут дать масштабируемое приложение для выполнения на многопроцессорной машине с разделяемой памятью.

Комплект программной разработки Sun Studio [3] - это обширный интегрированный набор компиляторов и инструментов для разработки и развертывания приложений на платформе Sun. Компиляторы Sun Studio поддерживают спецификацию OpenMP версии 2.5. Также инструменты Sun Studio поддерживают создание, отладку и анализ производительности OpenMP -приложений.

Вычислительная модель OpenMP

Основной машинной архитектурой для OpenMP является архитектура машин с разделяемой памятью, где все процессоры имеют доступ к одной общей памяти. Примерами машин с разделяемой памятью могут служить Sun Fire v890 с двуядерными процессорами UltraSPARC IV+ числом до 8, Sun Fire Enterprise E25K с числом процессоров до 72. Рисунок 1 показывает упрощенную схему машины с разделяемой памятью с n процессорами.

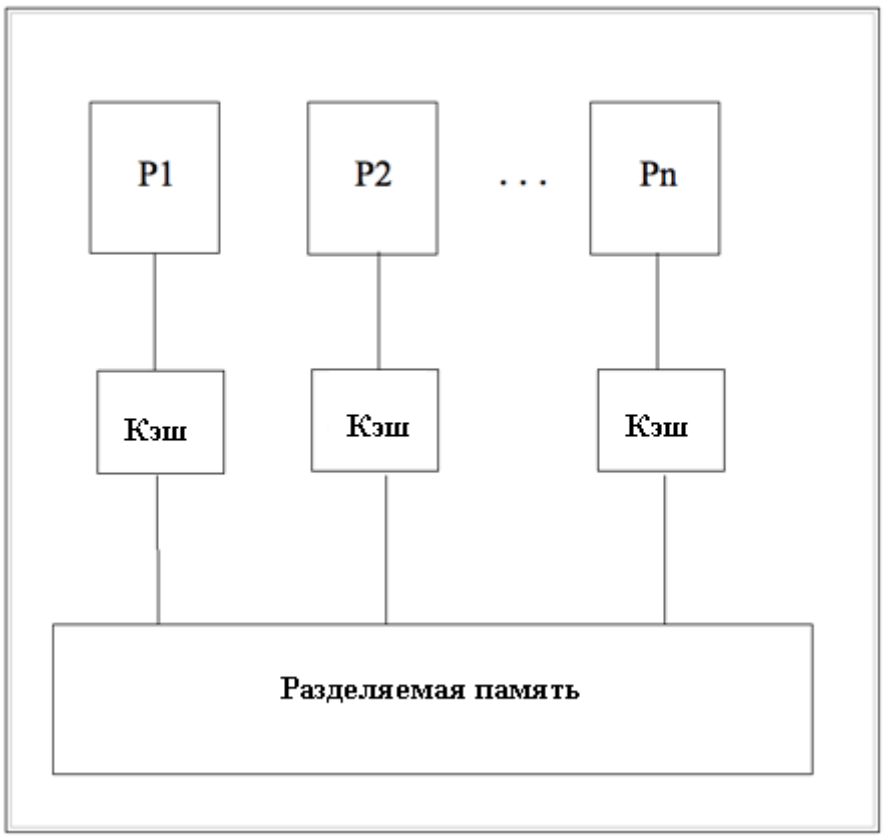

Рисунок 1: Система с разделяемой памятью

OpenMP использует модель разделения-слияния (fork-join) в параллельных вычислениях. Когда нить достигает параллельной конструкции она создает набор нитей, состоящий из себя и нескольких (возможно нуля) дополнительных нитей. Нить, которая достигла параллельной конструкции называется мастер-нитью (master-thread) набора. Остальные нити-подчиненные нити (slave-threads).

Все участники набора выполняют код в пределах параллельной конструкции. Когда нить заканчивает выполнение параллельной конструкции, она ждет у неявного барьера в конце конструкции. Когда все члены набора доберутся до барьера, мастер-нить в одиночестве продолжит выполнение кода пользователя после окончания параллельной конструкции. Любое количество параллельных конструкций может быть определено в программе.

Директивы OpenMP

В OpenMP содержится богатый набор директив, которые пользователь может использовать для организации параллелизма в программе. В этом разделе будут приведены примеры трех директив OpenMP, а именно: директивы PARALLEL, DO/for и SECTIONS. Более подробную информацию об этих и других директивах можно найти в [1] и [4].

Так как OpenMP основана на модели программирования с разделяемой памятью, переменные по умолчанию общие. Для явного определения области действия переменной могут быть использованы опции SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE и REDUCTION.

Число нитей, используемых для выполнения параллельной конструкции определяется пременной среды OMP_NUM_THREADS, которая может быть задана вызовом функции библиотеки времени выполнения omp_set_num_threads или использованием оператора NUM_THREADS.

Parallel

Директива PARALLEL определяет область кода, которая должна будет выполняться параллельно несколькими нитями. Все нити, участвующие в выполнении конструкции PARALLEL будут выполнять одинаковый участок кода. В результату участок кода будет растиражирован по нитям.

Когда нить достигает конструкции PARALLEL, она создает набор нитей и становится мастер-нитью. Мастер-нить имеет номер 0 в наборе. Другие нити пронумерованы числами 1, 2, ..., n-1, где n - общее количество нитей в наборе. По окончанию конструкции PARALLEL происходит неявная синхронизация нитей. Только мастер-нить продолжает выполняться после этой точки.

См. Приложение П.1 для примера использования конструкции PARALLEL.

DO/for

Директива DO/for - это директива разделения работы, применяемая к циклу DO (Fortran) или к циклу for (C/C++). Конструкция DO/for разделяет итерации цикла DO/for между нитями набора, достигающего этой конструкции. В конце конструкции происходит неявная синхронизация нитей, если не указана опция NOWAIT.

Гарантия того, что итерации цикла, выполняемого директивой DO/for не имеют зависимостей, должна обеспечиваться программистом. То есть результат одной итерации цикла не зависит от результата другой итерации цикла. Если это условие выполнено две разные итерации цикла могут быть выполнены параллельно двумя нитями.

См. Приложение П.2 для примера использования директивы DO/for.

Sections

Директива SECTIONS - это директива разделения работы, применяемая к отдельным блокам кода (каждый блок называется секцией, SECTION). Конструкция SECTIONS распределяет блоки кода по нитям из набора, достигшего конструкции. Каждый блок выполняется единожды одной нитью

набора. После окончания конструкции происходит неявная синхронизация нитей, если не указана опция NOWAIT.

Гарантия того, что отдельные блоки кода в конструкции SECTIONS независимы друг от друга и могут выполняться параллельно разными нитями, должна обеспечиваться самим программистом.

См. Приложение П.3 для примера использования директивы SECTIONS.

Поддержка OpenMP в программном обеспечении Sun Studio

Опция -xopenmp информирует компилятор Sun Studio о необходимости распознавать директивы OpenMP в программе.

Поддержка OpenMP в компиляторах состоит из двух частей. Во-первых, компилятор обрабатывает директивы OpenMP и преобразует код так, что он может выполняться несколькими нитями одновременно. Во-вторых, библиотека времени выполнения обеспечивает поддержку для управления нитями, синхронизации и планирования задач.

Поддержка компилятором

На рисунке 2 изображены различные этапы работы и компоненты компилятора. А именно: ориентированный на конкретный язык препроцессор, оптимизатор и машинно-зависимый блок генерации кода. Препроцессор распознает директивы OpenMP, обрабатывает их, и передает полученную информацию оптимизатору. Оптимизатор обрабатывает эту информацию и преобразует код так, что он может выполняться несколькими нитями одновременно. В процессе преобразования кода оптимизатор добавляет обращения к библиотеке поддержки OpenMP, libmtsk. И, наконец, генератор кода формирует машинный код.

Когда оптимизатор обрабатывает конструкцию PARALLEL в программе, происходит следующее:

Рисунок 2: Этапы компиляции

Во-первых, оптимизатор анализирует области действия переменных в конструкции PARALLEL. То есть он определяет имеют ли используемые в теле

конструкции PARALLEL переменные опции SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE или REDUCTION и пр.

Во-вторых, оптимизатор извлекает тело конструкции PARALLEL и размещает его в отдельных выделенных функциях (outlined routine). Переменный с опцией SHARED передаются этим функциям в качестве аргументов, то есть могут быть использованы несколькими нитями. Переменный с опцией PRIVATE определяется локально выделенной функции (outlined routine), так что отдельные копии этой переменной размещаются в стеках различных нитей. В outlined routine добавляется дополнительный код для инициализации переменной с опцией FIRSTPRIVATE, обновления переменных с опцией LASTPRIVATE, объединения результатов преобразований, пр.

В-третьих, оптимизатор заменяет тело конструкции PARALLEL вызовом функции __mt_MasterFunction_ библиотеки libmstk. Адрес выделенной функции передается в качестве аргумента __mt_MasterFunction_. Во время своего выполнения __mt_MasterFunction_ распределяет нити на выполнение выделенной функции (outlined routine).

Описанное выше преобразование имеет свои достоинства. Во-первых, выделенная функция (outlined routine) определяет контекст для параллельной работы, который легко может быть выполнен несколькими нитями. Во-вторых, упрощается управление памятью, т.к. переменные локальные в выделенной функции (outlined routine) будут автоматически размещены в различные стеки нитей, таким образом делая их локальными для каждой нити.

Рисунки 3а и 3б показывают суть преобразования. Тело конструкции PARALLEL на рисунке 3а извлекается и помещается в выделенную функцию (outlined routine) __mf_par_001. Так как у переменной n стоит опция SHARED в конструкции PARALLEL, ее адрес передается __mf_par_001 в качестве аргумента, так что все нити выполняющие __mf_par_001 будут иметь доступ к одной и той же переменной. С другой стороны, так как у переменной id стоит опция PRIVATE, она определяется локально функции __mf_par_001, так что у каждой нити, выполняющей __mf_par_001 будет своя копия переменной в стеке.

Рисунок 3б показывает как конструкция PARALLEL заменяется вызовом функции __mt_MasterFunction_, и адрес __mf_par_001 передается ей в качестве аргумента.

Другие конструкции, такие как параллельный DO/for и SECTIONS, обрабатывается оптимизатором примерно также. Оптимизатор, однако, заменяет параллельную конструкцию на вызов функции __mt_WorkSharing_ библиотеки libmstk.

```
    program  test
    integer n, id
    ...
!$omp parallel shared(n), private(id)
    id = omp_get_thread_num()
    call work (n, id)
!$omp end parallel
    ...
    end
```

Рисунок 3а: Пример программы с конструкцией PARALLEL

```
program test
integer n, id
task_info_t taskinfo
...
taskinfo = ...
call __mt_MasterFunction_
  (taskinfo, _mf_par_001, ...)
...
end

subroutine _mf_par_001 (n)
integer n       ! shared
integer id      ! private

id = omp_get_thread_num()
call work (n, id)
end subroutine _mf_par_001_
```

Рисунок 3б: Код, сгенерированный компилятором, содержащий вызов __mt_MasterFunction_ и выделенную функцию (outlined routine)

Автоматическое распараллеливание

Помимо OpenMP распараллеливания основанного на директивах, оптимизатор может также автоматически распараллеливать циклы. Когда программа компилируется с ключом -xautopar, оптимизатор исследует все циклы в программе и использует анализаторы потока данных чтобы определить какие циклы содержат итерации, которые могут быть выполнены независимо друг от друга. Затем оптимизатор преобразует их способом, похожим на описанный выше для OpenMP.

Поддержка библиотеки времени выполнения

Библиотека времени выполнения OpenMP, libmstk, обеспечивает поддержку управления нитями, синхронизацией и планированием задач. Библиотека является надстройкой более высокого уровня над библиотекой нитей POSIX (libpthreads).

Как было описано выше, оптимизатор заменяет код конструкции PARALLEL вызовом __mt_MasterFunction_. Когда нить вызывает __mt_MasterFunction_, она создает набор нитей для выполнения конструкции PARALLEL и становится мастер-нитью набора. Мастер-нить распределяет подчиненные нити для выполнения выделенной функции (outlined routine). Сама мастер-нить тоже принимает участие в выполнении выделенной функции. По окончании, мастер-нить синхронизируется с остальными нитями набора через вызов барьерной функции __mt_EndOfTask_Barrier_. Общая логика __mt_EndOfTask_Barrier_ показана на рисунке 4.

Библиотека libmstk реализует пул нитей, которые могут быть использованы в качестве подчиненных нитей в конструкциях PARALLEL. Нити в пуле создаются при помощи вызовов функции pthread_create библиотеки нитей POSIX. Когда мастер-нить должна создать набор из более чем одной нити, мастер нить проверяет пул и захватывает незанятые нити из него, делая их подчиненными

в наборе. Когда набор заканчивает выполнение блока PARALLEL, подчиненные нити возвращаются в пул.

В течение своего жизненного цикла подчиненная нить выполняет функцию времени выполнения slave_startup_function, где она попеременно ждет следующей задачи от PARALLEL и выполняет эту задачу. Во время ожидания подчиненная нить может ожидать в состоянии занятости (spinning) либо спать (sleeping). Поведение может быть задано с помощью переменной среды SUNW_MP_THR_IDLE. Когда нить заканчивает выполнение задачи, она синхронизируется с мастер-нитью и другими нитями набора при помощи вызова барьерной функции (barrier routine) _mt_EndOfTask_Barrier_. Общая логика slave_startup_function показана на рисунке 5.

OpenMP допускает определение блоков PARALLEL друг в друге. Библиотека libmstk поддерживает вложенный параллелизм. Если вложенный параллелизм разрешен установкой переменной среды OMP_NESTED или вызовом функции omp_set_nested, тогда вложенный блок PARALLEL может выполняться набором, состоящим более чем из одной нити.

В дополнение libmstk поддерживает пользовательские нити. Если программа распараллеливается явными вызовами функций библиотеки нитей POSIX (libpthread), libmstk будет обращаться с каждой пользовательской нитью как с мастер-нитью и поддерживать ее своим собственным набором подчиненных нитей.

```
__mt_MasterFunction_ (taskinfo, ...)
{
  - Создать набор нитей.
  - Распределить подчиненные нити для
  выполнения задачи.
  - Выполнить задачу.
  - Ждать у барьера пока другие нити
  выполнят задачу.
}
```

Рисунок 4: Общая схема функции **__mt_MasterFunction()**

```
slave_startup_function_ ()
{
  ...
  while (1)
  {
     - Ждать следующей доступной
     задачи.
     - Выполнять задачу.
     - Ждать у барьера пока другие нити
     выполнят задачу.
  }
}
```

Рисунок 5: Общая схем функции **slave_startup_function()**

Инструментальная поддержка

Комплект программной разработки Sun Studio обеспечивает доступ к широкому спектру инструментов, облегчающих и поддерживающих OpenMP программирование. Они включают как инструменты, помогающие программисту распараллеливать программу при помощи OpenMP, так и инструменты проверки, отладки и анализа производительности программ на OpenMP. Некоторые из этих средств описаны ниже.

Автоматическая оценка областей действия переменных

Процесс ручного задания областей определения переменных при написании программы на OpenMP является одновременно утомителен и располагающим к ошибкам. Для повышения эффективности в Sun Studio реализована возможность

применять автоматическое определение областей действия, являющееся Sun-ориентированным расширением OpenMP. На данный момент компиляторы Sun Studio являются единственными общедоступными компиляторами, поддерживающими эту возможность.

Возможность автоматического определения усиливает аналитические возможности оптимизатора по определению необходимых областей действия переменных. Программист задает те переменные в данной конструкции PARALLEL, для которых оптимизатором должна быть автоматически определена область действия. Оптимизатор определяет необходимую область действия для этих переменных, анализирую программу и применяя набор правил для автоопределения. Результаты показываются в исходном коде программы в виде комментария компилятора. Возможность автоматического задания областей действия предлагает компромисс между автоматическим и ручным распараллеливанием.

Для дополнительной информации стоит обратиться к [7].

Статическая проверка ошибок

Опции компилятора -vpara (Fortran) или -xvpara (C) управляют проверкой программы компилятором на наличие статических ошибок. Они включают неверную вложенность конструкций OpenMP, неверное задание областей действия переменных, гонки сигналов, пр.

Также с опцией -XListMP компилятор Fortran может выполнять межфункциональные анализы кода программы и сообщать о противоречиях и возможным проблемах во время выполнения. Проблемы, о которых компилятор может сообщить, включают в себя неправильное использование директив OpenMP, ошибки погрешностей, несоответствие числа или типа аргумента функции, пр.

Проверка ошибок времени выполнения

В случае, если переменная окружения SUNW_MP_WARN установлена в TRUE, библиотека времени выполнения проверяет программу на разнообразные ошибки выполнения. Набор отслеживаемых проблем включает в себя семантические ошибки, которые нарушают спецификацию OpenMP, неправильное вложение конструкций OpenMP, противоречивость использования директив OpenMP, неверные размеры блоков данных, тупиковые ситуации у барьеров.

Отладка программ OpenMP

Для отладки программ на C, C++ и Fortran’e в Sun Studio используется средство dbx. Сначала программа должна быть подготовлена к отладке - скомпилирована с опцией -xopenmp=noopt -g.

Все команды dbx, которые используются для работы с нитями, могут быть использованы для отладки OpenMP. dbx позволяет выполнять секцию PARALLEL пошагово, устанавливать точки останова в теле конструкций OpenMP, также как и отображать значения SHARED, PRIVATE, THREADPRIVATE и др. переменных для заданной нити.

Анализ производительности

Сборщик и анализатор производительности [8] - инструменты для сбора из анализа данных о производительности приложения. Средства могут быть

использованы как из командной строки, так при помощи графического интерфейса.

Средство сборщик собирает данные о производительности использую статистический метод - профилирование - и с помощью вызова функций трассировки. Данные могут включать стеки вызовов, информацию о микросостояниях, времена приостановки нитей для синхронизации, данные счетчика переполнений, данные о распределении памяти и краткую информацию об операционной системе и процессе.

Анализатор производительности обрабатывает данные полученные от сборщика и выводит различные показатели производительности уровней программы, функций, обращений и уровня исходного кода. Анализатор может также показывать необработанные данные в графическом виде как функцию времени.

Анализатор может представлять информацию о производительности программы на OpenMP в двух режимах: пользовательском и машинном. В пользовательском режиме анализатор представляет данные профайлинга в виде, доступном интуитивному пониманию пользователя. В этом режиме стеки вызовов мастер-нити и подчиненных нитей согласованы друг с другом, искусственно созданные функции со стандартными именами добавляются к стеку вызовов во время выполнения библиотеки OpenMP определенных операций.

В машинном режиме анализатор производительности показывает стек вызовов как есть: без преобразований и добавления искусственных функций -, таким образом раскрывая детали реализации библиотеки времени выполнения libmtsk.

Комментарии компилятора

Комментарии компилятора в исходном коде доводят до сведения пользователя информацию о различных преобразованиях и изменениях с целью оптимизации сделанных в исходном коде компилятором. Для генерации комментариев компилятора программа должна быть скомпилирована с ключом -g. Комментарии компилятора в исходном коде можно увидеть с помощью анализатора производительности или утилиты командной строки er_src.

Производительность OpenMP

Существует тестовая программа SPEC OMP2001 разрабатываемая группой высокопроизводительных систем (High-Performance Group, HPG) корпорации стандартизации оценки производительности (Standart Performance Evaluation Corporation, SPEC). Тест разработан для оценки производительности реальных научных и инженерных приложений распараллеленных при помощи OpenMP и представляет собой набор программ расчета из таких областей знаний как химия, механика, моделирование климата и физика (см. Таблицу 1).

Тест SPEC OMP2001 включает в себя два набора. Первый - SPEC OMPM2001, использует набор данных среднего размера и разработан для измерения производительности систем с разделяемой памятью с количеством процессоров от 4 до 32. Второй - SPEC OMPL2001, использует набор данных большого размера и разработан для измерения производительности систем с разделяемой памятью с бОльшим количеством процессоров.

Наименование	**Область применения**	**Язык прогр-я**
311.wupwise	Квантовая хромодинамика	Fortran
313.swim	Решение уравнений «мелкой воды»	Fortran
315.mgrid	Вычисления на множестве сеток	Fortran
317.applu	Решение уравнений в частных производных	Fortran
321.equake	Моделирование сейсмических задач	C
325.apsi	Расчет распространения загрязняющего вещества в зависимости от погодных условий.	Fortran
327.gafort	Генетические алгоритмы	Fortran
329.fma3d	Моделирование механических деформаций	Fortran
331.art	Моделирование нейронных сетей	C
318.galgel	Численные вычисления параметров движения жидкости в закрытом пространстве	Fortran
332.ammp	Вычислительная химия	C

Таблица 1: Тесты SPEC OMP2001

SPEC OMP2001 показывает выдающуюся масштабируемость и производительность на системах фирмы Sun. Рисунок 6 показывает масштабируемость набора SPEC OMPM2001 на системе Sun Fire 6800 построенной на 1.2Ггц UltraSPARC III Cu процессорах. Рисунок 7 показывает масштабируемость набора SPEC OMPL2001 на системе Sun Fire 15k построенной на 1.2Ггц UltraSPARC III Cu процессорах.

Компания Sun Microsystems объявила о нескольких мировых рекордах в области производительности, зафиксированных SPEC OMP2001:

> В июне 2003г. Sun Microsystems объявила о мировом рекорде зафиксированном SPEC OMPL2001 (пиковая производительность) на системе Sun Fire 15k построенной на 72 1.2Ггц UltraSPARC III Cu процессорах. Sun Fire 15k стал первым сервером, преодолевшим отметку 200,000 достигнув результата в 213,466 балла.
>
> В феврале 2004г. Sun Microsystems установила новый мировой рекорд зафиксированном SPEC OMPL2001 (пиковая производительность) на системе Sun Fire E25K построенной на 72 1.2Ггц UltraSPARC IV процессорах. Sun Fire E25K достиг результата в 316,182 балла.

В марте 2005г. Sun Microsystems объявила о мировом рекорде зафиксированном SPEC OMPM2001 в дву- и четырехнитевой категории. Пиковой производительности в 12,434 балла достигла машина Sun Fire V40z с четырьмя процессорами превзошел другие коммерческие решения сходного класса на более чем 43 процента.

Результаты SPEC OMP2001 в более подробной форме можно найти на [10].

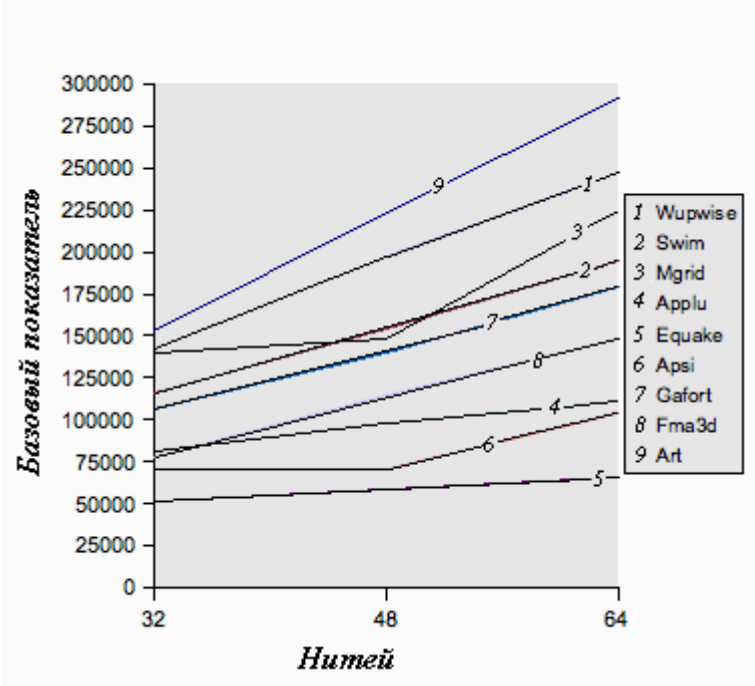

Рисунок 6: Масштабирование OMPL2001 (базовое) на Sun Fire 6800

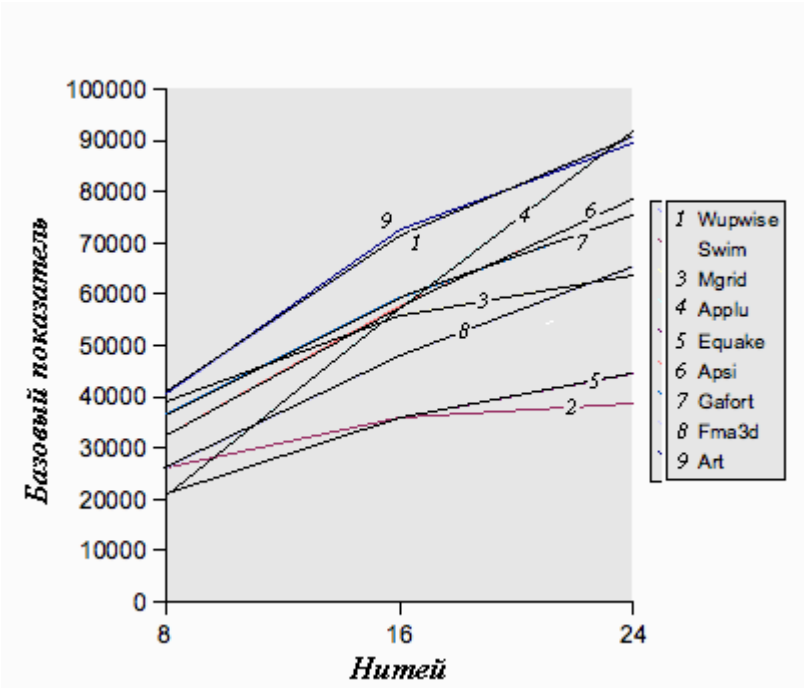

Рисунок 7: Масштабирование OMPL2001 (базовое) на Sun Fire 15k

Перспективные направления

Sun Microsystems продолжает высококачественную поддержку OpenMP в своих компиляторах и инструментах. Текущие и будущие проекты включают в себя следующее:

Новые возможности OpenMP. Sun продолжает отслеживать изменения в спецификации OpenMP и активно участвовать в ее развитии.

OpenMP-ориентированнам оптимизация для лучшей производительности. Это включает в себя как оптимизацию компилятора, такую как удаление излишних барьеров, так и оптимизацию библиотеки времени выполнения для уменьшения накладных расходов на распараллеливание.

Улучшенная инструментальная поддержка. Sun продолжает разрабатывать инструментальные средства помогающие программисту писат, отлаживать и анализировать производительность программы на OpenMP. Эти инструменты включают в себя улучшенную автоматической оценки областей действия переменных, обнаружение гонок данных, а также интерактивные средства распараллеливания приложений.

Архитектурная поддержка. Sun продолжает расширять границы применения OpenMP. Работа в этом направлении включает в себя улучшения производительности на машинах с гибридной памятью (Non-uniform Memory Access, NUMA) и машинах с технологией многопотоковости на кристалле (Chip Multi-threading, CMT).

Ссылки

1. OpenMP Specification, http://www.openmp.org/drupal/node/view/8
2. OpenMP Architecture Review Board, http://www.openmp.org
3. Sun Studio Software, http://www.sun.com/software/products/studio/index.html
4. Sun Studio 11 OpenMP API User's Guide, http://docs.sun.com/doc/819-3694
5. Sun Fire E25K server, http://www.sun.com/servers/highend/sunfire_e25k/index.xml
6. The SPEC OMP benchmark suite, http://www.spec.org/omp
7. Yuan Lin, Christian Terboven, Dieter an Mey, and Nawal Copty, "Automatic Scoping of Variables in Parallel Regions of an OpenMP Program", WOMPAT 2004. (PDF)
8.Sun Studio 11: Performance Analyzer, http:/docs.sun.com/app/docs/doc/819-3687
9. Myungho Lee, Brian Whitney, and Nawal Copty, "Performance and Scalability of OpenMP Programs on the Sun Fire E25K Throughput Computing Server", WOMPAT 2004.
10. SPEC OMP2001 benchmarks results, http://www.spec.org/omp/results

Приложение П.1: Пример использования директивы PARALLEL

Далее приведена простая программа “Hello,World” с директивой PARALLEL. Число нитей задано при помощи вызова функции omp_set_num_threads. Динамическое регулирование числа нитей отключено при помощи вызова функции omp_set_dynamic.

Начальная нить программы выполняется последовательно пока не достигнет конструкции PARALLEL. В этой точке начальная нить создает 9 других (подчиненных). Все нити набора выполняют код заключенный в конструкции PARALLEL одновременно. Когда нить достигает конца конструкции PARALLEL, она ждет там у неявного барьера. Когда все нити достигнут этой точки, далее только мастер-нить продолжает выполнение кода, следующего за конструкции PARALLEL.

Пример конструкции PARALLEL на Fortran:

```
       PROGRAM HELLO
       USE OMP_LIB
       INTEGER TID
       CALL OMP_SET_DYNAMIC (.FALSE.)
       CALL OMP_SET_NUM_THREADS (10)
!$OMP PARALLEL PRIVATE (TID)
! Obtain thread ID.
       TID = OMP_GET_THREAD_NUM()
! Print thread ID.
       PRINT *, 'Hello World from thread = ', TID
!$OMP END PARALLEL
       END
```

Пример конструкции PARALLEL на C/C++:

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
  int tid; omp_set_dynamic(0);
  omp_set_num_threads(10);
#pragma omp parallel private(tid)
  {
/* Получение идентификатора нити. */
    tid = omp_get_thread_num();
/* Print thread ID. */
    printf ("Hello World from thread = %d\n", tid);
  }
}
```

Приложение П.2: Пример использования директивы DO/for

Далее приведена программа с директивой DO/for. Начальная нить программы выполняется последовательно пока не достигнет конструкции PARALLEL. В этой точке начальная нить создает 20 других (подчиненных). В наборе содержится начальная нить (мастер-нить) и 19 других нитей (подчиненных нитей набора). Все нити набора выполняют код заключенный в конструкции PARALLEL одновременно.

Когда нити набора достигают конструкции DO/for, 100 итераций цикла распределяются между 20 нитями. То есть каждая нить выполняет по 5 итераций цикла. Нити выполняют свои итерации одновременно. Когда нить заканчивает свою работу, она ожидает у неявного барьера в конце конструкции DO/for. Когда все нити достигают барьера, они продолжают выполнять код области PARALLEL.

Директива DO в программе на Fortran:

```
       PROGRAM VECTOR_ADD
       USE OMP_LIB
       PARAMETER (N=100)
       INTEGER N, I
       REAL A(N), B(N), C(N)
       CALL OMP_SET_DYNAMIC (.FALSE.)
       CALL OMP_SET_NUM_THREADS (20)
! Инициализация массивов a и b.
       DO I = 1, N
         A(I) = I * 1.0
         B(I) = I * 2.0
       ENDDO
! Параллельно вычисляет значения элементов массива.
!$OMP PARALLEL SHARED(A, B, C), PRIVATE(I)
!$OMP DO
       DO I = 1, N
         C(I) = A(I) + B(I)
       ENDDO
!$OMP END PARALLEL
       PRINT *, C(10)
       END
```

Директива for в программе на C/C++:

```
#include <stdio.h>
#include <omp.h>
#define N 100
int main(void)
{
  float a[N], b[N], c[N];
  int i;
  omp_set_dynamic(0);
  omp_set_num_threads(20);
/* Инициализация массивов a и b. */
  for (i = 0; i < N; i++)
    {
      a[i] = i * 1.0;
      b[i] = i * 2.0;
    }
/* Параллельно вычисляет значения элементов массива. */
#pragma omp parallel shared(a, b, c) private(i)
  {
```

```
#pragma omp for
    for (i = 0; i < N; i++)
      c[i] = a[i] + b[i];
  }
  printf ("%f\n", c[10]);
}
```

Приложение П.3: Пример использования директивы Sections

Далее приведен пример программы с директивой SECTIONS примененной к трем секциям кода. Исходная нить программы выполняется последовательно пока не достигнет конструкции PARALLEL. В этой точке исходная нить создает набор из 3 нитей. Набор состоит из исходной нити (мастер-нити) и 2х других (подчиненных). Все нити в наборе выполняют код заключенный в конструкции PARALLEL одновременно.

Когда нити набора достигают конструкции SECTIONS, 3 секции распределяются между 3 нитями набора. Каждая секция выполняется только один раз нитью набора. Когда нить завершает выполнение, она ожидает у неявного барьера в конце конструкции SECTIONS. Когда все нити достигнут барьера, набор нитей продолжает выполнять код области PARALLEL.

Пример директивы SECTIONS в программе на Fortran:

```
       PROGRAM SECTIONS
       USE OMP_LIB
       INTEGER SQUARE
       INTEGER X, Y, Z, XS, YS, ZS
       CALL OMP_SET_DYNAMIC (.FALSE.)
       CALL OMP_SET_NUM_THREADS (3)
       X = 2
       Y = 3
       Z = 5
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
       XS = SQUARE(X)
       PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "XS =", XS
!$OMP SECTION
       YS = SQUARE(Y)
       PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "YS =", YS
!$OMP SECTION
       ZS = SQUARE(Z)
       PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "ZS =", ZS
!$OMP END SECTIONS
!$OMP END PARALLEL
       END
       INTEGER FUNCTION SQUARE(N)
       INTEGER N
       SQUARE = N*N
       END
```

Пример директивы SECTIONS в программе на C/C++:

```
#include <stdio.h>
#include <omp.h>
int square(int n);
int main(void)
{
  int x, y, z, xs, ys, zs;
  omp_set_dynamic(0);
  omp_set_num_threads(3);
  x = 2;
  y = 3;
  z = 5;
#pragma omp parallel
  {
```

```
#pragma omp sections
    {
#pragma omp section
      {
         xs = square(x);
        printf ("id = %d, xs = %d\n", omp_get_thread_num(), xs);
      }
#pragma omp section
      {
        ys = square(y);
        printf ("id = %d, ys = %d\n", omp_get_thread_num(), ys);
      }
#pragma omp section
      {
        zs = square(z);
        printf ("id = %d, zs = %d\n", omp_get_thread_num(), zs);
      }
    }
  }
}
int square(int n)
{
return n*n;
}
```