УДК 681.3.06:004.4:517.95

# Технологии программирования OpenMP и OpenACC на суперкомпьютерах и современный FORTRAN в задачах МДТТ

**Л. В. Ландик, И. В. Пестренина, В. М. Пестренин**
Пермский государственный национальный исследовательский университет
Россия, 614990, г. Пермь, ул. Букирева, 15
Ipestrenina@gmail.com; 8 (342) 2 396 375

Рассмотрены особенности распараллеливания алгоритмов с применением технологий программирования OpenMP и OpenACC, проявившиеся в процессе реализации итерационного конечно-элементного алгоритма для решения неклассических задач механики деформируемого твердого тела, содержащего особые точки. Приведен обзор современных средств создания высокопроизводительного математического обеспечения. Проведено сравнительное исследование распараллеливания алгоритмов с использованием OpenMP и OpenACC на компиляторах INTEL и PGI. На примерах операций умножения полных и разреженных матриц приведены временные характеристики компиляторов и технологий распараллеливания, влияние на эти характеристики встроенных функций современного языка Fortran и представления вещественных чисел. Показаны достоинства численных методов для разреженных матриц. Даются практические рекомендации по применению алгоритмов на языке Fortran и технологий OpenMP и OpenACC.

**Ключевые слова:** *алгоритмы распараллеливания; современный Fortran; метод конечных элементов; OpenMP; OpenACC.*

DOI: 10.17072/1993-0550-2018-3-54-68

## Введение

Современная вычислительная техника, используемая для решения проблем механики деформируемого твердого тела (МДТТ), – это мощные высокоскоростные многопроцессорные и многоядерные суперкомпьютеры, гетерогенные вычислительные кластеры, имеющие огромные ресурсы разделяемой и разделенной оперативной и внешней памяти – позволяет решать реальные практические задачи науки и техники [5–28]. При этом по-прежнему остаются важными следующие факторы – это физико-математическая постановка задачи, алгоритм решения, эффективные высокоточные численные методы и оптимально-разумное время выполнения программного модуля. Одна из особенностей современных задач – это большие объемы информации и множество матричных операций, которые требуют существенных затрат машинного времени. Снижение времени счета – одна из насущных проблем реализации построенных алгоритмов, которая зависит в первую очередь от самого алгоритма, от выбора алгоритмического языка и соответствующего компилятора, поддерживающего современные технологии программирования и эффективные библиотеки численных методов.

Архитектура гетерогенного вычислительного кластера объединяет в себе мультикомпьютер, состоящий из множества отдельных компьютеров-узлов вычислительного кластера [10, 11], построенных на базе центральных процессоров с несколькими ядрами (CPU) и нескольких графических процессоров-ускорителей (GPU), которые являются сопроцессорами для главного host-процессора. Коммуникационная среда вычислительных кластеров позволяет узлам взаимодействовать между собой посредством передачи сообщений. Современные GPU, как и узлы

кластера, имеют собственную память и средства кэширования для ускорения операций обмена [11–13]. В целом, кластер следует рассматривать как единую аппаратно-программную систему, имеющую единую коммуникационную систему, единый центр управления и планирования загрузки. Для использования ресурсов такой вычислительной системы необходимо оптимально распределять нагрузку на центральные и графические процессоры. Такое распараллеливание вычислений предполагает использование сразу нескольких технологий параллельного программирования, например, MPI для организации обменов данными между узлами кластера, OpenMP [8–10] для организации нескольких потоков процессора локально в рамках одного узла кластера, CUDA NVIDIA [11, 22–27] для организации вычислений на графических ускорителях.

Изначально кластер чаще всего однородный, но в процессе наращивания становится, как правило, неоднородным. Неоднородность создает ряд проблем. Различие в производительности процессоров усложняет задачу распределения работ между процессорами. Различие в архитектуре процессоров требует подготовки разных выполняемых файлов для разных узлов, а в случае различий в представлении данных может потребовать преобразования их форматов при передаче сообщений между узлами.

Применение многопроцессорных компьютеров и соответствующих технологий распараллеливания алгоритмов позволяет существенно уменьшать время вычислений за счет параллельной обработки данных. Основные критерии качества параллельной реализации алгоритма: это *ускорение* расчетов с ростом числа процессоров, *эффективность* и *адекватность* воспроизведения моделируемых явлений [11, 15]. Ускорение определяют как отношение времени счета последовательного алгоритма ко времени счета параллельного алгоритма, а эффективность – как отношение ускорения к количеству процессоров, на котором оно достигнуто [16, 17], в процентах. На практике обычно происходит снижение эффективности с ростом числа процессоров. Это связано со следующими факторами: программы могут иметь последовательные фрагменты, может иметь место разбалансировка вычислений в параллельных процессах, поэтому некоторое время может тратиться на межпроцессорные обмены. Для сбалансирования вычислений и минимизации обменов ключевая роль отводится выбору способа распределения данных и вычислений по процессорам. И, наконец, адекватность воспроизведения явлений – это результаты моделирования на нескольких процессорах, которые должны совпадать с результатами моделирования на одном процессоре с некоторой точностью, определяемой постановкой задачи. Установлено, что результаты вычислений трансцендентных и тригонометрических функций на GPU могут отличаться в младших битах от результатов на host-процессоре [11–13]. Алгоритм должен быть устойчив к такого рода ошибкам.

Проблема адаптации математических моделей к многопроцессорным вычислительным комплексам – это проблема наиболее эффективной реализации алгоритмов с сохранением точности результатов моделирования. Наиболее эффективным для многомерных задач МДТТ на многопроцессорных вычислительных системах с распределенной памятью считается принцип геометрического параллелизма, который предполагает декомпозицию расчетной области на подобласти соответственно числу процессоров. Технология этого принципа основана на разделении области по процессорам, исходя из требования равномерной загрузки. При этом разбиение области происходит строго по блокам. Если размерность сетки в блоке больше средней размерности в расчете на один процессор, то этот блок обслуживается несколькими процессорами, и наоборот, один процессор обслуживает несколько соседних блоков, если их суммарная размерность не превышает средней.

Для распределения одного блока между несколькими процессорами можно использовать 1D-, 2D-, 3D-разбиения. Сравнение показало преимущество 3D-сетки, так как число поверхностей, через которые будет происходить обмен данными, минимально. На наш взгляд, целесообразно выбирать способ разбиения из минимума объема пересылок. Для реализации алгоритма геометрического параллелизма разработаны библиотеки обмена сообщениями MPI [9, 15]. При этом каждый процессор кластера выполняет одни и те же вычисления для части расчетной области, распределенной на этот процессор. Вычисления сводятся к взаимно согласованной поэтапной реализации метода расщепления по пространственным переменным. Исключения

составляют процессоры, которые дополнительно выполняют склейку решений на внутренних границах. Условия склейки выполняются на каждом шаге по времени. Процессоры, обслуживающие соседние блоки, передают необходимую информацию одному из таких процессоров, который производит автономный расчет всей границы в целом и рассылает результаты в обратном направлении. Это происходит параллельно по всем блокам, незначительная задержка может возникнуть только из-за необходимости передачи данных исполняющим процессорам для склейки решений на противоположной границе своего блока. При этом все процессоры, кроме исполняющихся (т.е. активных в данный момент времени), находятся в состоянии ожидания.

Разработка параллельных программ усложняется также из-за следующих проблем: ресурсы (количество узлов, их архитектура, производительность), определяются только в момент обработки сетью заказа на выполнение задачи. Поэтому программисту приходится разрабатывать программу так, чтобы она могла динамически (без перекомпиляции) самонастраиваться на выделенную конфигурацию сети, с учетом неоднородности коммуникационной среды, что представляет собой весьма непростую задачу. В результате эффективная производительность кластерных вычислительных систем (**real applications performance** – **RAP**) специалисты оценивают как 5–15 % от их пиковой производительности [10, 27]. Для сравнения: у лучших малопроцессорных систем это соотношение оценивается как 30–50 %.

Здесь сразу можно отметить – повышается потребность в системах программирования высокого уровня, которые по возможности скрывают от программиста вопросы распределения ресурсов и автоматически обеспечивают динамические свойства прикладных параллельных программ, позволяя сосредоточиться на самом алгоритме задачи. Например, наиболее используемый стандарт MPI имеет недостаточно высокий уровень и заставляет пользователя оперировать примитивами типа "принять/получить" сообщение [9, 20]. Применение технологии CUDA также требует от программиста знания самой технологии и хороших знаний архитектуры используемой вычислительной системы. В результате в ноябре 2011 года фирма PGI (вместе с NVIDIA и др.) предложила технологию директивного (как в OpenMP) метода программирования OpenACC API для **host** + GPU, скрывающую от программиста все детали обращения к технологии CUDA [11].

В работе [28] авторы рассмотрели и дали несколько рекомендаций для эффективного использования гетерогенного кластера: нужно выделить наиболее критические фрагменты алгоритма, которые можно эффективно реализовать на графических ускорителях. На архитектуре кластера авторы реализовали сразу нескольких шаблонов параллелизма (MPI, OpenMP, CUDA). Для этого на каждом узле кластера запускается один MPI-процесс, который затем разветвляется на несколько процессорных потоков с помощью технологии OpenMP [8–10]. Технология OpenMP реализует параллельные вычисления с помощью многопоточности, в которой "главный" (**master**) поток создает набор "подчиненных" (**slave**) потоков и между ними распределяются вычисления. Затем с помощью директив препроцессора OpenMP порождается параллельная область с заданным количеством потоков CPU равным количеству GPU, и для каждого потока назначается **device**-устройство с помощью функции **cudaSetDevice**(ndev), где ndev – номер устройства. После этого на каждом GPU размещается соответствующая часть данных, например, часть матрицы необходимого размера и параллельная область закрывается. Далее, например, при умножении матрицы на вектор открывается параллельная область, при этом каждый поток использует то устройство, которое ему назначено. Это возможно благодаря тому, что OpenMP использует множество потоков, которое не уничтожается между параллельными и последовательными областями, а управление передается либо заданному количеству потоков для параллельной области, либо главному потоку для последовательной области. Постоянная поддержка множества потоков дает выигрыш в производительности, так как создание потоков в каждой параллельной области является медленной операцией [8–10, 28].

При изучении аспектов процесса распараллеливания алгоритмов для научных исследований нельзя обойти вниманием вопрос влияния на производительность представления данных с различной точностью [26], поскольку очень часто реальные расчеты необходимо вести с более высокой точностью (стандартной двойной – **real*8**, максимально

возможной – **real*16**). Согласно техническим характеристикам, производительность процессоров-узлов кластера сильно зависит от используемой точности вычислений. Наибольшая производительность достигается при вычислениях с одинарной точностью представления данных. При вычислениях с двойной точностью производительность снижается теоретически на порядок (при сравнении с пиковой производительностью, заявленной производителем) и примерно в два раза на практике [27].

Снижение быстродействия вычислений на графических ускорителях существенно больше, чем на обычных процессорах-узлах кластера. Поэтому фирмы-разработчики ускорителей работают над созданием новых моделей графических ускорителей. Так, например, в последних моделях графических процессоров компании NVIDIA производительность вычислений с двойной точностью была повышена в четыре раза [14].

Следует отметить, что производительность графических ускорителей повышается с увеличением размерности математической модели, так как время вычислений на графических процессорах увеличивается, а время обмена данными по сети снижается.

В научной литературе (при всей актуальности проблемы) недостаточно освещено создание эффективных программ задач МДТТ для суперкомпьютеров. При этом следует отметить, что широко известные пакеты программ не всегда могут правильно выполнить решение конкретной задачи, т.е. у пользователя возникает необходимость создать свою программу. Для решения таких задач часто используется метод конечных элементов (МКЭ) и множество его модификаций. Главная особенность МКЭ – большое количество матричных операций в процессе сборки базовой разрешающей системы уравнений МКЭ и ее прямоугольных модификаций, а также их сильная разреженность. На реальной конечно-элементной сетке размерности матриц могут быть очень большими, в результате выполнение решения требует значительных затрат машинного времени. Уменьшение временных затрат можно выполнить двумя способами: использовать высокоэффективные численные методы и/или применить технологии распараллеливания алгоритма.

Следует заметить, что программисту бывает трудно сориентироваться в выборе инструментов для реализации оптимального кода. Актуальным является сравнительный анализ эффективности применяемых приемов распараллеливания и выработка соответствующих рекомендаций.

В работе приведены результаты не только тестовых, но и реальных исследований [1–3] по применению современного стандарта алгоритмического языка Fortran и технологий распараллеливания OpenMP и OpenACC для моделей с разделяемой памятью, поддерживаемых компиляторами компаний INTEL и PGI. Был проведен сравнительный анализ компиляторов Intel и PGI для выбранных технологий распараллеливания и необходимой точности вычислений. Для сравнительного анализа был выбран современный стандарт языка Fortran [5, 6], так как он отвечает необходимым требованиям в наличии компиляторов, поддерживающих современные технологии программирования, разработанным известным пакетам программных комплексов (ANSYS, Fluent, и др.), а также созданным эффективным библиотекам численных методов. Компиляция и реальные расчеты были выполнены на суперкомпьютере с параллельной архитектурой **TESLA PGU** в Пермском государственном научном исследовательском университете.

## 1. О современном стандарте языка Fortran

**Fortran** – один из старейших алгоритмических языков, был разработан для решения инженерных задач и всегда был востребован для реализации практических задач науки и техники. Язык непрерывно развивается (Fortran 77, 90/95, 2003, 2008), удовлетворяет современным требованиям программирования [5, 6]: – динамическому выделению памяти, векторному программированию, эффективным встроенным функциям для матричных операций, неформатным файлам прямого и последовательного доступа и т.д.

Средства параллельного программирования впервые были введены в стандарт языка Fortran 90. Это было вызвано появлением в компьютерах аппаратных средств векторной обработки данных. В язык были введены средства явной спецификации векторных операций – это возможность работы с массивами и сечениями массивов как с целыми объектами на поэлементной основе; это встроенные функции для работы с массивами, позволяю-

щие эффективно использовать особенности архитектуры, включая параллельное выполнение. В последующих версиях языка поддержка параллельности получила дальнейшее развитие. Разработанные для многопроцессорных систем технологии параллельного программирования (OpenMP, MPI, OpenACC, и др.) являются фактически расширениями языка Fortran, поскольку используют новые возможности языка. Современный Fortran позволяет разрабатывать лучше структурированные программы, а значит их легче распараллелить.

Применение языка Fortran вызвано также тем, что именно на нем реализовано и опубликовано в литературе по вычислительной технике большинство высокоэффективных алгоритмов по численным методам. Многие исследовательские лаборатории работают над созданием не только эффективных численных методов, но и библиотек по численным методам, реализованных на языке Fortran. Например, в лаборатории the Numerical Analysis Group at the STFC Rutherford Appleton Laboratory (Harwell Oxford, http://www.hsl.rl.ac.uk/) была создана библиотека **HSL** (the Harwell Subroutine Library). Лаборатория, начиная с 1963 г., активно работает над созданием новых и усовершенствованием старых модулей библиотеки. Одно из востребованных направлений их разработок – это высокоскоростные методы решения линейных разреженных систем уравнений. Например, эти методы нашли широкое применение в методе конечных элементов (МКЭ) для решения задач механики деформируемого твердого тела (МДТТ) в самых различных постановках. В частности, в алгоритме решения неклассической задачи МДТТ [1–3], для получения обратных матриц в процессе сборки и решения сильноразреженной разрешающей системы были использованы исходные модули **Ma28** из библиотеки **HSL** модули**,** реализующие *LU*-разложение, а для прямоугольных подсистем был использован алгоритм сингулярного разложения матриц [7], реализованный также на современном стандарте языка Fortran.

## 2. О компиляторах с современного стандарта языка Fortran

Компании-разработчики компиляторов (**Intel, Portland Group, Compaq Visual** и др.) с языка Fortran постоянно реализуют новые элементы языка, учитывают все достижения в производстве вычислительной техники и суперкомпьютеров с параллельной архитектурой. Получаемый в результате модуль – это эффективный программный код, в котором можно предусмотреть графическую визуализацию, распараллеливание алгоритмов и т.д.

**Intel Fortran** – оптимизирующий компилятор, использующий все возможности современного стандарта языка Fortran для процессоров x86, x86-64, генерирует наиболее быстрый код для этих процессоров. В компиляторах Intel Fortran расширена поддержка стандартов языка Fortran 90/95, 2003, 2008. Вещественные данные могут иметь длину 4, 8 и 16 байтов.

Компиляция алгоритма решения неклассических задач МДТТ и реальные расчеты [1–3] были выполнены на суперкомпьютере с параллельной архитектурой **TESLA Fermi K20** в Пермском государственном национальном исследовательском университете. Рассмотрены академические версии компиляторов фирмы **Intel** (http://www.intel.ru, **ifort**, версия 11.8) и **The Portland Group Inc** (https://www.pgroup.com, **PGI**, версии 12.4, 13.1, 13.9). Максимально возможная точность вычислений (**real(16))** в компиляторах фирмы Intel для решения неклассических задач МДТТ оказалась просто необходимой. Наличие особых точек в расчетной области таких задач требует для получения точного решения достаточно мелкой конечно-элементной сетки. В результате при исследовании полей напряжений, имеющих значительные градиенты в малых областях, в которых характерный размер конечных элементов может оказаться настолько малым, что при вычислениях даже с двойной точностью погрешность оказывается соразмерной или даже превышающей величину площади элемента. Это означает, что вычисления нужно вести с максимально возможной точностью.

В компиляторе реализована возможность создания высокопроизводительных многопоточных приложений на основе технологии распараллеливания OpenMP 2.0. Для получения программного модуля с применением OpenMP [8–10] при вызове компилятора предусмотрена опция **– openmp**.

В **PGI**-компиляторе с языка Fortran допустимая длина вещественных данных – 4 и 8 байтов. Генерируемый код уступает по скорости Intel-компиляторам, но в рассмотренных версиях PGI реализована возможность распараллеливания на графических ускорителях.

Директивный язык программирования OpenACC упрощает разработку приложений для GPU. Для получения программного модуля в PGI с применением графических ускорителей в технологии OpenACC предусмотрена опция **– acc**.

Если опция **– openmp** в **Intel** или **acc** в **PGI** не указана, то компилятор все директивы рассматривает как комментарий, а откомпилированный модуль будет работать как однопоточное приложение.

Проведен сравнительный анализ компиляторов Intel и PGI для выбранных технологий распараллеливания и необходимой точности вычислений.

## 3. О технологиях распараллеливания OpenMP и OpenACC

Приведем некоторые понятия, используемые при распараллеливании, как на самом кластере, так и на графических ускорителях GPU. Из достаточно большого количества технологий для распараллеливания алгоритмов выберем технологию OpenMP [8–10] на узлах кластера и OpenACC [11–13] на графических ускорителях (GPU) для распараллеливания программ на языках C, C++ и Fortran.

Здесь за основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Предполагается, что создаваемая параллельная программа будет переносимой между различными компьютерами с разделяемой памятью, поддерживающими **OpenMP API** или **OpenACC**. Обе технологии нацелены на то, чтобы пользователь имел один вариант модуля для параллельного и последовательного выполнения. В общем виде программа – это набор последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода. Использование потоков для организации распараллеливания позволяет учесть преимущества многопроцессорных вычислительных систем с общей памятью. Прежде всего, потоки одной и той же параллельной программы выполняются в общем адресном пространстве, что обеспечивает возможность использования общих данных для параллельно выполняемых потоков без каких-либо трудоемких межпроцессорных передач сообщений (в отличие от процессов в технологии MPI для систем с распределенной памятью). И, кроме того, управление потоками (создание, приостановка, активизация, завершение) требует меньше накладных расходов для операционной системы.

Задача должна быть разделена на независимо выполняемые подзадачи-процессы – это параллельные циклы **do**, секции **sections**, **workshare**, **single** – которые используют собственные адресные пространства и взаимодействуют через специальные механизмы.

Один процесс (цикл **do**) может содержать очень большое число независимых итераций-потоков (**threads**). Например, процесс умножения квадратной матрицы **n**-порядка на вектор можно разбить на **n** подзадач-потоков.

**OpenMP** (**Open Multi-Processing**) – открытый стандарт распараллеливания алгоритмов на масштабируемых ***SMP-системах*** с общей памятью (shared memory model) и создания многопоточного приложения на кластерных системах. Наличие общей памяти упрощает программирование и исключает затраты времени на межпроцессорный обмен, но при одновременном обращении к общим данным нескольких процессоров требует их синхронизации. При этом требования к пропускной способности коммутатора общей памяти чрезвычайно высоки, что ограничивает число процессоров, используемых в системе. На TESLA K20 на каждом узле кластера – 2 SIMD-units с 6 скалярными ядрами (**thread processors**) каждый, т.е. на узле кластера могут работать синхронно 12 потоков и выполнять одну и ту же инструкцию в одно и то же процессорное время. Для пользователя OpenMP это совокупность директив, библиотечных процедур и переменных окружения.

**OpenACC** (for Accelerators) – стандарт, описывающий набор директив, похожих на OpenMP, для написания гетерогенных программ, задействующих как центральный процессор (**host, CPU**), так и графические процессоры-ускорители (GPU), присоединенные к узлам кластера. Стандарт OpenACC был анонсирован в ноябре 2011 г. как совместное детище суперкомпьютерных гигантов CRAY, CAPS и PGI и лидера рынка графических процессоров NVIDIA и как промежуточный стандарт между пользователем и **CUDA** (**Compute Unified Device Architecture**). Компании NVIDIA, CRAY, PGI и CAPS являются членами подкомитета OpenMP по ускорителям и намерены продолжать работу в рамках этой организации для создания **единого стандарта**.

**CUDA** – программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров NVIDIA. Стандарт OpenACC призван значительно упростить работу программиста и создать высокоуровневую прослойку над CUDA.

Графический ускоритель как программная архитектура – это некоторый компьютер, которому **main**-модуль отправляет данные и специальные **kernel**-функции для выполнения. В классическом программировании это аналогия цикла, применяемая к большому числу элементов. Для каждого распараллеливаемого гнезда циклов компилятор генерирует **kernel**-функцию (**_device**-функция, CUDA-ядро), которая не является самостоятельной и может быть запущена только CPU-приложением и выполнена в режиме распараллеливания на GPUs. При запуске **kernel**-функция определяет **device**-устройство и конфигурацию потоков, объединяет потоки в **блоки** и **варпы**.

В варпы объединены подгруппы потоков в блоке, которые исполняют физически одновременно одну и ту же инструкцию для разных данных. Все потоки варпа порождаются конкретной задачей, исполняют одну и ту же инструкцию (SIMT), обращаются к адресам памяти из диапазона в 128 или 256 байт и объединяются после ее выполнения. С помощью SIMT достаточно эффективно выполняется векторизация вычислений на скалярных ядрах. Варп в CUDA – группа из 32 потоков, является минимальным объемом данных, обрабатываемых SIMD-способом в мультипроцессорах CUDA. Вместо работы с варпами напрямую можно работать с **блоками (block)**, содержащими от 64 до 512 потоков.

Наконец, блоки собираются вместе в **сетки (grid)**. Преимущество подобной группировки заключается в том, что число блоков, одновременно обрабатываемых GPU, тесно связано с аппаратными ресурсами. Группировка блоков в сетки позволяет полностью абстрагироваться от этого ограничения и применить **ядро (kernel)** к большему числу потоков за один вызов, не думая о фиксированных ресурсах. Библиотеки CUDA гибко настраивают многомерные варианты **kernel**-ядер (1D, 2D или 3D). Кроме того, подобная модель хорошо масштабируется. Это использует компилятор PGI в технологии OpenACC. При запуске **kernel**-ядро определяют конфигурацию потоков – параметры **block** и **grid**. По умолчанию компилятор в 1D и 2D вариантах на GPU NVIDIA Fermi устанавливает в блоке 256 потоков, т.е. 8 варпов, а в 3D – 64 потока, т.е. 2 варпа. Приведем два примера:

1) 1D – вычисляется массив R(n), n=100000, block: [256], grid: [n/256]=[391];

2) 2D – вычисляется массив R(n,n), n=10000, block: [16;16], grid: [n/16; n/16]=[625; 625].

Если GPU имеет мало ресурсов, то блоки выполняются последовательно. Если количество вычислительных процессоров велико, то блоки могут выполняться параллельно. То есть один и тот же код может работать на GPU как начального уровня, так и на топовых и даже будущих моделях.

Каждый мультипроцессор GPU обладает определенным набором ресурсов. Есть небольшая область памяти под названием **"Общая память/Shared Memory GPU"**, по 16 кбайт на мультипроцессор, к которой потоки одного блока имеют доступ. Это не кэш-память, т.е. CUDA – это комбинация программных и аппаратных технологий. Данная область памяти открывает возможность обмена информацией между потоками *в одном блоке.* Все потоки в блоке гарантированно выполняются одним мультипроцессором.

Общая память – не единственная, к которой могут обращаться мультипроцессоры. Наибольшая эффективность применения графических ускорителей достигается при наличии отдельной видеопамяти (**device memory GPU**) объемом до 6Gb, которая имеет меньшую пропускную способность и большие задержки (**latency**) при выполнении операций обмена.

Данные между **device memory** GPU и **host-memory** передаются явным копированием. Чтобы снизить частоту обращения к этой памяти, NVIDIA оснастила мультипроцессоры кэш-памятью (~ 8 кбайт на мультипроцессор).

На TESLA Fermi K20 в ПГНИУ графические процессоры, подсоединяемые к узлам кластера, имеют по 14 мультипроцессоров, на каждом – 2 SIMD-units по 16 параллельных скалярных ядер (**thread processors**).

Стоит отметить, что до недавнего времени стандарт OpenACC не поддерживался в полной мере ни одним компилятором, но даже то, что уже есть, впечатляет своей простотой и результативностью. Теперь написание программы, выполняемой параллельно на ты-

сячах ядер современных GPU, не требует почти никаких усилий, они практически полностью перекладывается на компилятор. Все, что нужно сделать, – расставить директивы по коду программы на манер OpenMP. Набор директив достаточно велик, но простейшую программу можно сделать быстро, особенно если есть однопоточная реализация.

Отсюда и вытекает основная идея – спрятать от разработчика почти все детали архитектуры GPU, освободить его от тонкостей и оставить время на работу над научным или пользовательским проектом. Директивы в OpenMP и OpenACC по форме записи очень похожи и в языке Fortran начинаются с **!$omp** и **!$acc** соответственно. Следующие затем директивы дополняются одним или несколькими условиями. В табл. 1. приведены наиболее употребляемые директивные блоки для распараллеливания.

Таблица 1. *Наиболее употребляемые директивные блоки для распараллеливания*

| OpenMP | OpenACC |
|---|---|
| **!$omp parallel** <условия-атрибуты данных> | **!$acc parallel** <условия> |
| **!$omp do** <условия атрибуты данных> | **!$acc loop** <условия> |
| <цикл для распараллеливания> | <цикл для распараллеливания> |
| **!$omp end do** | **!$acc end loop** |
| **!$omp end parallel** | **!$acc end parallel** |
| !------------------------------ | !---------------------------------- |
| **!$omp parallel do** <атрибуты данных> | **!$acc data** copy(<список>) |
| <цикл для распараллеливания> | **!$acc kernels** <условия> |
| **!$omp end parallel do** | <цикл для распараллеливания> |
| | **!$acc end kernels** |
| | **!$acc end data** |

## 4. О технологиях OpenMP и OpenACC на компиляторах Intel и PGI

Технология **OpenMP** [8–10] реализована на современных компьютерах с общей (разделяемой) памятью. Программа начинает выполняться как один процесс, который в OpenMP называется *главной нитью* (**master thread**). Этот процесс выполняется последовательно до тех пор, пока не дойдет до первой параллельной конструкции (в простейшем случае – области заключенной парой директив **!$omp parallel** и !$**omp end parallel**). В этот момент создается "бригада"(**team**) нитей, а "бригадиром" для нее является главная нить. Один из основных объектов распараллеливания в OpenMP – цикл **do** (пара директив **!$omp do** и **!$omp end do**). После завершения выполнения распараллеленной конструкции нити бригады синхронизируются, а выполнение программы продолжает только главная нить. В модуле может быть много параллельных конструкций, соответственно, бригады нитей могут образовываться не один раз. OpenMP поддерживает возможность вложения параллельных конструкций.

Атрибуты данных (переменные, массивы) в распараллеливаемом блоке задаются с помощью ключей в директивах. Данные могут быть общими (**shared**) для всех нитей, а могут быть в каждой нити личными (**private**). По умолчанию все данные имеют тип **shared**. Однако это можно изменить, указав ключ **default(private**) или **default(none**).

Счетчики **do**-циклов следует делать личными (private) для каждой нити бригады.

Личные объекты с точки зрения их обработки эквивалентны, как если бы для каждой нити имелась своя копия объекта, а все ссылки на этот объект заменяются ссылками на копию объекта.

Переменные типа private при входе в параллельную конструкцию не определены для любой нити. Если же нужна инициализация из исходного объекта всех личных копий, то используется атрибут **firstprivate**.

В процессе распараллеливания часто возникает необходимость синхронизации, например, запись в одни и те же объекты типа **shared**. В OpenMP предусмотрен ряд директив и функций – это **!$omp barrier**, **!$omp ordered** для упорядоченного выполнения элементов цикла, критические секции, **замки**. Наиболее гибким механизмом синхронизации являются замки. Это можно сделать также с помощью условия **reduction**, которое позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной в reduction переменной инициализируется при входе в параллельную секцию в соответ-

ствии с указанным оператором (0 для оператора +). При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее значение и передается в основной поток.

Например, в итерационном алгоритме решения неклассической задачи МДТТ [1–3] на этапе формирования разрешающей системы уравнений МКЭ для каждой подструктуры (это часть расчетной области с конкретными одинаковыми физико-механическими свойствами) выполняется поэлементная сборка подструктурных матриц, а также подструктурная сборка глобальной матрицы жесткости для всей расчетной области. Эту часть алгоритма можно распараллелить. В табл. 2 приведены фрагменты распараллеливаемой программы с поэлементной сборкой глобальной матрицы (пусть A) с использованием простого замка и условия **reduction в цикле.**

Тесты показали высокую эффективность применения <замка **lock**> в OpenMP.

Приведем пример: собрать из элементарных матриц $G^{(e)}$ глобальную матрицу $G[30000\times 30000]=\sum_{(e)} G^{(e)}[20\times 20],\ (e)=1,50000$.

Вычисление одной элементарной матрицы требует 100 матричных операций умножения. Тип данных **real(16).**

В результате: на компиляторе Intel в однопоточном модуле на сборку нужно ~ 20 мин., а в режиме OpenMP на 12 нитей и простого замка для синхронизации ~ 2 мин.

Применение условия **reduction** в данной версии компилятора более жесткое – это существенное ограничение на размерность глобальной матрицы.

**OpenACC** [11–13] – набор директив для написания гетерогенных программ, задействующих как центральный (host, CPU), так и графические процессоры (GPU), присоединенные к узлам кластера. Чаще всего используются директивы: **parallel, kernels, loop** и **data**.

Таблица 2. *Фрагменты распараллеливания программы*

| OpenMP, замок – переменная **lock** | OpenMP, условие **reduction** |
|---|---|
| ! описанин переменной- простого замка | !**$omp parallel do** <атрибуты данных> |
| **integer (kind=omp_lock_kind) lock** | !**$omp private**(C,B,iel,j,i,ii,jj,val) |
| !---------- | !**$omp shared**(ne,m,n,m1,n1) reduction( :A) |
| ! инициализация простого замка | **do** iel=1,ne ! цикл по элементам |
| **call omp_init_lock(lock)** | ! формирование матрицы B в элементе |
| ! ---------- | **call form_mat_el(m1,iel,B)** |
| !**$omp parallel do** <атрибуты данных> | !---------- |
| !**$omp private**(C,B,iel,j,i,ii,jj,val) | ! выполнение операции A=A+B |
| !**$omp shared**(A,ne,m,n,m1,n1,lock) | **call sum_el(A,B,m,n,m1,n1)** |
| **do** iel=1,ne ! цикл по элементам | **end do** !конец цикла |
| ! формирование матрицы B в элементе | !**$omp end parallel do** |
| **call form_mat_el**(m1,iel,B) | |
| !---------- | **! замечание** – операция A=A+B происходит внутри цикла в полном соответствии с синтаксисом алгоритмического языка, а в случае **reduction** это операция внутри **OpenMP** и зависит от компилятора.<br>Если размеры массива типа **shared** очень большие, то применение замка предпочтительнее. |
| ! нить ждет освобождения замка<br>! затем захватывает простой замок<br>**call omp_set_lock(lock)**<br>! выполнение операции A=A+B<br>**call sum_el(A,B,m,n,m1,n1)**<br>! освобождение простого замка<br>**call omp_unset_lock(lock)**<br>**end do** ! конец цикла<br>!**$omp end parallel do**<br>! перевод замка в не инициализированное сост.<br>**call omp_destroy_lock(lock)** | |

Директива **parallel** указывает на необходимость распараллеливания. Компилятор, проводя анализ кода, определяет необходимость исполнения различных его частей на нитях узла кластера или на GPU, или на host. Директива **kernels** аналог **parallel**, указывает на то, что для каждого нового цикла компилятор создает отдельную **kernel**-функцию. Директива **loop** предшествует оператору цикла и используется для спецификации его свойств. Современные компиляторы не требуют ее явного указания.

Несмотря на всю мощь компилятора PGI, иногда нужно в OpenACC подсказывать с помощью директивы **data** <условия передачи данных>, какие данные необходимо передать с host-процессора на графическое устройство и обратно, а поскольку очень часто время на копирование больше, чем время счета, нужно заранее продумать, **где и как оптимизировать доступ к данным**. Опции-условия передачи данных (массивы или части массивов) –**opy/copyin/copyout/create/present**.

Эти условия можно использовать только с директивами **kernels, parallel и data**. Поясним приведенные условия: **copy** – говорит компилятору скопировать данные на устройство перед выполнением ядра и назад после его завершения; **copyin** – данные на GPU используются только для чтения, и нет необходимости копировать их обратно на host; **copyout** – данные появятся только в результате выполнения ядра на GPU, и никак не зависят от предыдущих значений по этому адресу (т.е. их нужно скопировать на **host** после выполнения **kernel**-функции);

**create** – выделяет в памяти устройства место для данных, не требующих какого-либо копирования (например, массив для хранения промежуточных результатов); **present** – данные уже были переданы на устройство ранее.

В работе [13] M. Wolfe обращает внимание на ограничения, существующие в рассмотренных версиях компилятора PGI. Современное программирование нельзя представить без использования процедур, функций и библиотек. Однако в рассмотренных версиях компилятора OpenACC-программы поддерживали процедуры и библиотеки только в линейной части программы, выполняющейся на host-процессоре, но не в параллельной части на графическом ускорителе. Это ограничение снято в новых версиях компилятора (14.x).

Для этого в OpenACC API введена новая директива **(!acc routine**), в которой следует указать какой тип параллелелизма (**seq, vector, gang, worker**) используется внутри самой процедуры. При этом циклы с **reduction** пока не реализованы. В PGI не реализован тип вещественных данных **real*16**, но это скорее всего связано с техническими возможностями современных GPU. Как и в Intel в новых версиях PGI все массивы в процедурах должны быть динамическими.

## 5. Результаты тестирования технологий OpenMP и OpenACC

Все эксперименты по применению технологий распараллеливания выполнены на суперкомпьютере **TESLA PGU** в ПГНИУ.

На тестовых примерах получены некоторые сравнительные характеристики компиляторов Intel и PGI. Установлено: **на графических ускорителях компилятор PGI** (версия 12.4,13.1) **с языка Fortran не поддерживает встроенные функции и личные процедуры и функции;** не реализованы условие **reduction** и механизм замков для синхронизации потоков, что несколько снижает эффективность применения GPU. В новых версиях кампания-разработчик планирует это сделать.

Поскольку МКЭ, и в частности итерационный алгоритм решения неклассических задач МДТТ [1–3], содержит много матричных операций, приведем примерные затраты времени на выполнение операции умножения в последовательном режиме и с применением технологий распараллеливания.

Тип данных – **double precision**, т.е. **real(8).** Фрагменты программы с применением OpenMP и OpenACC приведены в табл. 3, а затраты времени на умножение полных матриц – в табл. 4.

Таблица 3. *Фрагменты программы с применением OpenMP и OpenACC*

| Intel, OpenMP | PGI, OpenACC |
|---|---|
| !$**omp parallel shared**(a,b,C,m,n,p) | !$**acc data copyin**(A,B) **copyout**(C) |
| !$**omp firstpivate**(A,B | !$**acc kernels** |
| !$**omp do schedule(static) private**(i,j,k,val) | **do** j=1,p |
| **do** j=1,p | **do** i=1,m |
| **do** i=1,m | val=0.0d0 |
| val=0.0d0 | **do** k=1,n |
| **do** k=1,n | val=val+A(i,k)*B(k,j) |
| val=val+A(i,k)*B(k,j) | **end do** |
| **end do** | C(i,j)=val |
| C(i,j)=val | **end do** |
| **end do** | **end do** |
| **end do** | !$**acc end kernels** |
| !$**omp end do** | !$**acc end data** |
| !$**omp end parallel** | – |

Таблица 4. *Примерные затраты времени на умножение полных матриц*

| n-размерность A=B C | Intel, t[сек] | | PGI, t[сек] | |
|---|---|---|---|---|
| | 1поток | OpenMP(12) | 1поток | OpenACC |
| 1000 | 0.54 | 0.07 | 8.75 | 0.06 |
| 2000 | 45 | 1.37 | 80.2 | 0.41 |
| 4000 | 63.7 | 10.5 | 79 6 | 3.14 |
| 8000 | 52 2 | 83.3 | – | 24.9 |

Тестовые примеры показывают, что однопоточные фрагменты программных модулей в **Intel** выполняются быстрее, чем в **PGI**.

Следует помнить, на суперкомпьютерах процесс обмена данными между CPU и мультипроцессорами – это медленная операция (узкое место – **bottleneck, latency**).

Поэтому распараллеливание будет эффективно только тогда, когда время, затрачиваемое каждым потоком на вычисления, будет значительно больше, чем время, необходимое для передачи данных от потока в центральный процессор. Поэтому эффективность применения OpenMP и графических ускорителей растет с увеличением порядка матриц.

Поскольку в МКЭ матрицы являются сильно разреженными, было проведено сравнение матричного умножения полных и разреженных квадратных матриц, а также оценено использование встроенных матричных функций на технологии OpenMP на Intel-компиляторе (табл. 5–6, рис. 1–2). Размерность матриц – 8000.

Сравнение показывает, что использование встроенных функций в расчетах со стандартной двойной точностью эффективнее простого учета разреженности матриц, а для максимально возможной точности наиболее эффективным является учет разреженности матриц. Применение технологии распараллеливания OpenMP особенно эффективно с учетом разреженности для данных типа real(16) и встроенных функций для real(8).

В результате проведенных экспериментов на данном этапе для реализации алгоритма решения неклассических задач был выбран компилятор фирмы Intel, тем более что возможности OpenACC в недалеком будущем кампании-разработчики планируют реализовать в OpenMP.

Данные, приведенные в таблицах, показывают, что использование встроенных функций в расчетах со стандартной двойной точностью эффективнее учета разреженности матриц, а для максимально возможной точности эффективней является учет разреженности матриц. Применения технологии распараллеливания OpenMP эффективно с учетом разреженности для данных типа real(16) и встроенных функций для real(8).

Таблица 5. *Время выполнения операции умножения полных матриц (100 %)*

| Матрицы полные, 100 % | Без учета разреженности матриц / с встроенными функциями | | |
|---|---|---|---|
| | 1поток t[сек] | OpenMP, 8 нитей, t[сек] | OpenMP, 12 нитей, t[сек] |
| real(8) | ~7434 / 415 | ~1095 / 100 | ~900 / 82 |
| Real(16) | ~42268 / 40867 | ~6275 / 6053 | ~4293 / 4108 |

Таблица 6. *Время выполнения операции умножения разреженных матриц (50 %)*

| Матрицы разряженные, 50% | С учетом разреженности матриц/ встроенных функций | | |
|---|---|---|---|
| | 1поток t[сек] | OpenMP, 8 нитей, t[сек] | OpenMP, 12 нитей, t[сек] |
| real(8) | ~7431 / 415 | ~1176 /100 | ~932 / 82 |
| Real(16) | ~13310 /32990 | ~1905 / 5086 | ~1450 / 3492 |

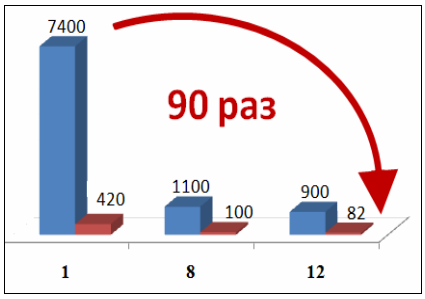


Рис. 1. *Влияние встроенных функций и количества нитей на время выполнения алгоритма (табл. 5) для real(8)*

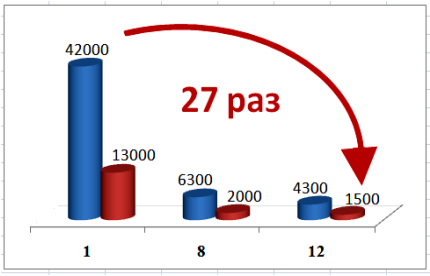


Рис. 2. *Влияние учета разреженности матриц и количества нитей на время выполнения алгоритма (табл. 6) real(16)*

Для применения элементов технологии OpenMP при реализации итерационного численно-аналитического метода неклассических задач МДТТ [1–3] на языке Fortran были написаны несколько процедур, выполняющих матричные операции (умножение и др.), в которых предусмотрено не только распараллеливание циклов, но и учтена возможная разреженность матриц.

Алгоритм итерационного метода в [1–3] можно разделить на 2 части: сборка разрешающей системы уравнений и ее решение.

Процесс сборки также можно разделить на 2 части:

1-я – формирование матриц подструктур из элементных матриц;

2-я – формирование матрицы жесткости подструктуры и ее включение в глобальную матрицу (подробное описание алгоритма и необходимые обозначения даны в работах [1–3].

При формировании матриц подструктур и матриц жесткости алгоритм суммирования матриц построен так, что при распараллеливании не нужно предусматривать синхронизацию процессов, выполняющихся на разных нитях, что дополнительно должно уменьшить время выполнения соответствующих операций.

Численные эксперименты показали, что, так как в МКЭ размерности элементных матриц много меньше размерностей матриц подструктур и тем более глобальной матрицы жесткости, то время выполнения 1-й части сборки в подструктурах будет значительно меньше чем во 2-й части сборки. Поэтому 1-я часть сборки выполняется в последовательном режиме, а вторая часть – для контроля и в параллельном и последовательном режиме.

Приведем результаты экспериментов на следующем простейшем примере: плоская задача, общее число элементов – 1680, число узлов – 5229, глобальное число степеней свободы 10458, 2 подструктуры, число элементов в подструктурах – 787 и 893.

Сравнение приведенных результатов в этом простейшем примере показывает, что для стандартной двойной точности эффект применения OpenMP для 2-й части сборки составляет ~2–2,5 раза, а для максимально возможной точности – ~7,5–8,4 раза (см. табл. 7).

Таблица 7. *Временные затраты на сборку разрешающей системы МКЭ*

| Тип данных | 1-я часть, 2 подструктуры, последовательный режим, t[сек] | 2-я часть, 2 подструктуры | |
|---|---|---|---|
| | | Послед. режим, t[сек] | OpenMP, 12 нитей, t[сек] |
| real(8) | ~0,11+0,13 | ~600+800 | ~245+325 |
| real(16) | ~1,9+2,5 | ~13100+20200 | ~1800+2400 |

В задачах с большим числом элементом эффект распараллеливания во 2-й части сборки будет увеличиваться. Можно также предположить, что применение OpenMP для 1-й части сборки будет целесообразно в случае, когда число конечных элементов будет ~ 100000.

Решение сильно разреженных систем МКЭ выполняют высокоэффективные модули **Ma28** из библиотеки **HSL**, в которых кроме разреженности матриц также предусмотрено распараллеливание некоторых циклов, которое заметно проявляется в модуле с максимально возможной точностью. Координатные портреты разреженных матриц (базовой и на итерациях) записывается в неформатные файлы последовательного доступа.

В табл. 8 приведены временные затраты на выполнение решения системы уравнений порядка 10458 в рассмотренном выше примере (плоская задача) для полной и двух разреженных матриц.

Таблица 8. *Временные затраты на решение системы уравнений*

| Тип данных | Разреженность матрицы (число ненулевых элементов) | | |
|---|---|---|---|
| | 100 %, t[сек] | 49,9 % (базовое решение), t[сек] | 44,8 % (на итерациях), t[сек] |
| Real(8) | ~2524,6 | ~680 | ~680 |
| Real(16) | ~24236,6 | ~6612,4 | ~5614,3 |

Результаты показывают, что при решении задач методами конечных элементов численные методы, реализованные для разреженных матриц, являются очень эффективными.

Численные расчеты по разработанному численно-аналитическому алгоритму [1–3] показали, что применение технологии OpenMP и использование библиотеки HSL позволяет уменьшить машинное время в 4–5 раз.

Расчеты выполнены на МВК "**ПГУ-Тесла**" и "**ПГНИУ–Кеплер**" [29]. Язык программирования – **Fortran**, компиляторы – **Intel, PGI** + технологии **OpenMP, OpenACC.** Тип модуля (однопоточный/ распараллеленный) задают опции в командах запуска компилятора (**ifort** –для Intel, **pgfortran** – для PGI).

Создается скрипт для запуска задания в командной оболочке, т.е. в интерпретаторе команд ОС типа Linux, обычно это **bash (Bourne-again Shell).** Скрипт – содержит необходимые требования задания к ресурсам МВК (память, время, число ядер, нитей и т.д.), атрибуты задания, подключение библиотек и команды для выполнения программных модулей. Команды скрипта соответствуют установленной на МВК платформе управления нагрузкой при распределении ресурсов и системе пакетной обработки заданий. На ПГУ-Тесла это **PBS (Portable Batch System**, команда **#PBS**), на ПГНИУ-Кеплер – **IBM Platform LSF** (команда **#BSUB**). Выполнение задания, т.е. запуск скрипта – это команда **qsub** для ПГУ-Тесла или **bsub** для ПГНИУ-Кеплер с параметром – имя скрипта.

Для запуска задания с применением технологии OpenMP в файле-скрипте нужно задать максимально возможное число процессоров.

## 6. Рекомендации по реализации алгоритмов на языке Fortran и применению технологий OpenMP и OpenACC

- Необходимо выбирать версию компилятора, в которой достаточно эффективно реализованы необходимые библиотеки численных методов и возможности выбранных технологий распараллеливания.
- Для обеспечения необходимой точности лучше всего использовать в алгоритме процедуры нормирования данных и весовые коэффициенты.
- Алгоритм должен быть хорошо структурирован:
  – блочная структура алгоритма;
  – каждый блок в виде процедуры или функции;
  – циклы только в процедурах.
- Для выполнения распараллеливания
  – выделить независимые, последовательные и параллельные блоки;
  – минимизировать узкие места **bottleneck**) передачи данных между блоками;
  – соблюдать баланс (**load balancing**) при использовании быстрой и медленной памяти суперкомпьютера;
  – распараллеливание выполнять в процедурах.
- Максимально использовать возможности современного языка Fortran (векторные операции, сечения массивов, встроенные матричные и векторные функции), это особенно эффективно для массивов стандартной двойной точности в компиляторах корпорации Intel.
- Использовать в алгоритмах блочную, ленточную и разреженную структуру матриц, применять распараллеливание для больших матриц, особенно для массивов типа **real(16)**, и высокоэффективные численные методы и библиотеки для операций с матрицами такого типа.

## Список литературы


1. *Пестренин В.М., Пестренина И.В., Ландик Л.В.* Нестандартные задачи для однородных элементов конструкций с особенностями в виде клиньев в условиях плоской задачи // Вестник Томского государственного университета. Математика и механика. Томск, ТГУ. 2014. Т. 1(27). С. 95–109.
2. *Пестренин В.М., Пестренина И.В., Ландик Л.В.* Итерационный конечно-элементный алгоритм исследования напряженного состояния элементов конструкций с особыми точками и его реализация // Вестник Пермского национального исследовательского политехнического университета. Механика. 2015. № 4. С. 171–187. DOI:10.15593/perm.mech/2015.4.11.
3. *Пестренин В.М., Пестренина И.В., Ландик Л.В.* Исследование напряженного состояния в составной пластинке вблизи края линии соединения в зависимости от толщины и материальных параметров соединяющей прослойки // Вестник ПНИПУ. Механика. 2014. Т. 1. С. 153–166.
4. *Аптуков В.Н., Ландик Л.В., Скачков А.П.* Технологии использования современных пакетов прикладных программ при решении задач механики сплошных сред: учеб. пособие. Пермь, ПГУ, 2007. 153 с.
5. *Горелик А.М.* Современный фортран для компьютеров традиционной архитектуры и для параллельных вычислений // Вычислительные методы и программирование. М.: Изд-во МГУ, 2004. Т. 5(3). С. 1–12 с.
6. *Горелик А.М.* Эволюция языка программирования Фортран (1957-2007) и перспективы развития // Вычислительные методы и программирование. М.: Изд-во МГУ, 2008. Т. 9(2). С. 53–71.
7. *Форсайт Д.Ж., Малькольм М., Моулер К.* Машинные методы математических вычислений. М.: Мир, 1980. 280 с.

8. *Антонов А.С.* Параллельное программирование с использованием технологии OpenMP: учеб. пособие. М.: Изд-во МГУ, 2009. 78 с.
9. *Антонов А.С.* Технологии параллельного программирования MPI и OpenMP: учеб. пособие. М.: Изд-во Моск. ун-та, 2012. 339 с.
10. *OpenMP* Architecture Board. URL: //http://www/openmp/org (дата обращения: 04.09.2018).
11. *Wolfe M.* OpenACC Features in PGI Accelerator FORTRAN Compilers. Part 1 // PGInsider. Technical News from PGI, 2012/ URL: //https://www.pgroup. com/lit/articles/-insider/v4n1a1a.htm (дата обращения: 04.09.2018).
12. *Wolfe M.* The OpenACC applications programming interface. Version 2.0. // PGInsider. Technical News from PGI, 2013. URL //.http://www.openfcc.org /sites/files/OpenACC% 202%200.pdf (дата обращения: 04.09.2018).
13. *Wolfe M.* Using the OpenACC Routine Directive // PGInsider. Technical News from PGI, 2014. URL: https://www.pgroup. com/lit/articles/ insider/ v6n1a1.htm (дата обращения: 04.09.2018).
14. *NVIDIA's* Next Generation CUDA Compute Architecture: Fermi // NVIDIA Corporation, 2009.
15. *Кучумова Е.В., Садовский В.М.* Численное исследование распространения сейсмических волн в блочных средах на многопроцессорных вычислительных системах // Вычислительные методы и программирование. М.: Изд-во МГУ, 2008. Т. 9(1). С. 66–76.
16. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
17. *Ортега Дж.* Введение в параллельные и векторные методы решения линейных систем. М.: Мир, 1991. 367 с.
18. *Рычков А.Д.* Численные методы и параллельные вычисления: учеб. пособие. Новосибирск: СибГУТИ, 2007. 144 с.
19. *Паасонен В.И.* Параллельный алгоритм для компактных схем в неоднородных областях // Вычислительные технологии. 2003. Т. 8(3). С. 98–106.
20. *Арыков С.Б., Малышкин В.Е.* Система асинхронного параллельного программирования "АСПЕКТ" // Вычислительные методы и программирование. М.: Изд-во МГУ, 2008. Т. 9 (2). С. 48–52.
21. *Левин М.П.* Параллельное программирование с использованием OpenMP. М.: Бином, 2008. 120 с.
22. *Теплов А.М.* Об одном подходе к сравнению масштабируемости параллельных программ // Вычислительные методы и программирование. М.: Изд-во МГУ, 2014. Т. 15. С. 697–711.
23. *Сандерс Дж., Кэндрот Э.* Технология CUDA в примерах: введение в программирование графических процессоров. М.: ДМК Пресс, 2011. 232 с.
24. *Горобец А.В., Муков С.А., Железняков А.О., Богданов П.Б., Четверушкин Б.Н.* Применение GPU в рамках гибридного двухуровневого распараллеливания MPI+OpenMP на гетерогенных вычислительных системах // Параллельные вычислительные технологии. Челябинск: Издательский центр ЮурГУ, 2011. С. 452–460.
25. *Волков К.Н., Емельянов В, Карпенко А.Г, Курова И.В., Серов А.Е., Смирнов П.Г.* Численное решение задач гидродинамики на графических процессорах общего А.Е назначения // Вычислительные методы и программирование. М.: Изд-во МГУ, 2013. Т. 14(2). С. 82–90.
26. *Богданов П.Б., Ефремов А.А., Горобец А.В., Суков С.А.* Применение планировщика для эффективного обмена данными на суперкомпьютерах гибридной архитектуры с массивно-параллельными ускорителями // Вычислительные методы и программирование. М.: Изд-во МГУ, 2013. Т.14(2). С. 122–134.
27. *Галимов М.Р., Биряльцев Е.В.* Некоторые технологические аспекты применения высокопроизводительных вычислений на графических процессорах в прикладных программных системах // Вычислительные методы и программирование. М.: Изд-во МГУ, 2010. Т. 11(2). С.77–93.
28. *Губайдуллин Д.А., Никифоров А.И., Садовников Р.В.* Об особенностях использования архитектуры гетерогенного кластера для решения задач механики сплошных сред // Вычислительные методы и программирование. М.: Изд-во МГУ, 2011. Т. 12 (2). С. 450–460.
29. *Деменев А.Г.* Параллельные компьютерные технологии и высокопроизводительные вычисления как инновационное образовательное направление в Пермском государственном университете / В сб.: На-

учный сервис в сети Интернет: масштабируемость, параллельность, эффективность: тр. Всерос. суперкомпьютерной конференции. Суперкомпьютерный консорциум университетов России, 2009. С. 434–437.

30. *Деменев А.Г.* Программирование для параллельных вычислительных систем: учеб.-метод. пособие. А.Г. Деменев. Федеральное агентство по образованию. ГОУ ВПО "Пермский гос. ун-т". Пермь, 2007. 128 с.

# OpenMP and OpenACC Programming technologies on supercomputers and modern FORTRAN in solid mechanics problems

**L. V. Landik, I. V. Pestrenina, V. M. Pestrenin**
Perm State University; 15, Bukireva st., Perm, 614990, Russia
Ipestrenina@gmail.com; 8 (342) 2 396 375

The paper considers the features of parallelization of algorithms using OpenMP and OpenACC programming technologies. It covers the features that were manifested when implementing the iterative finite element algorithm for solving non-classical problems of mechanics of solids with singular points. A review of modern means of creation of high-performance mathematical software is provided. A comparative study of the parallelization of algorithms using OpenMP and OpenACC on INTEL and PGI compilers is conducted. Time characteristics of the compilers and parallelization technologies are given on the examples of multiplication operations for complete and sparse matrices. The influence of modern Fortran built-in functions on these characteristics and the representation of real numbers is also illustrated. The advantages of numerical methods for sparse matrices are shown. Practical recommendations for the application of Fortran algorithms as well as OpenMP and OpenACC technologies are given.

**Keywords**: *parallelization of algorithms; modern Fortran; finite element method; OpenMP; OpenACC.*