ЗАДАНИЕ К ПРАКТИЧЕСКОЙ РАБОТЕ №3 «OpenMP» ПО КУРСУ ОПП

Выбираете один из двух нижеприведенных вариантов задания по своему усмотрению.

ВАРИАНТ 1

1. Последовательную программу из предыдущей практической работы, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP.
   ОБЯЗАТЕЛЬНОЕ УСЛОВИЕ: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.
2. Замерить время работы программы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

ВАРИАНТ 2

1. Вторую программу из практической работы по ЭВМиПУ на вычисление обратной матрицы (собственная векторизация) распараллелить при помощи OpenMP. Как минимум, должны быть распараллелены операции умножения матриц, сложения матриц, умножения матрицы на скаляр, суммирования модулей элементов матрицы по строкам и столбцам. ОБЯЗАТЕЛЬНОЕ УСЛОВИЕ: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.
2. Замерить время работы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках:
   - простой программы (без векторизации, без OpenMP)
   - векторизованной программы
   - комбинированной (векторизация+OpenMP) программы.

   Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

КОММЕНТАРИЙ К ПРАКТИЧЕСКОЙ РАБОТЕ №3 «OpenMP» ПО КУРСУ ОПП

ПРО MPI И OPENMP

Данная практическая работа направлена на знакомство со следующей технологией параллельного программирования – OpenMP. Как и MPI, OpenMP

является стандартом. Но только реализуется этот стандарт не в виде отдельной библиотеки, подключаемой к вашей программе на этапе компиляции, а в самом компиляторе. То есть OpenMP-директивы, которые Вы напишите в сишный код, обязаны восприниматься любым компилятором, поддерживающим эту технологию. Ну и естественно, что все популярные широко используемые компиляторы языков C/C++ и Fortran эту технологию поддерживают.

О том как использовать OpenMP на кластере рассказано здесь:

http://nusc.nsu.ru/wiki/doku.php/doc/openmp/openmp

Добавлю только, что в случае gcc поддержка OpenMP включается при помощи ключа «-fopenmp», а в случае интеловского icc или icpc – при помощи «-openmp». То есть:

```
g++ -fopenmp omp_hello.cpp -o g++_omp_hello.out
icpc -openmp omp_hello.cpp -o icpc_omp_hello.out
```

MPI-программа ориентирована на запуск множества **процессов**. Процесс – самостоятельная программная единица. Часть процессов можно запустить на одной машине под одной операционной системой, а другую часть – на другой (именно это вы и делали в первой лабе). В отличие от MPI, технология OpenMP ориентирована на запуск множества **потоков** (нитей, тредов) в пределах одного процесса. И поскольку процесс со всеми своими потоками должен размещаться в пределах одного экземпляра операционной системы, установленного на одной машине, то OpenMP-программу нельзя запустить на нескольких машинах, входящих в кластер.

Может возникнуть вопрос: а для чего тогда нужен OpenMP, если он накладывает такие жесткие ограничения на масштабируемость? В чем его плюсы перед MPI? Ответ: основной плюс – скорость разработки. Распараллелить программу с помощью OpenMP на порядок проще, чем с помощью MPI. Для того, чтобы получить существенное ускорение иногда достаточно просто поставить директиву распараллеливания перед основным циклом программы и несколько вспомогательных директив. Кроме того, поток является существенно более легкой единицей, чем процесс. Для порождения и последующего содержания потока нужно меньше оперативной памяти, времени и др. В связи с этим программа на OpenMP в *n* потоков зачастую оказывается быстрее, чем эквивалентная ей MPI-программа в *n* процессов, запущенная на одном узле.

Схожесть MPI и OpenMP в использовании программной модели SPMD (Single Program Multiple Data). То есть вы пишите одну программу, код которой выполняют все процессы (в MPI) или все потоки (в OpenMP). Конечно вы можете выбирать исполнителя для какого-то участка при помощи, например, конструкции «*if (myrank == P) {...}*», но это частности. К слову это будет являться отличием с технологией POSIX Threads (Pthreads), где для каждой нити задается свой собственный код (с Pthreads мы познакомимся в заключительной лабе).

Если продолжать сравнивать MPI и OpenMP, то специфичные для каждой технологии **проблемы также различны**.

Для MPI – проблема грамотной организации коммуникаций между процессами:

- На каждый (I,S,B,R,Is,Ib,Ir)Send обязательно должен быть соответствующий (I)Recv.
- Коллективная коммуникация должна быть вызвана НА ВСЕХ ПРОЦЕССАХ коммуникатора.
- Служебные параметры (тэги, типы данных, количество передаваемых/принимаемых элементов) при коммуникации на всех процессах должны друг другу соответствовать.
- ...

В этом списке только обязательные условия для того, чтобы MPI-программа в принципе работала корректно. Но этого недостаточно. Программа должна еще быть эффективной. Например, если процесс-приемник вызовет Recv и будет долго-долго ждать данных от процесса-отправителя, который вызовет Send только перемолотив большой пласт работы, то это очень негативно скажется на эффективности.

В случае OpenMP одной из основных проблем является синхронизация доступа к объектам, общим для потоков. Потоки отличаются от процессов в частности тем, что делят общую память, принадлежащую всему процессу. Конечно можно объявить переменную как private и таких проблем с ней не будет, но, к сожалению, от shared-объектов редко удается избавиться... Если у вас shared-объект изменяется несколькими нитями, то вам сразу нужно думать о том, как обезопасить доступ к нему.

Еще одним отличием является то, что в OpenMP мы легко на протяжении программы можем создавать и уничтожать секции из такого количества потоков, из какого захотим. Например, может быть так:

Последовательная секция *//один поток*

Параллельная секция на 4 потока

Последовательная секция

Параллельная секция на 2 потока

Последовательная секция

Для этого служит параметр «num_threads» при создании параллельной секции:

```
#pragma omp parallel num_threads(2)
```

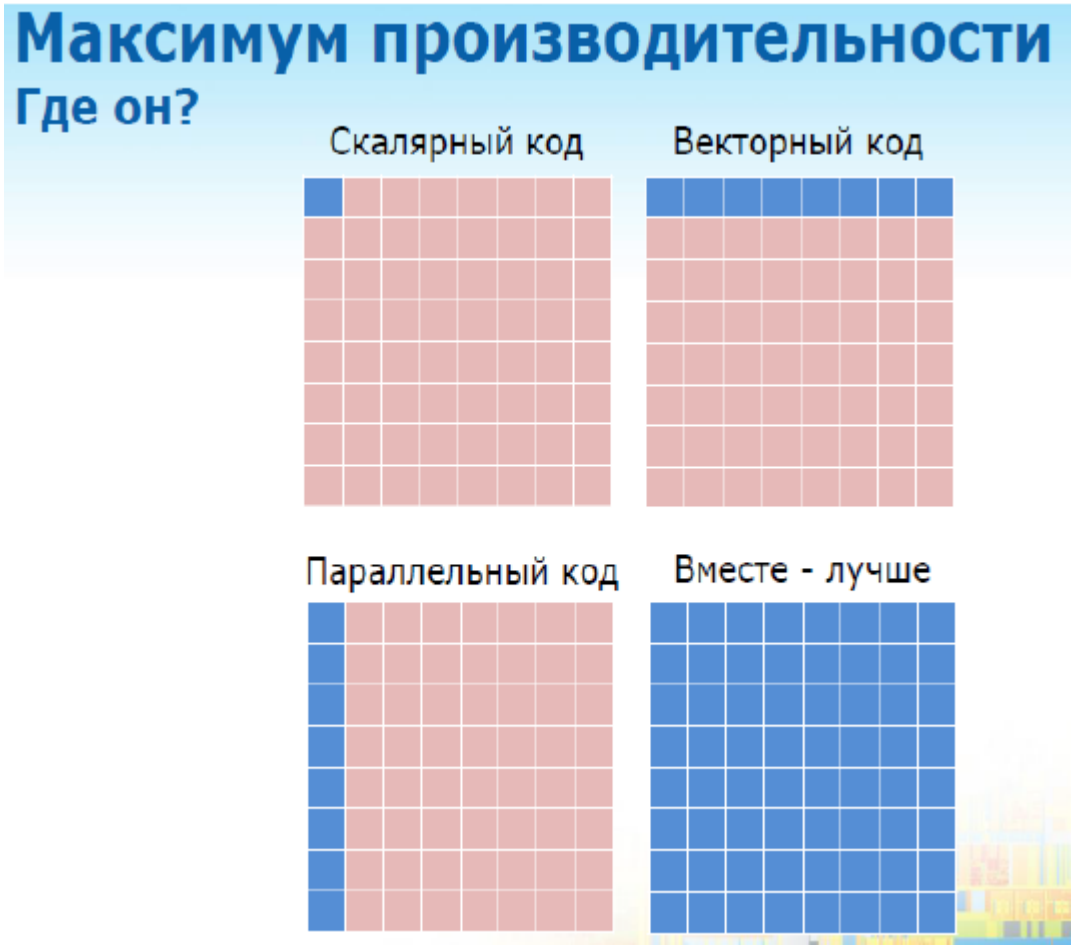
ПРО ПЕРВЫЙ ВАРИАНТ ПРАКТИЧЕСКОЙ

Каждый делает ровно тот итерационный метод, который был у него в предыдущей практике. Там вы уже должны были подготовить последовательную программу, а в рамках этой практики ее нужно только распараллелить OpenMP-директивами и построить графики ускорения и эффективности для сравнения обозначенных двух подходов.

ПРО ВТОРОЙ ВАРИАНТ ПРАКТИЧЕСКОЙ

Я хочу предложить вам еще один альтернативный вариант задания для знакомства с OpenMP. Для этого нужно вытащить из небытия код практики по ЭВМ и ПУ про векторизацию. Там решалась задача вычисления обратной матрицы итерационным методом. Писали 3 программы: 1) простую последовательную, 2) с помощью собственной векторизации, 3) с помощью BLAS. Вам нужно поработать со ВТОРОЙ.

Векторное расширение и векторные регистры – это то, что дает параллелизм в пределах одного процессорного ядра. Но мы можем загрузить работой несколько ядер. Для этого существуют технологии многопоточной обработки, к которым относится OpenMP. Совмещение этих 2-х подходов может дать наибольшую производительность. Схематично это отражено на картинке из презентации, которую я рассылал к этой практике в прошлом семестре:



Успешной работы!