**ЗАДАНИЕ К ПРАКТИЧЕСКОЙ РАБОТЕ** **№1 ПО КУРСУ ОПП**

1. Написать 3 программы (одну последовательную и 2 параллельные) на языке C или C++, которые реализуют итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A– матрица размером N×N, x и b – векторы длины N. Тип элементов –double.

2. 2 параллельные программы реализовать с помощью MPI с разрезанием матрицы A по строкам (в первой программе) или по столбцам (во второй) на близкие по размеру, возможно не одинаковые, части.   
Программа 1: векторы x и b дублируются в каждом MPI-процессе,  
Программа 2: векторы x и b разрезаются между MPI-процессами.  
Уделить внимание тому, чтобы при запуске программы на различном  
числе MPI-процессов решалась одна и та же задача (исходные данные  
заполнялись одинаковым образом).

3. Замерить время работы последовательного варианта программы и 2-х параллельных при использовании различного числа процессорных ядер. Минимально на 2, 4, 8, 16, 24 (на каждом из наших узлов кластера по 12 ядер), но чем на большем количестве процессов будут выполнены замеры, тем лучше. Также чем больше замеров будет выполнено на одном и том же количестве процессов, тем лучше. В этом случае для построения графиков следует брать минимальное время.  
Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

4. Выполнить профилирование двух вариантов программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и ядер.

5. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программ

**КОММЕНТАРИЙ К ПРАКТИЧЕСКОЙ РАБОТЕ №1 ПО КУРСУ ОПП**

**Про «идеологию высокопроизводительных вычислений»**

Во время написания своих параллельных программ в этом курсе вам нужно «вжиться в шкуру» прикладного исследователя-вычислителя. Его задача состоит не в том, чтобы распараллелить, а в том, чтобы постараться максимально сократить время работы своей большой программы. И распараллеливание – это только один из инструментов ускорения программы. То есть вы должны стараться избавляться от всех лишних вычислений, стараться экономить время работы на всем, на чем можно, распараллеливать не только самые тяжелые операции (например, умножение матрицы на вектор), а применять параллелизм везде, где это разумно. Например, для сложения/вычитания/умножения на скаляр, вычисление модулей больших векторов уместно использовать распараллеливание. Конечно нужно помнить и о тех моментах, о которых мы говорили в предыдущем семестре. Например, для того, чтобы эффективно использовать предвыборку данных в кэш, матрицу нужно стараться обходить по строкам, а не по столбцам и т.д.

То есть эффективность с точки зрения сокращения времени должна быть вашей целью.

**Про работу на кластере НГУ**

Головная страница: <http://nusc.nsu.ru/wiki/doku.php>

Обязательно почитайте раздел «Правила работы»

Запуск тяжелых вычислительных программ ТОЛЬКО через очередь: http://nusc.nsu.ru/wiki/doku.php/doc/pbs/pbs

Работа с MPI: http://nusc.nsu.ru/wiki/doku.php/doc/mpi/mpi

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

**Про профилирование**

Широкую известность приобрели 2 инструмента профилирования MPI-программ: бесплатный Jumpshot и коммерческий Intel Trace Analyzer and Collector (ITAC).

Здесь расскажу про ITAC, потому как это средство обладает бОльшим функционалом и возможностями для анализа. Естественно, что лучше выбирать интеловскую реализацию MPI. Для того, чтобы воспользоваться ITAC нужно в файл ~/.bashrc вписать 2 строки, загружающие необходимые переменные окружения:

source /opt/intel/composer\_xe\_2015.2.164/bin/iccvars.sh intel64

source /opt/intel/itac/8.1.3.037/bin/itacvars.sh

Вписав эти 2 строки и переоткрыв putty-терминал, можно делать так:

$mpiicpc mympi.cpp –o mympi.out *//компиляция MPI-программы. 2 буквы «i» в слове mpiicc не случайны*

В файле описания задачи для qsub:

$mpirun -trace -machinefile $PBS\_NODEFILE -np $MPI\_NP -perhost 2 ./mympi.out

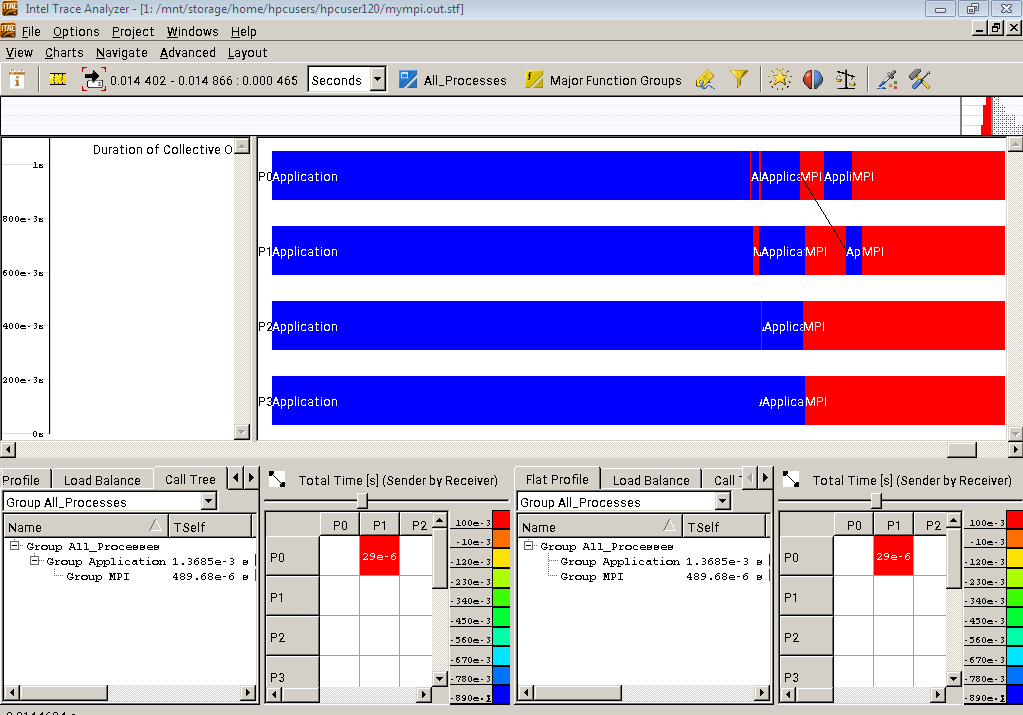
*Примечание*: опция «perhost» указывает по скольку процессов подряд размещать на каждом узле. То есть в случае «-perhost 2» если у нас, к примеру, 3 узла и 7 процессов, на первом узле будут процесс 0 и 1, на втором узле — 2 и 3, на третьем — 4 и 5, потом 6-й процесс снова будет размещен на 1-м узле.

Во время работы программы формируется файл трассы (mympi.out.stf). Этот файл потом можно посмотреть графической утилитой traceanalyzer. Но как мы знаем, putty предоставляет нам консольный интерфейс и для графических программ мы должны использовать какой-либо X-сервер, например Xming. Как пользоваться Xming-ом я писал вам в комментарии к практической по ЭВМ и ПУ про библиотеку OpenCV.

На кластере запускаете строку:

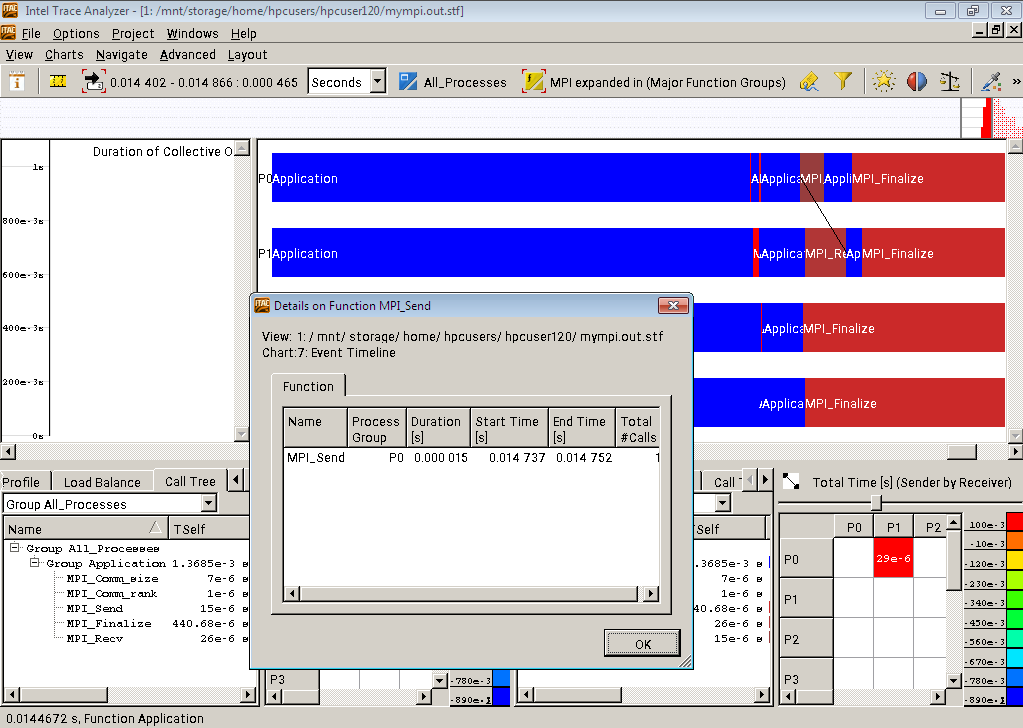
$traceanalyzer mympi.out.stf

После чего в окне xming покажется следующая картинка:



Traceanalyzer может показать множество разных представлений, но вам достаточно продемонстрировать Event Timeline (Charts -> Event Timeline).

Для того, чтобы на таймлайне отображалось не просто слово «MPI» на MPI-функциях, а наименования конкретных операций, нужно кликнуть по одной из них правой кнопкой и выбрать «Ungroup». Тогда картина будет примерно следующей:



Сделать профилирование – это не просто предоставить картинку, а еще провести по ней анализ – на что и почему ушло много или мало времени. А также мочь ответить на вопросы.

**Некоторые замечания по практической №1**

1) Матрицу можно генерировать как угодно: читать одним процессом из файла, а потом разрезать ее между остальными, генерировать на каждом процессе свой кусок, генерировать всю матрицу на одном процессе, а потом разрезать ее между остальными. Не стоит только читать из файла всем процессам и НЕЛЬЗЯ раздавать матрицу полностью от одного процесса всем (например при помощи MPI\_Bcast).  
Рекомендую взять симметричную матрицу с утяжеленной главной диагональю. Если элементы матрицы будете генерировать в отрезке [-100; 100], то утяжелять можно, к примеру, прибалением 200.

2) Обязательно при замерах времени запускать последовательную программу на кластере и ее брать за T1 при расчетах ускорения и эффективности.

3) Мерить времена нужно строго на одном железе. То есть и последовательную и параллельные программы с разным количеством процессов – на кластере. И даже лучше всё в одном задании, чтобы точно оно отработало на одних и тех же узлах. То есть в одном задании - последовательная и 2 параллельные программы на 2,4, 8, 16 и, опционально – 24 ядрах.

4) В MPI-программе время следует замерять при помощи функции MPI\_Wtime между «засечками», первую из которых необходимо установить СРАЗУ после вызова MPI\_Init, а вторую — после получения итогового вектора решения на одном процессе (имеется ввиду уже собранного полностью, а не какой-либо части). То есть после второй «засечки» может быть только проверка корректности полученного вектора решения.

5) В основном цикле программы включите счетчик итераций. Если процесс сходится за 1-2 итерации (особенно это характерно для метода сопряженных градиентов), то это абсолютно непоказательно. Ухудшите начальные условия (матрицу, вектор b или сделайте начальное приближение подальше от решения). Этот же счетчик итераций поможет определить вам, что итерационный процесс расходится (приближения не стремятся к решению). В основном цикле программы вы можете наряду с основным условием выхода (отношение определенных норм меньше эпсилон) поставить еще второе — счетчик итераций например больше 50000, поскольку за такое количество итераций ваша задача должна уже сойтись. Если не поставить это условие, а ваш процесс будет расходиться (из-за неудачно выбранного начального приближения или из-за ошибки в программе), то программа будет работать и бессмысленно тратить ресурсы машины до тех пор, пока не «свалится» или ее кто-нибудь не завершит принудительно. ОСОБЕННО это плохо, когда вы запускаете такую программу на кафедральном сервере. На кластере в каждой задаче есть предельное время, после завершения которого она принудительно снимается с выполнения,а на кафедральном сервере нет системы пакетной обработки.

6) В методе простой итерации фигурирует параметр тау, который остается постоянным в течение всей работы программы. В описании практической работы сказано, что если процесс начал расходиться, то нужно поменять этот параметр на противоположный. НО ЭТО НУЖНО ДЕЛАТЬ НЕ «НА ХОДУ», а остановив программу, поменяв знак в исходном коде и запустив программу заново.

7) Для информации — расширения AVX на кластере нет, поэтому если хотите быть большими молодцами и внедрить векторизацию, то пользуйтесь только SSE. Хотя еще раз подчеркну, что я этого не требую.

**Про 2 варианта параллельной программы**

При выполнении лабораторной №1 необходимо написать 3 программы: одну последовательную и 2 параллельных. Последовательную программу вы пишите без MPI, используя тот функционал языка C, который хорошо знаете с 1 семестра 1 курса. Параллельные программы, согласно задания, должны различаться тем, что в одном случае вектором *b* обладает каждый процесс, а во втором случае векторы *b* и *x* разрезаны между всеми процессами.

Естественен вопрос: чем принципиально отличается первый параллельный вариант от второго? Ведь во втором варианте вначале работы программы мы можем собрать весь вектор *b* со всех процессов и раздать полученный целый вектор всем процессам (кстати – как это сделать одной MPI-функцией?). А потом запустить тот же самый итерационный механизм, который запускали в первом варианте. Но такая схема работы второй параллельной программы меня не интересует.

Есть несколько способов решения задачи при разрезании векторов *b* и *x* между процессами. Здесь я предложу только один. Предлагаю разрезать матрицу *A* не по строкам, а по столбцам. В каждом варианте нужно вычислять разность (*Axn – b).* Если приглядитесь – в методе сопряженных градиентов эта разность с противоположным знаком. Давайте разберем как можно ее посчитать, если матрица *A* разрезана по столбцам, а векторы *x* и *b* распределены между процессами. При этом процессу с номером *i* принадлежат следующие столбцы матрицы и элементы векторов *x* и *b*: *i, n+i, 2\*n+i*…, где *n* – число процессов и их нумерация идет с единицы (можете реализовать вариант, когда каждому процессу принадлежат элементы и столбцы с *n\*i* по *n\*(i+1)*).

Предположим, что у нас порядок матрицы 6 и 3 процесса. Умножение p=*Ax* (для простоты не буду писать нижний индекс) можно выполнить так:

Процессу №1 (еще раз повторю – предположим, что нумерация процессов идет с единицы) принадлежат столбцы 1 и 4 и соответствующие элементы вектора *x*. Он производит умножение своих элементов *x* на свои столбцы *A*, после чего складывает получившиеся столбцы. Аналогичным образом поступает каждый процесс.

После этого у каждого процесса получается по одному столбцу. Для получения итогового произведения *p* эти столбцы необходимо сложить. Тут необходимо использовать функционал MPI (какой MPI-функцией можно со всех процессов сложить столбцы и результат сохранить у каждого?).

Затем необходимо вычислить разницу полученного вектора *p* и вектора *b*. Но вектор *b* распределен аналогично вектору *x*. Значит нужно раздать полученный вектор *p* по процессам и вычесть соответствующие компоненты *b*. То есть наш процесс №1 в итоге посчитает:

*p1-b1*

*p4-b4*

Как совершить другие арифметические операции – догадаетесь сами :)