Министерство образования Российской Федерации

Санкт-Петербургский государственный

архитектурно-строительный университет

Кафедра прикладной математики и информатики

Дипломная работа

АЛГОРИТМЫ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ ПРИ ИССЛЕДОВАНИИ УСТОЙЧИВОСТИ ПОДКРЕПЛЕННЫХ ПОЛОГИХ ОБОЛОЧЕК

Выполнил студент

группы ПМ-V

Сизов А.С.

Дипломный руководитель

доктор технических наук

проф. Карпов В.В.

Санкт-Петербург 2010

Оглавление

[Введение](#_Toc277374088)

[Глава 1. Математические модели деформирования подкрепленных пологих оболочек при учете различных свойств материала](#_Toc277374089)

[Глава 2. Традиционные алгоритмы решения задач устойчивости для подкрепленных пологих оболочек](#_Toc277374090)

[2.1 Программа PologObolochka](#_Toc277374091)

[Глава 3. Распараллеливание процесса вычисления. Основы, принципы, практическое применение](#_Toc277374092)

[3.1 Message Passing Interface](#_Toc277374093)

[3.2 MPICH](#_Toc277374094)

[3.3 Принципы работы MPICH](#_Toc277374095)

[3.4 Установка MPICH в Windows](#_Toc277374096)

[3.5 Настройка MPICH](#_Toc277374097)

[3.6 Создание общего сетевого ресурса](#_Toc277374098)

[3.7 Запуск MPI-программ](#_Toc277374099)

[Глава 4. Алгоритмы решения задач устойчивости для подкрепленных пологих оболочек, основанные на распараллеливании процесса вычисления](#_Toc277374100)

[4.1 Программа и результаты](#_Toc277374101)

[Заключение](#_Toc277374102)

[Литература](#_Toc277374103)

[Приложения](#_Toc277374104)

## Введение

Работа выполнена в соответствии с грантом Минобнауки РФ "Развитие научного потенциала высшей школы (2009-2010 г. г)", тема №2.1 2/6146. Разрабатывается программный комплекс расчетов прочности и устойчивости подкрепленных оболочек вращения с учетом различных свойств материала. Используются наиболее точные модели деформирования оболочек. Усложнение расчетных уравнений приводит к существенному увеличению времени расчета одного варианта задачи на ЭВМ при последовательных вычислениях. Чем тоньше оболочка, тем больше изменяемость формы поверхности оболочки при деформировании. Это приводит к увеличению числа членов разложения искомых функций в ряды в методе Ритца, чтобы точность расчетов была высока. Так, при удержании 9 членов в разложении искомых функций (N = 9) время расчета одного варианта, в зависимости от кривизны оболочки и числа подкрепляющих ее ребер, составляет 1-3 часа, при N = 64 - несколько суток. Для существенного сокращения времени расчета одного варианта задачи на ЭВМ требуется оптимизация программы. Один из путей решения данной проблемы состоит в распараллеливании процессов вычисления.

Расширение возможностей в конструировании вычислительной техники всегда оказывало влияние на развитие вычислительной математики - в первую очередь численных методов и численного программного обеспечения. В условиях появления больших параллельных систем и создания сверхмощных новых систем перед математиками, и в особенности математиками-прикладниками, открывается обширнейшая область исследований, связанная с совместным изучением параллельных структур численных методов и вычислительных систем. [5]

## Глава 1. Математические модели деформирования подкрепленных пологих оболочек при учете различных свойств материала

Оболочки покрытия и перекрытия строительных сооружений не могут допускать прогибы, соизмеримые с их толщиной, поэтому можно вести их расчет в геометрически линейной постановке, при этом существенно упрощаются все соотношения. В этом случае функционал полной энергии деформации ребристой оболочки при действии на нее поперечной нагрузки  будет иметь вид

 (1)

где , если учитывается физическая нелинейность, , если учитывается ползучесть материала.

Для металлов ползучесть может развиваться только при больших температурах, поэтому будем считать . Для оргстекла зависимость "" практически линейная, поэтому можно принять .

Здесь







 (2)





 (3)









 (4)

или, если принимается итерационный процесс по временной координате,









 (5)

где 

 (6)

Срединная поверхность пологой оболочки прямоугольного плана постоянной толщины образована перемещением пологой дуги окружности радиуса  вдоль пологой дуги окружности радиуса .

Поверхность такого вида называют поверхностью переноса. Оболочка называется пологой, если отношение стрелы подъема  оболочки к наименьшему линейному размеру  удовлетворяет соотношению  (рис.1). Так как стрела подъема пологой оболочки мала , то геометрия пологой оболочки близка к геометрии пластины, поэтому параметры Ляме приближенно принимаются равными единице: 

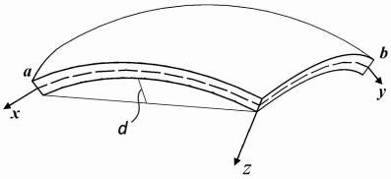


Рис.1. Пологая оболочка двоякой кривизны

Для пологих оболочек считается, что при деформации образуются сравнительно мелкие вмятины, размеры которых малы по сравнению с радиусами кривизны оболочки, и также считается, что функция прогиба является быстро изменяющейся функцией, т.е. отношение *W* к ее первой производной является малой величиной.

Геометрические соотношения для пологих оболочек при учете поперечных сдвигов принимают вид



 (7)



Кроме того,  имеют вид (2.7),





Функции изменения кривизн  и кручения  принимают вид

  (8)

Выражения для  здесь принимают вид

  (9)

## Глава 2. Традиционные алгоритмы решения задач устойчивости для подкрепленных пологих оболочек

После применения метода Ритца к функционалу (1) при аппроксимации функций перемещений в виде

, ,

, (10)

получим систему интегро-алгебраических уравнений

 (11)

для определения неизвестных числовых параметров , , .

Следует обратить внимание на то, что для конических, сферических и торообразных оболочек аппроксимирующие функции по переменной  должны иметь и симметричные, и несимметричные составляющие (если это синусы, то должны быть  и т.д.).

Систему (11) распишем подробно, отдельно вычислив



В начале вычислим , учитывая, что для оболочек вращения , (сомножитель  опускаем)









 (12)





 (13)











 (14)

Для решения системы уравнений предполагается использовать метод упругих решений А.А. Ильюшина [9], т.е. метод итераций, когда на каждой итерации решается линейно-упругая задача с изменяющейся правой частью ()



 (15)



где  равны  или . Здесь

, , ,

, , .

В выражениях, стоящих в левой части системы (15), пренебрегается сомножителем , поэтому его не будет и в  и .

Так как  и  будут вычислены при известных , , , то расписывать эти выражения через , ,  нет смысла. Правые части системы (15) играют роль фиктивной нагрузки.

При вычислении  примем

, (16)

где

.

Для металла, не имеющего площадки текучести,  принимает значение от  до и вычисляется эмпирически, для железобетона

.

Аппроксимация (16) справедлива при малой нелинейности.

Выражение для  представим в виде

,

где

 (17)

Так как

,

то



 (18)



где

 , ,

, .

Теперь вычислим  (опуская сомножитель

)







 (19)



 (20)









 (21)

Систему (15) кратко можно записать в виде

 (22)

где  равняется  или

; 

 - левые части системы (15);



При решении физически-нелинейной задачи для каждого значения параметра нагрузки решается итерационная задача

 (23), до тех пор, пока

.

Начальное приближение  находится из решения линейно-упругой задачи

 (24)

Метод упругих решений - самый простой и распространенный метод решения нелинейно упругих задач. В работе [15] к уравнениям равновесия применялся метод последовательных нагружений при исследовании напряженно-деформированного состояния плиты в условиях нелинейного деформирования, но для ребристых оболочек такая методика приводит к громоздким уравнениям.

При вычислении  опускаем сомножитель . В результате получим два варианта соотношений. Первый вариант получается, если  взять в виде (4) и тогда



 (25)



 (26)



 (27)

Решение задач ползучести для оболочек возможно лишь при применении приближенных методик.

Чтобы избежать решения интегральных уравнений, интегралы по переменной  на отрезке  разобьем на сумму интегралов по частичным отрезкам , обозначив , и последние вычислим приближенно по формуле прямоугольников. Такая методика применялась в работах [14, 8].

В результате  примут вид









 (28)



 (29)









 (30)

Здесь  имеют вид (6). Например, для оргстекла [14]

 (31), где



 и тогда

 (32)

для старого бетона [2]

 (33), где

 и тогда

 (34)

При решении задач ползучести для оболочек при каждом значении параметра нагрузки  решается итерационная задача

 (35)

до тех пор, пока прогибы не будут резко возрастать (в 10-15 раз по сравнению с первоначальным значением).

Начальное приближение  находится из решения линейно-упругой задачи (24).

Второй вариант соотношений  получается, если взять  в виде (5) и тогда (так как деформации при  считаются известными, то производные от них по  равны нулю)







 (36)





 (37)











 (38)

Таким образом, выражения  оказываются одинаковыми, как для , взятого в виде (4), так и для , взятого в виде (5). При учете геометрической нелинейности такого полного совпадения не будет. При использовании  в виде (5) значение правых частей системы (15) будут несколько больше, чем при использовании  в виде (4), что пойдет в запас прочности.

## 2.1 Программа PologObolochka

Программа предназначена для расчетов прочности и устойчивости оболочек при учете геометрической и физической нелинейностей и ползучести материала и разработана Беркалиевым Р.Т. [11] Программа может быть запущенна под любой версией ОС Windows, начиная с версии NT.

Программа состоит из нескольких базовых блоков:

Получение коэффициентов С систем алгебраических уравнений линейно-упругой задачи;

Метод итераций для геометрически и физически-нелинейной задачи;

Построения графиков устойчивости;

Построение 3-D графиков устойчивости;

Метод итераций ползучести (с построением графиков);

Построение 3-D графиков ползучести.

От физической модели не зависит блок 1, все остальные блоки зависят от нее. Таким образом, в зависимости от физической линейности или нелинейности вызываются соответствующие блоки. Блок 1 и блок 2 являются базовыми для расчета любой задачи.

*Блок 1: Получение коэффициентов С систем алгебраических уравнений линейно-упругой задачи.*

Блок вычисляет коэффициенты C для составления базовой системы уравнений модели расчета, и записывает их в файл, чтобы в дальнейшем, для этой же задачи загрузить их из файла. Коэффициенты вычисляются по ряду параметров, которые запрещено менять в дальнейших вычислениях, при расчете текущей задачи.

*Блок 2: Метод итераций для геометрически и физически-нелинейной задачи.*

Метод итераций проводит расчет устойчивости как физически линейных, так и физически нелинейных моделей оболочек, в зависимости от выбранной модели.

В процессе метода итераций, по заданному фильтру PausePW, сохраняются значения *P*, *U*, *V*, *W*, *PS*, *PN* в файл, и сохраняется последнее состояние метода. В дальнейшем оттуда берутся данные для продолжения метода итераций, в случае его аварийного завершения. Таким образом, метод итераций может быть прерван в любой момент времени и потом продолжен с момента его прерывания.

*Блок 3: Построения графиков устойчивости.*

По заданному фильтру составляется зависимость *P-W*. Из файла метода Итераций берутся эти значения, и вычисляются прогибы, для *P-W*, и заносятся в файл. Блок составляет файл зависимости *P-W*, по которому может быть построен график зависимости.

*Блок 4: Построение 3-D графиков устойчивости.*

По заданному фильтру составляются поля прогибов и напряжений для оболочки для физически линейной или нелинейной модели. Вычисления ведутся только для тех значений, нагрузка-P, которые попадают в список заданных нагрузок, для которых запрашивались прогибы и напряжения.

Из файла блока метода итераций считывается *U*, *V*, *W*, *PS*, *PN*, для заданных *P*, и вычисляют поля прогибов *W* (*x*,*y*) и напряжений  (*x*,*y*) и они записываются в файлы. По сохраненным расчетам могут быть построены графики. Интенсивность напряжений  (*x*,*y*) вычисляется в зависимости от материала (хрупкие или пластичные).

*Блок 5: Метод итераций ползучести.*

По заданному фильтру блок производит расчет ползучести оболочки для физически линейной или нелинейной модели. Из файла метода итераций считываются *U*, *V*, *W*, *PS*, *PN*, для заданных *P*. Для каждого *P* инициализируется система уравнений ползучести, и ведется расчет ползучести материала, и составляется зависимость *W-t*, которая записывается в файл. Эта зависимость может быть в дальнейшем представлена в графическом виде.

*Блок 6: Построение 3-D графиков ползучести.*

По заданному фильтру производится расчет полей прогибов и напряжений для физически линейной или нелинейной задачи. Из файла метода итераций считываются *U*, *V*, *W*, *PS*, *PN*, для выбранного *P*. Инициализируется система уравнений, и ведется расчет. Для заданных точек по времени составляются поля прогибов и напряжений, в зависимости от материала и физической модели оболочки, которые записываются в файлы. По результатам расчета могут быть построены графические зависимости.

Для графического представления зависимостей используется математический пакет Maple, начиная с версии Maple 6.

Для расчета любой задачи входными данными являются выбор физической модели: физически линейная или физически нелинейная (рис.2).

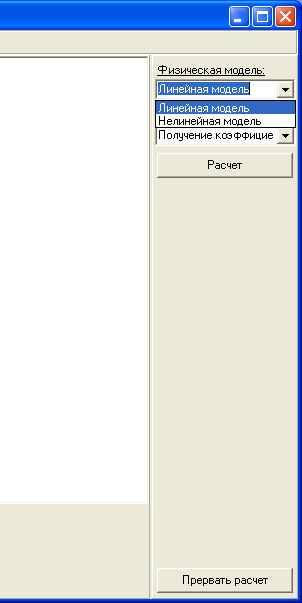


Рис.2. Выбор модели

*Входными данными являются:*

Dim\_a - Линейный размер оболочки;

Kksi - Кривизна оболочки вдоль направления *X*;

Keta - Кривизна оболочки вдоль направления *Y*;

lambda - Соотношение длин сторон вдоль направления *X* и *Y* оболочки;

Mu - Коэффициент Пуассона;

J\_m - Количество ребер жесткости вдоль направления *X*;

I\_n - Количество ребер жесткости вдоль направления *Y*;

AutomaticModeRebra - Режим ввода центральных координат ребер жесткости;

F\_j - Высота ребер жесткости вдоль направления *X*;

F\_i - Высота ребер жесткости вдоль направления *Y*;

rj - Ширина ребер жесткости вдоль направления *X*;

ri - Ширина ребер жесткости вдоль направления *Y*;

temporaryRebraX - Центральные координаты ребер жесткости вдоль направления *X*;

temporaryRebraY - Центральные координаты ребер жесткости вдоль направления *Y*;

N - Количество аппроксимирующих функций;

N\_Simps - Разбиение при вычислении интегралов для коэффициентов СЛАУ;

P\_KON - Конечная нагрузка;

d\_P - Шаг по нагрузке метода Итераций;

PausePW - Шаг по нагрузке вывода результатов расчета;

epsilon - Точность метода Итераций;

Iter\_X1 - Координата *X* вычисления прогиба оболочки для метода Итераций;

Iter\_Y1 - Координата *Y* вычисления прогиба оболочки для метода Итераций;

Ust\_X1 - Координата *X* вычисления прогиба оболочки для построения 1-й зависимости *P-W*;

Ust\_Y1 - Координата *Y* вычисления прогиба оболочки для построения 1-й зависимости *P-W*;

Ust\_X2 - Координата *X* вычисления прогиба оболочки для построения 2-й зависимости *P-W*;

Ust\_Y2 - Координата *Y* вычисления прогиба оболочки для построения 2-й зависимости *P-W*;

Ust\_P - Значения нагрузок для вывода полей прогиба и интенсивности оболочки;

CreepEnd - Порог превышения прогиба для вычисления ползучести оболочки;

Creep\_X1 - Координата *X* вычисления прогиба оболочки для построения 1-й зависимости *W-t*;

Creep\_Y1 - Координата *Y* вычисления прогиба оболочки для построения 1-й зависимости *W-t*;

Creep\_X2 - Координата *X* вычисления прогиба оболочки для построения 2-й зависимости *W-t*;

Creep\_Y2 - Координата *Y* вычисления прогиба оболочки для построения 2-й зависимости *W-t*;

Creep\_P - Значения нагрузок для вычисления ползучести оболочки;

DimP\_Time - Значение нагрузки для вывода динамических полей прогиба и интенсивности оболочки;

CreepTime - Значения времени для вывода динамических полей прогиба и интенсивности оболочки при заданной нагрузке;

Material - Тип материала (для упругих материалов задача ползучести не рассчитывается);

constM - Константа физической нелинейности.

При вводе данных программа показывает подсказку по каждому параметру (рис.3).

После ввода всех параметров можно приступать к расчету.

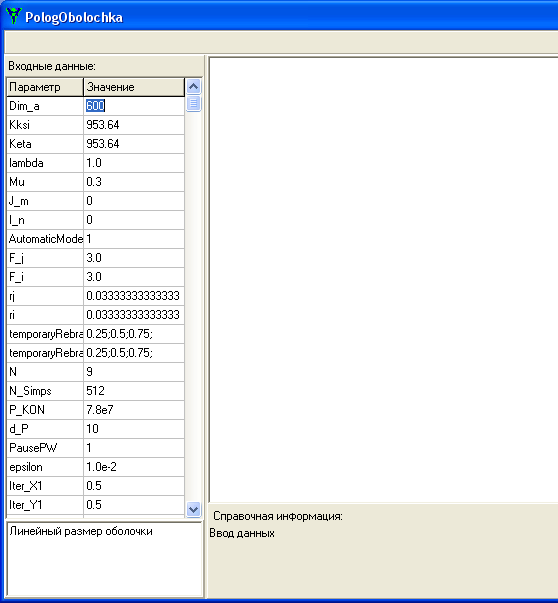


Рис.3. Ввод входных данных

## Глава 3. Распараллеливание процесса вычисления. Основы, принципы, практическое применение

Распараллеливание процесса вычисления - процесс адаптации [алгоритмов](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), записанных в виде программ, для их эффективного исполнения на вычислительной системе [параллельной архитектуры](http://ru.wikipedia.org/w/index.php?title=%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0&action=edit&redlink=1). Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный трансляторам целевой вычислительной системы, либо к вставке специальной разметки (например, инструкций [MPI](http://ru.wikipedia.org/wiki/MPI) или [OpenMP](http://ru.wikipedia.org/wiki/OpenMP)[[1]](#footnote-1)). [1]

История

В 1973 году Джон Шох и Джон Хапп из калифорнийского научно-исследовательского центра [Xerox PARC](http://ru.wikipedia.org/wiki/Xerox_PARC) написали программу, которая по ночам запускалась в локальную сеть PARC и заставляла работающие компьютеры выполнять вычисления.

В 1988 году [Арьен Ленстра](http://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BD%D1%81%D1%82%D1%80%D0%B0,_%D0%90%D1%80%D1%8C%D0%B5%D0%BD) и Марк Менес написали программу для [факторизации](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) длинных чисел. Для ускорения процесса программа могла запускаться на нескольких машинах, каждая из которых обрабатывала свой небольшой фрагмент. Новые блоки заданий рассылались на компьютеры участников с центрального сервера проекта по [электронной почте](http://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D1%87%D1%82%D0%B0). Для успешного разложения на множители числа длиной в сто знаков этому сообществу потребовалось два года и несколько сотен персональных компьютеров.

С появлением и бурным развитием [интернета](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82) всё большую популярность стала получать идея добровольного использования для распределённых вычислений компьютеров простых пользователей, соединённых через [интернет](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82).

В январе 1996 года стартовал проект [GIMPS](http://ru.wikipedia.org/wiki/GIMPS) по поиску простых [чисел Мерсенна](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE_%D0%9C%D0%B5%D1%80%D1%81%D0%B5%D0%BD%D0%BD%D0%B0).

28 января 1997 года стартовал конкурс RSA Data Security на решение задачи взлома методом простого перебора 56-битного ключа шифрования информации [RC5](http://ru.wikipedia.org/wiki/RC5). Благодаря хорошей технической и организационной подготовке проект, организованный некоммерческим сообществом [distributed.net](http://ru.wikipedia.org/wiki/Distributed.net), быстро получил широкую известность.

17 мая 1999 года на базе платформы [BOINC](http://ru.wikipedia.org/wiki/BOINC) запущен проект [SETI@home](http://ru.wikipedia.org/wiki/SETI@home), занимающийся поиском внеземного разума путем анализа данных с [радиотелескопов](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B4%D0%B8%D0%BE%D1%82%D0%B5%D0%BB%D0%B5%D1%81%D0%BA%D0%BE%D0%BF) (анализ данных проводился и раньше, но без использования [грид](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B8%D0%B4)).

Такие проекты распределённых вычислений в [интернете](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82), как [SETI@Home](http://ru.wikipedia.org/wiki/SETI@Home) и [Folding@Home](http://ru.wikipedia.org/wiki/Folding@Home) обладает не меньшей вычислительной мощностью, чем самые современные [суперкомпьютеры](http://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80). Интегральный объем вычислений на платформе [BOINC](http://ru.wikipedia.org/wiki/BOINC) по данным на 16 мая 2010 года составляет 5.2 пета[флопс](http://ru.wikipedia.org/wiki/FLOPS), в то время как пиковая производительность самого мощного суперкомпьютера ([Jaguar - Cray XT5-HE](http://ru.wikipedia.org/wiki/Jaguar_%28%D1%81%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%29)) - "всего" 2.3 петафлопс. Проект отмечен в [Книге рекордов Гиннеса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BD%D0%B8%D0%B3%D0%B0_%D1%80%D0%B5%D0%BA%D0%BE%D1%80%D0%B4%D0%BE%D0%B2_%D0%93%D0%B8%D0%BD%D0%BD%D0%B5%D1%81%D0%B0) как самое большое вычисление.

Распараллеливание может быть ручным, автоматизированным и полуавтоматизированным. Для оценки эффективности его качества применяются следующие критерии:

Ускорение , где  - время исполнения распараллеленной программы на p процессорах,  - время исполнения исходной программы. В идеальном случае (отсутствие накладных расходов на организацию параллелизма) равна .

Загруженность , показывающая долю использования процессоров. В идеальном случае равна 1, или 100%. Эта величина зачастую гораздо более наглядно характеризует эффективность параллелизма в серии испытаний при разных , чем , особенно на графиках.

При распараллеливании важно учитывать не только формальный параллелизм структуры алгоритма, но и то, что обменные операции в параллельных [ЭВМ](http://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C) происходят, как правило, значительно медленнее арифметических. С этим связано существование львиной доли накладных расходов на организацию параллелизма.

В [информатике](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) параллельный алгоритм, противопоставляемый традиционным [последовательным алгоритмам](http://en.wikipedia.org/wiki/Sequential_algorithm), - [алгоритм](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата.

Некоторые алгоритмы достаточно просто поддаются разбиению на независимо выполняемые фрагменты. Например, распределение работы по проверке всех чисел от 1 до 100000 на предмет того, какие из них являются [простыми](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE), может быть выполнено путем назначения каждому доступному процессору некоторого подмножества чисел с последующим объединением полученных множеств простых чисел (похожим образом реализован, например, проект [GIMPS](http://ru.wikipedia.org/wiki/GIMPS)[[2]](#footnote-2)).

С другой стороны, большинство известных алгоритмов вычисления значения числа π не допускают разбиения на параллельно выполняемые части, т.к требуют результата предыдущей итерации выполнения алгоритма. Итеративные [численные методы](http://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), такие как, например, [метод Ньютона](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9D%D1%8C%D1%8E%D1%82%D0%BE%D0%BD%D0%B0) или [задача трёх тел](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D1%82%D1%80%D1%91%D1%85_%D1%82%D0%B5%D0%BB), также являются сугубо последовательными алгоритмами. Некоторые примеры [рекурсивных](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F) алгоритмов достаточно сложно поддаются распараллеливанию.

Параллельные алгоритмы весьма важны ввиду постоянного совершенствования [многопроцессорных систем](http://en.wikipedia.org/wiki/multiprocessing) и увеличения числа [ядер](http://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D1%8F%D0%B4%D0%B5%D1%80%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) в современных процессорах. Обычно проще сконструировать компьютер с одним быстрым процессором, чем с множеством медленных процессоров (при условии достижения одинаковой [производительности](http://en.wikipedia.org/wiki/Throughput)). Однако производительность процессоров увеличивается главным образом за счет совершенствования техпроцесса (уменьшения норм производства), чему мешают [физические ограничения](http://ru.wikipedia.org/wiki/%D0%A2%D1%83%D0%BD%D0%BD%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82) на размер элементов микросхем и тепловыделение. Указанные ограничения могут быть преодолены путем перехода к многопроцессорной обработке, что оказывается эффективным даже для малых вычислительных систем.

Сложность последовательных алгоритмов выражается в объеме используемой памяти и времени (числе тактов процессора), необходимых для выполнения алгоритма. Параллельные алгоритмы требуют учета использования еще одного ресурса: подсистемы связей между различными процессорами. Существует два способа обмена между процессорами: использование общей памяти и системы передачи сообщений.

[Системы с общей памятью](http://en.wikipedia.org/wiki/Shared_memory) требуют введения дополнительных [блокировок](http://en.wikipedia.org/wiki/Lock_%28computer_science%29) для обрабатываемых данных, налагая определенные ограничения при использовании дополнительных процессоров.

[Системы передачи сообщений](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8) используют понятия каналов и блоков сообщений, что создает дополнительный трафик на шине и требует дополнительных затрат памяти для организации очередей сообщений. В дизайне современных процессоров могут быть предусмотрены специальные коммутаторы (кроссбары) с целью уменьшения влияния обмена сообщениями на время выполнения задачи.

Еще одной проблемой, связанной с использованием параллельных алгоритмов, является [балансировка нагрузки](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BD%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B8). Например, поиск простых чисел в диапазоне от 1 до 100000 легко распределить между имеющимися процессорами, однако некоторые процессоры могут получить больший объем работы, в то время как другие закончат обработку раньше и будут простаивать. Проблемы балансировки нагрузки еще больше усугубляется при использовании гетерогенных вычислительных сред, в которых вычислительные элементы существенно отличаются по производительности и доступности (например, в [грид](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B8%D0%B4)-системах).

Разновидность параллельных алгоритмов, называемая [распределенными алгоритмами](http://en.wikipedia.org/wiki/Distributed_algorithms), специально разрабатываются для применения на [кластерах](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_%28%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2%29) и в [распределенных вычислительных системах](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) с учетом ряда особенностей подобной обработки.

В данной работе был выбран интерфейс MPI для распараллеливания процесса, так как он наиболее удобен для создаваемого программного пакета. Ведь заранее неизвестно, какие вычислительные машины и какой конфигурации будут использоваться. MPI позволяет эмулировать[[3]](#footnote-3) вычислительные машины на ядрах одного процессора, что, в некоторых случаях, гораздо более эффективнее при вычислениях.

## 3.1 Message Passing Interface

Message Passing Interface (MPI, интерфейс передачи сообщений) - программный интерфейс ([API](http://ru.wikipedia.org/wiki/API)) для передачи [информации](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F), который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан [Уильямом Гроуппом](http://en.wikipedia.org/wiki/William_Gropp), [Эвином Ласком](http://en.wikipedia.org/wiki/Ewing_Lusk) и другими.

MPI является наиболее распространённым стандартом интерфейса обмена данными в [параллельном программировании](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для [кластеров](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_%28%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2%29) и [суперкомпьютеров](http://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80). Основным средством коммуникации между [процессами](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) в MPI является передача сообщений друг другу. Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в [приложениях](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) пользователя. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков [Фортран](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D1%82%D1%80%D0%B0%D0%BD) 77/90, [Си](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) и [Си++](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%D0%BF%D0%BB%D1%8E%D1%81_%D0%BF%D0%BB%D1%8E%D1%81).

*Стандарты MPI*

Большинство современных реализаций MPI поддерживают версию 1.1 Стандарт MPI версии 2.0 поддерживается большинством современных реализаций, однако некоторые функции могут быть реализованы не до конца.

В MPI 1.1 (опубликован [12 июня](http://ru.wikipedia.org/wiki/12_%D0%B8%D1%8E%D0%BD%D1%8F) [1995 года](http://ru.wikipedia.org/wiki/1995_%D0%B3%D0%BE%D0%B4)) поддерживаются следующие функции:

передача и получение сообщений между отдельными процессами;

коллективные взаимодействия процессов;

взаимодействия в группах процессов;

реализация [топологий](http://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BF%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%8F) процессов;

В MPI 2.0 (опубликован [18 июля](http://ru.wikipedia.org/wiki/18_%D0%B8%D1%8E%D0%BB%D1%8F) [1997 года](http://ru.wikipedia.org/wiki/1997_%D0%B3%D0%BE%D0%B4)) дополнительно поддерживаются следующие функции:

динамическое порождение процессов и управление процессами;

односторонние коммуникации (Get/Put);

параллельный ввод и вывод;

расширенные коллективные операции (процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов).

*Пример программы*

Ниже приведён пример программы на C с использованием MPI:

// Подключение необходимых заголовков

#include <stdio. h>

#include <math. h>

#include "mpi. h"

// Функция для промежуточных вычислений

double f (double a)

{

return (4.0/ (1.0+ a\*a));

}

// Главная функция программы

int main (int argc, char \*\*argv)

{

// Объявление переменных

int done = 0, n, myid, numprocs, i;

double PI25DT = 3.141592653589793238462643;

double mypi, pi, h, sum, x;

double startwtime = 0.0, endwtime;

int namelen;

char processor\_name [MPI\_MAX\_PROCESSOR\_NAME] ;

// Инициализация подсистемы MPI

MPI\_Init (&argc, &argv);

MPI\_Comm\_size (MPI\_COMM\_WORLD,&numprocs);

MPI\_Comm\_rank (MPI\_COMM\_WORLD,&myid);

MPI\_Get\_processor\_name (processor\_name,&namelen);

// Вывод номера потока в общем пуле

fprintf (stdout, "Process%d of%d is on%s\n", myid,numprocs,processor\_name);

fflush (stdout);

while (! done)

{

// количество интервалов

if (myid==0)

{

fprintf (stdout, "Enter the number of intervals: (0 quits)");

fflush (stdout);

if (scanf ("%d",&n) ! = 1)

{

fprintf (stdout, "No number entered; quitting\n");

n = 0;

}

startwtime = MPI\_Wtime ();

}

// Рассылка количества интервалов всем процессам (в том числе и себе)

MPI\_Bcast (&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (n==0)

done = 1;

else

{

h = 1.0/ (double) n;

sum = 0.0;

// Обсчитывание точки, закрепленной за процессом

for (i = myid + 1; (i <= n); i += numprocs)

{

x = h \* ( (double) i - 0.5);

sum += f (x);

}

mypi = h \* sum;

// Сброс результатов со всех процессов и сложение

MPI\_Reduce (&mypi, &pi, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

// Если это главный процесс, вывод полученного результата

if (myid==0)

{

printf ("PI is approximately%.16f, Error is%.16f\n", pi, fabs (pi - PI25DT));

endwtime = MPI\_Wtime ();

printf ("wall clock time =%f\n", endwtime-startwtime);

fflush (stdout);

}

}

}

// Освобождение подсистемы MPI

MPI\_Finalize ();

return 0;

}

*Реализации MPI*

[MPICH](http://ru.wikipedia.org/wiki/MPICH) - самая распространённая бесплатная реализация, работает на [UNIX](http://ru.wikipedia.org/wiki/UNIX)-системах и [Windows NT](http://ru.wikipedia.org/wiki/Windows_NT);

[WMPI](http://www.criticalsoftware.com/hpc) - бесплатная реализация MPI для [Windows](http://ru.wikipedia.org/wiki/Windows);

[LAM/MPI](http://www.lam-mpi.org/) - ещё одна бесплатная реализация MPI;

[MPI/PRO](http://www.mpi-softtech.com/mpi_pro.php) for Windows NT - коммерческая реализация для [Windows NT](http://ru.wikipedia.org/wiki/Windows_NT);

[Intel MPI](http://software.intel.com/en-us/intel-mpi-library/) - коммерческая реализация для [Windows](http://ru.wikipedia.org/wiki/Windows) / [GNU/Linux](http://ru.wikipedia.org/wiki/GNU/Linux);

[HP-MPI](http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,1238,00.html) - коммерческая реализация от [HP](http://ru.wikipedia.org/wiki/Hewlett-Packard);

[SGI](http://ru.wikipedia.org/wiki/SGI) [MPT](http://www.sgi.com/products/software/mpt/) - платная библиотека MPI от [SGI](http://ru.wikipedia.org/wiki/SGI);

[Mvapich](http://mvapich.cse.ohio-state.edu/) - бесплатная реализация MPI для [Infiniband](http://ru.wikipedia.org/wiki/Infiniband);

[Open MPI](http://www.open-mpi.org/) - бесплатная реализация MPI, наследник LAM/MPI;

[Oracle HPC ClusterTools](http://www.sun.com/software/products/clustertools/index.xml) - бесплатная реализация для [Solaris](http://ru.wikipedia.org/wiki/Solaris_%28%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0%29) [SPARC](http://ru.wikipedia.org/wiki/SPARC)/[x86](http://ru.wikipedia.org/wiki/X86) и [Linux](http://ru.wikipedia.org/wiki/Linux) на основе Open MPI.

## 3.2 MPICH

MPICH ("MPI Chameleon") - это одна из самых первых разработанных библиотек [MPI](http://ru.wikipedia.org/wiki/MPI). На ее базе было создано большое количество других библиотек как [OpenSource](http://ru.wikipedia.org/wiki/OpenSource), так и [коммерческих](http://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%BE%D0%BC%D0%BC%D0%B5%D1%80%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B9_%D0%9F%D0%9E&action=edit&redlink=1). В настоящее время существует две ветви исходных кодов: MPICH1и MPICH2. Разработка ветви MPICH1 заморожена. Ветвь MPICH2 активно разрабатывается в Арагонской лаборатории, с участием IBM, Cray, SiCortex, Microsoft, Intel, NetEffect, Qlogic, Myricom, Ohio state university, UBC.

MPICH2 - легко портируемая быстрая реализация стандарта MPI. Отличительные особенности:

Поддерживает различные вычислительные и коммуникационные платформы, включая общедоступные кластеры (настольные системы, системы с общей памятью, многоядерные архитектуры), высокоскоростные сети (Ethernet 10 ГБит/с, [InfiniBand](http://ru.wikipedia.org/wiki/InfiniBand), [Myrinet](http://ru.wikipedia.org/wiki/Myrinet), Quadrics) и эксклюзивные вычислительные системы ([Blue Gene](http://ru.wikipedia.org/wiki/Blue_Gene), [Cray](http://ru.wikipedia.org/wiki/Cray), [SiCortex](http://ru.wikipedia.org/w/index.php?title=SiCortex&action=edit&redlink=1)).

Модульная структура для создания производных реализаций, предоставляющая широкие возможности для исследования технологии MPI.

В дальнейшем будем предполагать, что имеется сеть из нескольких компьютеров (будем называть их ***вычислительными узлами***), работающих под управлением Windows. Для учебных целей можно запускать все вычислительные процессы и на одном компьютере. Если компьютер одноядерный, то, естественно, прирост быстродействия не получится, - только замедление. В качестве среды разработки использовалась IDE Microsoft Visual Studio 2008 C++ Express Edition. Microsoft Visual Studio - линейка продуктов компании [Майкрософт](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82), включающих [интегрированную среду разработки](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) программного обеспечения и ряд других инструментальных средств. Microsoft Visual Studio 2008 C++ Express Edition - легковесная среда разработки, представляющая собой урезанную версию Visual Studio. В отличие от полной версии, такая среда предназначена для языка программирования C/C++. Она включает в себя небольшой набор инструментов, в отличие от полных версий: отсутствует дизайнер классов и многие другие инструменты, а также поддержка [плагинов](http://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B3%D0%B8%D0%BD) и удалённых [баз данных](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) в дизайнере данных. Компиляторы в 64-битный код также недоступны в Express редакциях. [Microsoft](http://ru.wikipedia.org/wiki/Microsoft) позиционирует эту линейку [IDE](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) для студентов. Программу, использующую MPI, и предназначенную для запуска на нескольких вычислительных узлах, будем называть ***MPI-программой***.

## 3.3 Принципы работы MPICH

MPICH для Windows состоит из следующих компонентов:

Менеджер процессов smpd. exe, который представляет собой системную службу (сервисное приложение). Менеджер процессов ведёт список вычислительных узлов системы, и запускает на этих узлах MPI-программы, предоставляя им необходимую информацию для работы и обмена сообщениями;

Заголовочные файлы (. h) и библиотеки стадии компиляции (. lib), необходимые для разработки MPI-программ;

Библиотеки времени выполнения (. dll), необходимые для работы MPI-программ;

Дополнительные утилиты (. exe), необходимые для настройки MPICH и запуска MPI-программ.

Все компоненты, кроме библиотек времени выполнения, устанавливаются по умолчанию в папку *C: \Program Files\MPICH2*; dll-библиотеки устанавливаются в *C: \Windows\System32*.

Менеджер процессов является основным компонентом, который должен быть установлен и настроен на всех компьютерах сети (библиотеки времени выполнения можно, в крайнем случае, копировать вместе с MPI-программой). Остальные файлы требуются для разработки MPI-программ и настройки "головного" компьютера, с которого будет производиться их запуск.

Менеджер работает в фоновом режиме и ждёт запросов к нему из сети со стороны "головного" менеджера процессов (по умолчанию используется сетевой порт 8676). Чтобы обезопасить себя от хакеров и вирусов, менеджер требует пароль при обращении к нему.

Когда один менеджер процессов обращается к другому менеджеру процессов, он передаёт ему свой пароль.

Отсюда следует, что нужно указывать один и тот же пароль при установке MPICH на компьютеры сети.

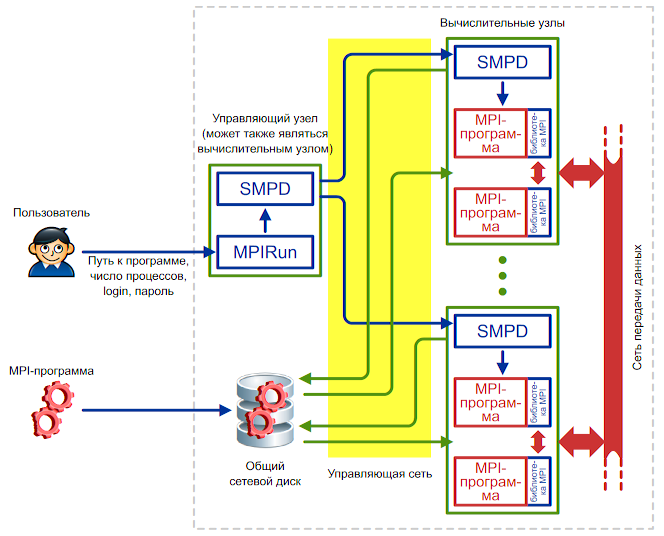


Рис.4. Схема работы MPICH на кластере.

В современных кластерах сеть передачи данных обычно отделяется от управляющей сети.

**Запуск** MPI-программы производится следующим образом (рис.4):

Пользователь с помощью программы Mpirun (или Mpiexec, при использовании MPICH2 под Windows) указывает имя исполняемого файла MPI-программы и требуемое число процессов. Кроме того, можно указать имя пользователя и пароль: процессы MPI-программы будут запускаться от имени этого пользователя.

Mpirun передаёт сведения о запуске локальному менеджеру процессов, у которого имеется список доступных вычислительных узлов.

Менеджер процессов обращается к вычислительным узлам по списку, передавая запущенным на них менеджерам процессов указания по запуску MPI-программы.

Менеджеры процессов запускают на вычислительных узлах несколько копий MPI-программы (возможно, по несколько копий на каждом узле), передавая программам необходимую информацию для связи друг с другом.

Очень важным моментом здесь является то, что перед запуском MPI-программа не копируется автоматически на вычислительные узлы кластера. Вместо этого менеджер процессов передаёт узлам путь к исполняемому файлу программы точно в том виде, в котором пользователь указал этот путь программе Mpirun. Это означает, что если, например, запускается программа C: \program. exe, то все менеджеры процессов на вычислительных узлах будут пытаться запустить файл C: \program. exe. Если хотя бы на одном из узлов такого файла не окажется, произойдёт ошибка запуска MPI-программы.

Чтобы каждый раз не копировать вручную программу и все необходимые для её работы файлы на вычислительные узлы кластера, обычно используют ***общий сетевой ресурс***. В этом случае пользователь копирует программу и дополнительные файлы на сетевой ресурс, видимый всеми узлами кластера, и указывает путь к файлу программы на этом ресурсе. Дополнительным удобством такого подхода является то, что при наличии возможности записи на общий сетевой ресурс запущенные копии программы могут записывать туда результаты своей работы.

**Работа** MPI-программы происходит следующим образом:

Программа запускается и инициализирует библиотеку времени выполнения MPICH путём вызова функции MPI\_Init.

Библиотека получает от менеджера процессов информацию о количестве и местоположении других процессов программы, и устанавливает с ними связь.

После этого запущенные копии программы могут обмениваться друг с другом информацией посредством библиотеки MPICH. С точки зрения операционной системы библиотека является частью программы (работает в том же процессе), поэтому можно считать, что запущенные копии MPI-программы обмениваются данными напрямую друг с другом, как любые другие приложения, передающие данные по сети.

Консольный ввод-вывод всех процессов MPI-программы перенаправляется на консоль, на которой запущена Mpirun. Насколько я понимаю, перенаправлением ввода-вывода занимаются менеджеры процессов, так как именно они запустили копии MPI-программы, и поэтому могут получить доступ к потокам ввода-вывода программ.

Перед завершением все процессы вызывают функцию MPI\_Finalize, которая корректно завершает передачу и приём всех сообщений, и отключает MPICH.

Все описанные выше принципы действуют, даже если вы запускаете MPI-программу на одном компьютере.

## 

## 3.4 Установка MPICH в Windows

Необходимо загрузить последнюю версию MPICH2 со страницы [http://www.​mcs. ​anl. ​gov/​research/​projects/​mpich2/​downloads/​index. php? ​ s=downloads](http://www.​mcs.​anl.​gov/​research/​projects/​mpich2/​downloads/​index.php?​%20s=downloads). Загруженный инсталлятор необходимо запустить с привилегиями администратора на всех компьютерах, на которых планируется запуск MPI-программ.

Во время установки необходимо ввести пароль для доступа к менеджеру процессов SMPD. Он должен быть одинаковый на всех компьютерах:

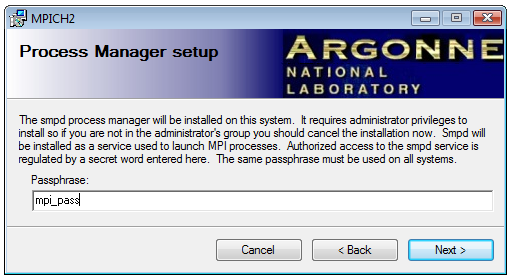


Рис.5. Указание пароля для доступа к менеджеру процессов

В окне указания пути установки рекомендуется оставить каталог по умолчанию и выбрать пункт "Everyone":

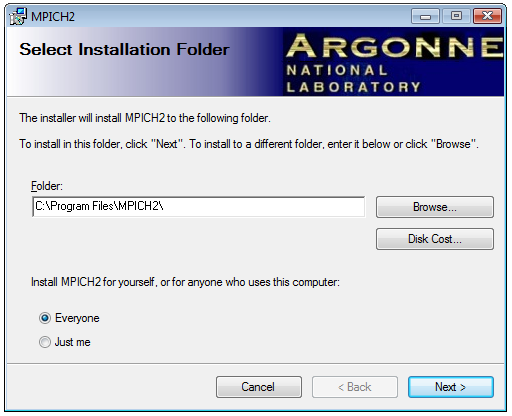


Рис.6. Указание пути установки

Если Брандмауэр Windows спросит, разрешить ли доступ в сеть программе smpd. exe, то необходимо "Разрешить".

MPICH2 правильно установлен на ваш компьютер.

Однако, прежде чем переходить к настройке, обязательно следует проверить две вещи: запущена ли служба "MPICH2 Process Manager", и разрешён ли этой службе доступ в сеть.

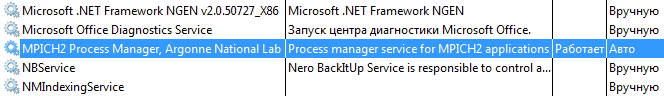


Рис.7. Служба "MPICH2 Process Manager" в списке служб

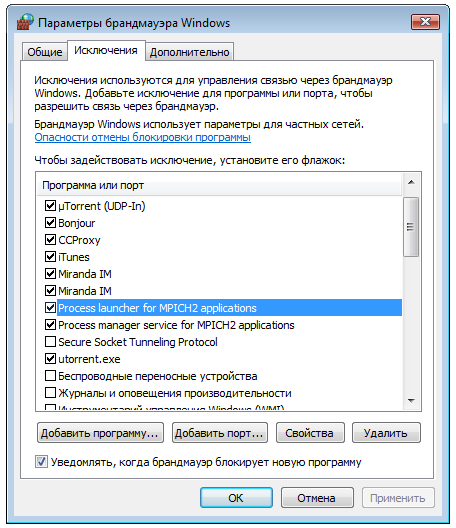


Рис.8. Программы MPICH в списке исключений брандмауэра

Если какая-то из перечисленных программ отсутствует в списке разрешённых программ, то вы необходимо добавить её вручную. Соотвественно, C: \​program files\​mpich2\​bin\​mpiexec. exe, если отсутствует "Process launcher for MPICH2 applications", и C: \​program files\​mpich2\​bin\​smpd. exe, если отсутствует "Process manager service for MPICH2 applications".

## 3.5 Настройка MPICH

Рассмотрим настройку MPICH на примере конфигурации из двух компьютеров, объединённых в локальную сеть: один компьютер имеет сетевое имя MrBig и IP-адрес 192.168.1.4, другой - имя Small и адрес 192.168.1.3 MPI-программы планируется запускать с компьютера MrBIG. На обоих компьютерах установлены русскоязычные версии Windows. На MrBIG установлена Windows Vista, на Small - Windows XP. Каждый компьютер имеет двухъядерный процессор.

Прежде всего **необходимо создать на всех компьютерах пользователя с одинаковым именем и паролем**; от имени этого пользователя будут запускаться MPI-программы.

*Wmpiregister*

Как уже было сказано ранее, любое действие система MPICH выполняет от указанного имени пользователя.

Для того, чтобы спрашивать имя пользователя и пароль, используется программа Wmpiregister. Проблема в том, что имя пользователя и пароль спрашиваются достаточно часто.

Для того, чтобы этого избежать, Wmpiregister может сохранять имя пользователя и пароль в реестре Windows.

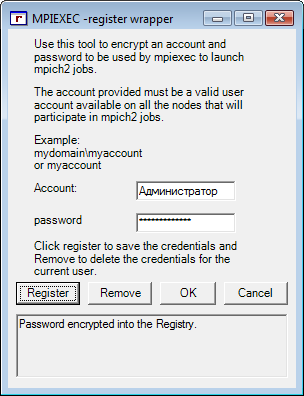


Рис.9. Программа Wmpiregister

"Cancel" - закрыть программу без выполнения какого-либо действия.

"OK" - передать введённые имя пользователя и пароль вызывающей программе.

"Remove" - нажатие этой кнопки удаляет сохранённые ранее имя пользователя и пароль из реестра Windows.

"Register" - сохраняет имя пользователя и пароль в реестре.

*Wmpiconfig*

Если все предыдущие шаги сделаны правильно, то в поле "version" в левой колонке таблицы будет написана версия установленного менеджера процессов (рис.10).

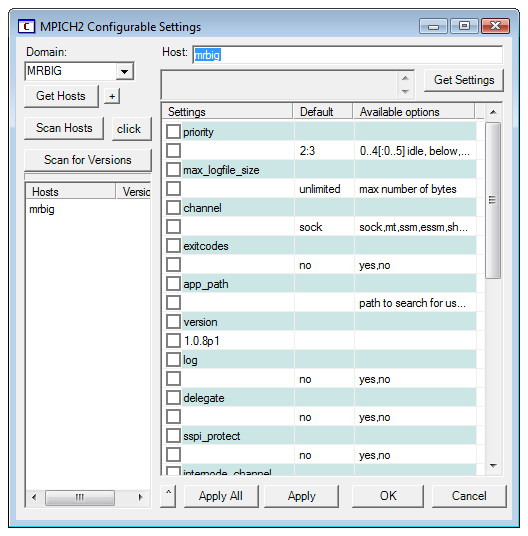


Рис.10. Программа Wmpiconfig

Wmpiconfig предназначена для настройки менеджеров процессов на текущем компьютере и других компьютерах сети. Для этого она подсоединяется к менеджерам процессов на выбранных компьютерах, читает имеющиеся у них настройки, и сообщает им новые настройки, если нужно. Элементы управления программы Wmpiconfig выполняют следующие действия:

Слева-внизу имеется список компьютеров, с которыми работает программа настройки. Имя компьютера на белом фоне означает, что не было попыток связаться с этим компьютером; зелёный фон означает, что связь произведена успешно; серый фон означает, что при установлении связи возникла ошибка. Удалить компьютер из списка можно клавишей Del. Следует иметь в виду, что этот список предназначен только для удобства настройки, и не имеет никакого отношения к списку компьютеров, на которых будет запущена MPI-программа.

Кнопка "Get Hosts" получает список компьютеров в заданном домене или рабочей группе (задаётся в выпадающем списке "Domain"). Полученный список заменяет имеющийся список компьютеров или, если нажата кнопка "+", добавляет компьютеры к текущему списку.

Кнопка "Scan Hosts" получает настройки со всех компьютеров списка; кнопка "Scan for Versions" получает только номера версий.

Кнопка "Get Settings" получает текущие настройки того компьютера, имя которого введено в поле ввода "Host". При выборе компьютера в списке компьютеров его имя автоматически вводится в поле "Host". Если нажата кнопка "Click", то настройки будут получены автоматически при выборе компьютера из списка.

Справа в окне расположена таблица настроек. Пустое поле означает, что используется настройка по умолчанию, указанная во втором столбце. Настройки, предназначенные к изменению, следует отмечать установкой галочки слева.

Кнопка "Apply" применяет выделенные галочкой настройки к тому компьютеру, имя которого находится в поле "Host". Кнопка "Apply All" применяет настройки ко всем компьютерам списка.

Кнопка "Cancel" закрывает программу.

На том компьютере, с которого планируется запуск программ, **необходимо указать список доступных вычислительных узлов**. Этот список вводится через пробел в поле hosts левого столбца таблицы (рисунок 11). На рисунке показан пример, когда сам компьютер, с которого производится запуск MPI-программ, является одним из вычислительных узлов.

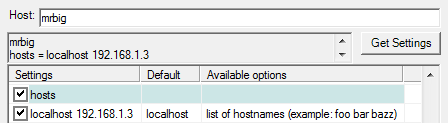


Рис.11. Список доступных вычислительных узлов

## 3.6 Создание общего сетевого ресурса

Для удобного запуска MPI-программ **следует создать на одном из компьютеров общий сетевой ресурс** с правом полного доступа для всех пользователей.

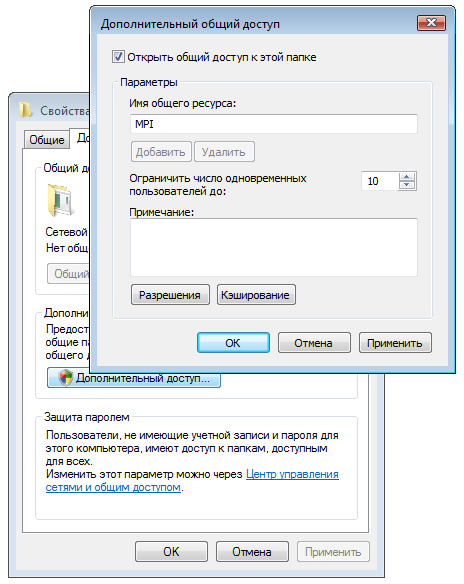


Рис.12. Окно свойств папки

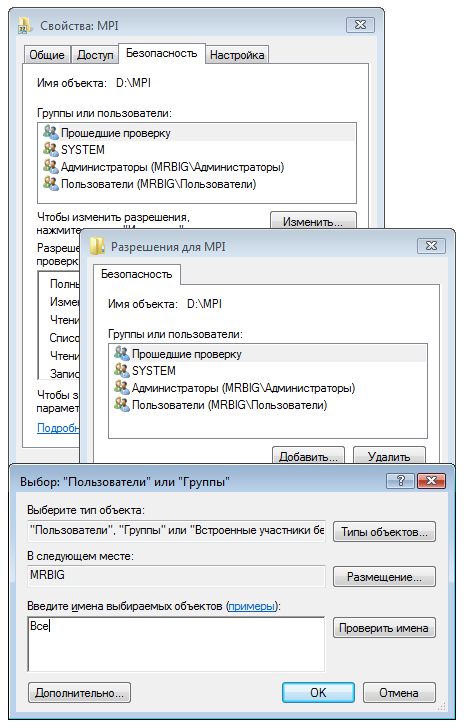


Рис.13. Добавление разрешений для доступа к папке

## 3.7 Запуск MPI-программ

Для запуска MPI-программ в комплект MPICH2 входит программа с графическим интерфейсом Wmpiexec, которая представляет собой оболочку вокруг соответствующей утилиты командной строки Mpiexec. Графический интерфейс нагляднее консольного приложения, но в ряде случаев запуск Mpiexec из консоли гораздо удобнее. Разницы для процесса вычисления нет, в любом случае используется Mpiexec.

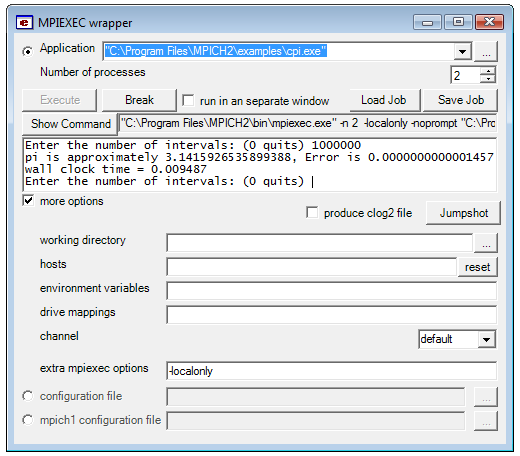


Рис.14. Программа Wmpiexec

Элементы управления окна имеют следующий смысл:

Поле ввода "Application": путь к MPI-программе. Как уже было сказано ранее, путь передаётся в неизменном виде на все компьютеры сети, поэтому желательно, чтобы программа располагалась в общей сетевой папке. Например, \\mrbig\mpi\mpi. exe.

"Number of processes": число запускаемых процессов. По умолчанию процессы распределяются поровну между компьютерами сети, однако это поведение можно изменить при помощи конфигурационного файла.

Кнопка "Execute" запускает программу; кнопка "Break" принудительно завершает все запущенные экземпляры.

Флажок "run in a separate window" перенаправляет вывод всех экземпляров MPI-программы в отдельное консольное окно.

Кнопка "Show Command" показывает в поле справа командную строку, которая используется для запуска MPI-программы (напоминаю: Wmpiexec - всего лишь оболочка над Mpiexec). Командная строка собирается из всех настроек, введённых в остальных полях окна.

Далее идёт большое текстовое поле, в которое попадает ввод-вывод всех экземпляров MPI-программы, если не установлен флажок "run in a separate window".

Флажок "more options" показывает дополнительные параметры.

"working directory": сюда можно ввести рабочий каталог программы. Опять же, этот путь должен быть верен на всех вычислительных узлах. Если путь не указан, то в качестве рабочего каталога будет использоваться место нахождения MPI-программы.

"hosts": здесь можно указать через пробел список вычислительных узлов, используемых для запуска MPI-программы. Если это поле пустое, то используется список, хранящийся в настройках менеджера процессов текущего узла.

"environment variables": в этом поле можно указать значения дополнительных переменных окружения, устанавливаемых на всех узлах на время запуска MPI-программы. Синтаксис следующий: имя1=значение1, имя2=значение2.

"drive mappings": здесь можно указать сетевой диск, подключаемый на каждом вычислительном узле на время работы MPI-программы. Синтакис: Z: \\winsrv\wdir.

"channel": позволяет выбрать способ передачи данных между экземплярами MPI-программы.

"extra mpiexec options": в это поле можно ввести дополнительные ключи для командной строки Mpiexec.

Одним из наиболее полезных дополнительных ключей является ключ -localonly, позволяющий эмулировать виртуальные вычислительные модули на одном компьютере. В случае с двухядерным процессором и двумя вычислительными процессами, процессы "раскидываются" на два ядра. Программа считается параллельно. Так как не тратится время на пересылку пакетов по сети, то время вычислений на виртуальных вычислительных модулях на одном локальном процессоре превосходит время вычислений на реальных модулях. Таким образом, в большинстве случаев нет необходимости строить локальную сети обмена данными, а достаточно сэмулировать этот процесс.

## Глава 4. Алгоритмы решения задач устойчивости для подкрепленных пологих оболочек, основанные на распараллеливании процесса вычисления

При исследовании устойчивых подкрепленных оболочек с учетом геометрической нелинейности приходится многократно решать системы алгебраических уравнений. Коэффициенты этих уравнений этих уравнений представляют собой двойные интегралы от громоздких выражений и их вычисление требует существенных затрат машинного времени. Чтобы сократить это время используется процесс распараллеливания вычислений.

При использовании одного и того же набора аппроксимирующих функций, программа каждый раз производит большое количество вычислений интегралов от различных произведений этих функции. Количество таких произведении пропорционально квадрату числа аппроксимирующих функций. Не смотря на это, программа производит, по сути, одни и те же вычисления. Вычисление интегралов является одной из самых длительных операций для ЭВМ. Учитывая, что эти действия многократно повторяются в циклах, эта часть вычислений становится самой долгой в контексте всего процесса вычисления.

Решить эту задачу можно путем заготовки заранее вычисленных интегралов и сохранение их в отдельной базе данных (БД). Каждому набору аппроксимирующих функций будет соответствовать БД с вычисленными интегралами. Главная задача будет состоять в построении правильной выборки значений для каждого количества аппроксимирующих членов. Схематично этот процесс показан ниже (табл.1).

Посторенние корректного запроса и его посылка в БД занимают гораздо меньше временных и процессорных ресурсов, чем вычисление интеграла. Из этого последует выигрыш по времени в 300-400% при решении комплекса задач при различных параметрах (при различной кривизне и толщине оболочки, при различных величинах нагрузки).

Следующим шагом в ускорении вычислений является частичное распараллеливание процесса вычисления.

Для этого используется MPI.

Табл.1. Пример проведения выборки из БД в зависимости от числа аппроксимирующих функций

|  |  |  |
| --- | --- | --- |
| N=4 | N=9 | N=16 |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | 5 | | 1 |  |  |  |  |  | | 2 |  |  |  |  |  | | 3 |  |  |  |  |  | | 4 |  |  |  |  |  | | 5 |  |  |  |  |  | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | 5 | | 1 |  |  |  |  |  | | 2 |  |  |  |  |  | | 3 |  |  |  |  |  | | 4 |  |  |  |  |  | | 5 |  |  |  |  |  | | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | 5 | | 1 |  |  |  |  |  | | 2 |  |  |  |  |  | | 3 |  |  |  |  |  | | 4 |  |  |  |  |  | | 5 |  |  |  |  |  | |

Одно из главных правил оптимизации заключается в упрощении действий, выполняемых многократно в циклах. Логично было бы применить этот же принцип и для распараллеливания.Т. е. "рутинная" многократная работа будет разделена между определенным количеством процессоров. Так как нет смысла распараллеливать всю программу, можно выделить "головную" ЭВМ, которая будет выполнять основные шаги вычисления. Остальные же машины будут обрабатывать пакет задач, принимаемые от ведущей машины.Т. е. получаем схему коммуникации процессоров, похожую на схему "клиент-сервер", за исключением того, что клиенты также могут обмениваться сообщениями, но только при согласовании своих действий с сервером (рис.14).

П

П

П

П

П

П

ЦП

Рис.14. Схема коммуникационной среды (ЦП - "головная" ЭВМ, П - "ведомая" ЭВМ)

Основные проблемы, которые могут возникнуть в процессе распределенных вычислений:

некорректно сформированный пакет задания или некорректный прием/сохранение результатов вычислений;

возникновение тупиковых ситуаций при приеме/передаче сообщений между процессорами;

неправильное разделение заданий между процессорами.

Ведущая ЭВМ на этапах распараллеливания будет освобождена от выполнения "рутинных" вычислений. Взамен она будет контролировать распределение заданий между ведомыми ЭВМ, фиксировать транзакции, отслеживать загруженность процессоров, принимать решение в случае возникновения ошибок, перенаправлять задания / повторно высылать задания в случае возникновения сбоя в среде коммуникации, собирать и проверять результаты вычислений.

Данная схема рассчитана на большое количество ЭВМ в коммуникационной среде.

На данном этапе для отработки схем распараллеливания и расчета выигрыша по времени используются две ЭВМ со схемой коммуникации типа "точка-точка", где обе машины являются равноправными.

Целью этой работы является создание программного интерфейса "PSS" (parallel solving for Shell) на основе MPI, позволяющего, не обращаясь к низкоуровневым командам MPI, производить "прозрачное" распараллеливание вычислений. Таким образом, общая блок-схема программы не изменится, за исключением того, что отдельные блоки в ней выполняются параллельно.

Ниже показана принципиальная блок-схема выполнения программы (рис.15). Блоки 2 и 6 можно производить в коммуникационной среде посредством нескольких процессоров, используя интерфейс PSS. Блок 3 можно распараллелить при использовании метода Гаусса для решения систем алгебраических уравнений.

Выигрыш по времени определяется следующими основными параметрами:

количество ЭВМ (процессоров) в коммуникационной среде;

отношение части кода программы, пригодной для распараллеливания, ко всему объему кода программы;

степень сложности логики проверки и контроля за вычислениями со стороны программного интерфейса PSS;

особенности аппаратной части (пропускная способность канала, скорость чтения/записи, частота процессоров, …).

(4) Вычисление U, V ,W, …

(2)Вычисление

коэф-ов Ci(I,l)

(3) Решение системы алг. ур-ий

(6) Вычисление правых частей

(5) Усл. выхода

Рис.15. Блок-схема выполнения программы

Если мы будем использовать N ЭВМ с одинаковой производительностью, то выигрыш по времени составит приблизительно (T/N) \*1.3, где T - время выполнения программы на одном процессоре, а 1.3 - 30% -ая поправка, включающая в себя проверку и контроль со стороны PSS и особенности аппаратуры.

Все вышеуказанные положения особенно важны, принимая во внимание исследовательский характер комплекса "Оболочка", что подразумевает проведение большого количества вычислений с использованием различных параметров оболочек, нагружений, закреплений, ребер и т.д. Поэтому даже небольшое ускорение процесса вычисления одного эксперимента будет ощутимой экономией времени при многократном повторе этого эксперимента.

## 4.1 Программа и результаты

Программа PSS была написана на языке C с использованием MPI. Таким образом, возможна компиляция исходного кода на различных операционных системах и архитектурах, что делает программу универсальной и кросс-платформенной[[4]](#footnote-4). Код программы представлен в Приложении 1.

После написания программы были получены серии результатов. Для каждого N (числа членов в разложении искомых функций) было проведено три эксперимента по замеру времени выполнения задачи:

Однопроцессорная система - один последовательный процесс;

Две однопроцессорных системы, связанных Ethernet патчкордом со скоростью обмена данных 100 Мбит - два параллельных процесса.

Одна однопроцессорная двухядерная система с общей памятью - два параллельных процесса.

В качестве тестовых систем были использованы следующие ПК:

Intel Core 2 Duo 2,0 Ггц, 2 Гб оперативной памяти;

Intel Xeon 2x2,66 Ггц, 4 Гб оперативной памяти.

Были получены следующие результаты:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Последовательный  процесс, сек | Две однопроцессорные –  по одному ядру, сек | Одна однопроцессорная -  два ядра, сек | Maple 12, сек |
| 1 | 0.051569 | 0.061439 | 0.042440 | 0.8 |
| 4 | 2.124557 | 0.779514 | 0.611919 | 2.9 |
| 9 | 5.022527 | 3.339019 | 3.080036 | 7.8 |
| 16 | 12.646657 | 10.744853 | 9.732358 | 13.0 |
| 25 | 31.078364 | 26.457036 | 23.840874 | 38.6 |
| 121 | 770.707305 | 644.396220 | 586.622127 | 952.3 |



## Заключение

При исследовании устойчивости подкрепленных оболочек с учетом геометрической и физической нелинейности при последовательном вычислении требуется достаточно большое время для расчета одного варианта - до нескольких часов. Расчеты, проведенные с помощью ПК Ansys, показали, что на подготовку входных данных и расчет варианта также требуется несколько часов. Таким образом, при распараллеливании процессов вычисления время расчета одного варианта существенно сокращается, что позволит проводить вычисления самой "затратной" задачи максимум в течение часа, а критические нагрузки при линейно-упругом деформировании в течение нескольких минут.

Работа докладывалась на седьмой всероссийской конференции "Математическое моделирование и краевые задачи (ММ-2010)", проходившей 3-6 июня 2010 в г. Самара. Получила одобрение и положительные отзывы от оргкомитета и участников конференции.

## Литература

1. Антонов А.С. Параллельное программирование с использованием технологии MPI - М.: Изд-во МГУ, 2004. - 71 с.
2. Бондаренко В.М., Бондаренко С.В. Инженерные методы нелинейной теории железобетона. - М.: Стройиздат, 1982. - 288 с.
3. Васильев А.Н. Самоучитель C++ с задачами и примерами - М.: Наука и Техника, 2010. - 480 с.
4. Воеводин В.В. Вычислительная математика и структура алгоритмов - М.: Изд-во МГУ, 2006. - 112 с.
5. Воеводин В.В. Математические модели и методы в параллельных процессах - М.: Наука. Гл. ред. физ. - мат. лит., 1986. - 296 с.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления - СПб.: БХВ-Петербург, 2002. - 608 с.
7. Воеводин Вл.В., Жуматий С.А. Вычислительное дело и кластерные системы - М.: Изд-во МГУ, 2007. - 150 с.
8. Жгутов В.М. Математические модели и алгоритмы исследования устойчивости пологих ребристых оболочек при учете различных свойств материала // Известия Орловского гос. техн. ун-та. Серия "Строительство, транспорт". - 2007. №4. - С. 20-23.
9. Ильюшин А.А. Пластичность. М.: Гостехиздат. 1948. - 376 с.
10. Карпов В.В. Математическое моделирование, алгоритмы исследования модели, вычислительный эксперимент в теории оболочек: учеб. пособие - СПб.: СПбГАСУ, 2006. - 330 с.
11. Карпов В.В., Баранова Д.А., Беркалиев Р.Т. Программный комплекс исследования устойчивости оболочек - СПб.: СПбГАСУ, 2009. - 102 с.
12. Карпов В.В., Сальников А.Ю. Вариационные методы и вариационные принципы механики при расчете строительных конструкций: учеб. пособие - СПб.: СПбГАСУ, 2009. - 75 с.
13. Керниган Б, Ритчи Д. Язык программирования C - М.: Вильямс, 2009. - 304 с.
14. Климанов В.И., Тимашев С.А. Нелинейные задачи подкрепленных оболочек. Свердловск: УНЦ АН СССР, 1985. - 291 с.
15. Петров В.В., Овчинников И.Г., Ярославский В.И. Расчет пластинок и оболочек из нелинейно-упругого материала. - Саратов: Изд-во Сарат. ун-та, 1976. - 136 с.
16. Хьюз К., Хьюз Т. Параллельное и распределенное программирование с использованием C++ - М.: Вильямс, 2004. - 672 с.

## Приложения

Приложение 1. Код программы

Main. cpp:

/\*

Main. cpp

Сизов А.С. ПМ5

2010 г.

\*/

// Подключение необходимых заголовочных файлов

#include "mpi. h" // библиотека mpi

#include <stdio. h>

/\*

stdio. h (от англ. standard input/output header -

стандартный заголовочный файл ввода/вывода) заголовочный файл

стандартной библиотеки языка Си, содержащий определения макросов,

константы и объявления функций и типов, используемых для различных

операций стандартного ввода и вывода. Функциональность унаследована

от "портативного пакета ввода/вывода" ("portable I/O package"),

написанного Майком Леском из Bell Labs в начале 1970-х.

Функции, объявленные в stdio. h, являются весьма популярными благодаря тому,

что являясь частью Стандартной библиотеки языка Си, они гарантируют работу

на любой платформе, поддерживающей Си. Приложения на отдельных платформах

могут, тем не менее, иметь причины для использования функций ввода/вывода

самой платформы вместо функций stdio. h.

\*/

#include <stdlib. h>

/\*

stdlib. h - заголовочный файл стандартной библиотеки

общего назначения языка Си, который содержит в себе функции,

занимающиеся выделением памяти, контроль процесса выполнения программы,

преобразования типов и другие.

\*/

#include <math. h>

/\*

math. h - заголовочный файл стандартной библиотеки языка программирования С,

разработанный для выполнения простых математических операций. Большинство

функций привлекают использование чисел с плавающей точкой. Все эти функции

принимают double, если не определено иначе. Для работы с типами float и

long double используются функции с постфиксами f и l соответственно.

Все функции, принимающие или возвращающие угол, работают с радианами.

\*/

#include "matrix. h"

/\*

matrix. h - заголовочный файл библиотеки, содержащей определения

класса матрицы matrix, использованного в дальнейшем в программе.

Содержит объявление класса, методы для работы с ним, такие как:

произведение матрицы на матрицу;

сложение матрицы с матрицей;

выделение памяти под матрицу;

выгрузка данных;

загрузка данных;

копирование матрицы;

сравнение полученной единичной матрицы с эталоном единичной матрицы;

обращение матрицы.

\*/

// Объявление переменных

double \*X1, \*X2, \*X3, \*Y1, \*Y2, \*Y3; // Коэффициенты аппроксимирующих // функций

double \*C1, \*C2; // Коэффициенты системы

int N, n, m, E;

double q, h, mu, r, Pi, mu1, Kx, Ky, R1, R2, W;

double a,b;

/\*

N - количество членов в приближении по методу Ритца;

n - количество ребер по x;

m - количество ребер по y;

E - модуль упругости материала;

q - поперечная равномерно-распределенная нагрузка;

h - толщина оболочки;

mu - коэффициент Пуассона;

r - ширина ребер;

Pi - число Пи;

R1 - радиус кривизны по x;

R2 - радиус кривизны по y;

a - длина оболочки по x;

b - длина оболочки по y;

\*/

// Процедуры вычисления производных функций

double sin\_0 (double k, double x)

{

return sin (k \* x);

}

double sin\_1 (double k, double x) // первая производная

{

return k \* cos (k \* x);

}

double sin\_2 (double k, double x) // вторая производная

{

return - k \* k \* sin (k \* x);

}

double Fx (int k, int i, int j, double arg) // выборка произведений

{ // аппроксимирующих

switch (k) // функций для х

{

case 1:

return sin\_1 (X1 [i], arg) \* sin\_1 (X1 [j], arg);

case 2:

return sin\_0 (X1 [i], arg) \* sin\_0 (X1 [j], arg);

case 3:

return sin\_1 (X1 [i], arg) \* sin\_0 (X2 [j], arg);

case 4:

return sin\_0 (X1 [i], arg) \* sin\_1 (X2 [j], arg);

case 5:

return sin\_1 (X1 [i], arg) \* sin\_0 (X3 [j], arg);

case 6:

return sin\_0 (X2 [i], arg) \* sin\_0 (X2 [j], arg);

case 7:

return sin\_1 (X2 [i], arg) \* sin\_1 (X2 [j], arg);

case 8:

return sin\_0 (X2 [i], arg) \* sin\_0 (X3 [j], arg);

case 9:

return sin\_0 (X3 [i], arg) \* sin\_1 (X1 [j], arg);

case 10:

return sin\_0 (X3 [i], arg) \* sin\_0 (X3 [j], arg);

case 11:

return sin\_2 (X3 [i], arg) \* sin\_2 (X3 [j], arg);

case 12:

return sin\_2 (X3 [i], arg) \* sin\_0 (X3 [j], arg);

case 13:

return sin\_1 (X3 [j], arg) \* sin\_1 (X3 [j], arg);

case 14:

return sin\_0 (X3 [j], arg);

default:

return 0;

}

}

double simpsonFx (double a, double b, int k, int i, int j)

{ // метод Симпсона для х

int n = 1000;

double locI, h, xi, s1 = 0.0, s2 = 0.0, y [10000] ;

int f;

h = (b - a) / n;

xi = a;

for (f = 0; f <= 2 \* n; f++)

{

y [f] = Fx (k, i,j,xi);

xi = xi + h/2;

}

locI = y [0] + y [2\*n] ;

for (f = 2; f < 2\*n; f = f + 2)

s1 = s1 + y [f] ;

for (f = 1; f < 2\*n; f = f + 2)

s2 = s2 + y [f] ;

locI = (locI + 2.0 \* s1 + 4.0 \* s2) \* (h / 6);

return (locI);

}

double Fy (int k, int i, int j, double arg) // выборка произведений

{ // аппроксимирующих

switch (k) // функций для y

{

case 1:

return sin\_0 (Y1 [i], arg) \* sin\_0 (Y1 [j], arg);

case 2:

return sin\_1 (Y1 [i], arg) \* sin\_1 (Y1 [j], arg);

case 3:

return sin\_0 (Y1 [i], arg) \* sin\_1 (Y2 [j], arg);

case 4:

return sin\_1 (Y1 [i], arg) \* sin\_0 (Y2 [j], arg);

case 5:

return sin\_0 (Y1 [i], arg) \* sin\_0 (Y3 [j], arg);

case 6:

return sin\_1 (Y2 [i], arg) \* sin\_1 (Y2 [j], arg);

case 7:

return sin\_0 (Y2 [i], arg) \* sin\_0 (Y2 [j], arg);

case 8:

return sin\_1 (Y2 [i], arg) \* sin\_0 (Y3 [j], arg);

case 9:

return sin\_0 (Y3 [i], arg) \* sin\_0 (Y1 [j], arg);

case 10:

return sin\_0 (Y3 [i], arg) \* sin\_0 (Y3 [j], arg);

case 11:

return sin\_2 (Y3 [i], arg) \* sin\_2 (Y3 [j], arg);

case 12:

return sin\_2 (Y3 [i], arg) \* sin\_0 (Y3 [j], arg);

case 13:

return sin\_1 (Y3 [i], arg) \* sin\_1 (Y3 [j], arg);

case 14:

return sin\_0 (Y3 [j], arg);

default:

return 0;

}

}

double simpsonFy (double a, double b, int k, int i, int j)

{ // метод Симпсона для y

int n = 1000;

double locI, h, xi, s1 = 0.0, s2 = 0.0, y [10000] ;

int f;

h = (b - a) / n;

xi = a;

for (f = 0; f <= 2 \* n; f++)

{

y [f] = Fy (k, i,j,xi);

xi = xi + h/2;

}

locI = y [0] + y [2\*n] ;

for (f = 2; f < 2\*n; f = f + 2)

s1 = s1 + y [f] ;

for (f = 1; f < 2\*n; f = f + 2)

s2 = s2 + y [f] ;

locI = (locI + 2.0 \* s1 + 4.0 \* s2) \* (h / 6);

return (locI);

}

double C (int k, int i, int j) // коэффициэнты в ФПЭД

{

switch (k)

{

case 1:

return 2\*h\*simpsonFx (0.0,a,1, i,j) \*simpsonFy (0.0,a,1, i,j) +2\*mu1\*h\*simpsonFx (0.0,a,2, i,j) \*simpsonFy (0.0,a,2, i,j);

case 2:

return 2\*h\*mu\*simpsonFx (0.0,a,3, i,j) \*simpsonFy (0.0,a,3, i,j) +2\*mu1\*h\*simpsonFx (0.0,a,4, i,j) \*simpsonFy (0.0,a,4, i,j);

case 3:

return - 2\*h\* (Kx+mu\*Ky) \*simpsonFx (0.0,a,5, i,j) \*simpsonFy (0.0,a,5, i,j);

case 4:

return 2\*h\*mu\*simpsonFx (0.0,a,3, i,j) \*simpsonFy (0.0,a,3, i,j) +2\*mu1\*h\*simpsonFx (0.0,a,4, i,j) \*simpsonFy (0.0,a,4, i,j);

case 5:

return 2\*h\*simpsonFx (0.0,a,6, i,j) \*simpsonFy (0.0,a,6, i,j) +2\*mu1\*h\*simpsonFx (0.0,a,7, i,j) \*simpsonFy (0.0,a,7, i,j);

case 6:

return - 2\*h\* (mu\*Kx+Ky) \*simpsonFx (0.0,a,8, i,j) \*simpsonFy (0.0,a,8, i,j);

case 7:

return - 2\*h\* (Kx+mu\*Ky) \*simpsonFx (0.0,a,9, i,j) \*simpsonFy (0.0,a,9, i,j);

case 8:

return - 2\*h\* (mu\*Kx+Ky) \*simpsonFx (0.0,a,8, i,j) \*simpsonFy (0.0,a,8, i,j);

case 9:

return 2\*h\* (Kx\*Kx+2\*mu\*Kx\*Ky+Ky\*Ky) \*simpsonFx (0.0,a,10, i,j) \*simpsonFy (0.0,a,10, i,j) +pow (h,3) /12

\* (2\*simpsonFx (0.0,a,11, i,j) \*simpsonFy (0.0,a,10, i,j) +2\*mu\*simpsonFx (0.0,a,12, i,j)

\*simpsonFy (0.0,a,12, i,j) +2\*mu\*simpsonFx (0.0,a,12, i,j) \*simpsonFy (0.0,a,12, i,j) +2\*simpsonFx (0.0,a,10, i,j)

\*simpsonFy (0.0,a,11, i,j) +8\*mu1\*simpsonFx (0.0,a,13, i,j) \*simpsonFy (0.0,a,13, i,j));

case 10:

return 2\* (1-pow (mu,2)) \*simpsonFx (0.0,a,14, i,j) \*simpsonFy (0.0,a,14, i,j);

default:

return 0;

}

}

int main (int argc,char \*argv [])

{

// объявление переменных.

int myid, numprocs;

int namelen;

char processor\_name [MPI\_MAX\_PROCESSOR\_NAME] ;

double startwtime = 0.0, endwtime;

int rc;

MPI\_Status status;

rc = MPI\_Init (&argc,&argv);

rc|= MPI\_Comm\_size (MPI\_COMM\_WORLD,&numprocs);

rc|= MPI\_Comm\_rank (MPI\_COMM\_WORLD,&myid);

if (rc! = 0)

printf ("error initializing MPI and obtaining task ID information\n");

MPI\_Get\_processor\_name (processor\_name,&namelen);

fprintf (stdout,"Process%d of%d is on%s\n",

myid, numprocs, processor\_name);

fflush (stdout);

// функция начала замера времени вычисления.

if (myid == 0) {

startwtime = MPI\_Wtime ();

}

N = 4; // количество членов

a = 1, b = 1;

n = 6, m = 6, E = 21000;

q = 0.0037, h = 0.001, mu = 0.3, r = 0.2;

Pi = 3.14;

int i, j;

double rv;

bool xyz;

R1 = 440\*h;

R2 = 440\*h;

mu1 = (1-mu) /2;

Kx = 1/ (double) R1;

Ky = 1/ (double) R2;

// выделение памяти под массивы для аппроксимирующих функций

X1 = (double\*) malloc (N \* sizeof (double));

X2 = (double\*) malloc (N \* sizeof (double));

X3 = (double\*) malloc (N \* sizeof (double));

Y1 = (double\*) malloc (N \* sizeof (double));

Y2 = (double\*) malloc (N \* sizeof (double));

Y3 = (double\*) malloc (N \* sizeof (double));

int sqrtN = pow (N, 0.5);

/\*

вычисление коэффициентов аппроксимирующих функций на

нескольких процессах.

\*/

for (i = 1; i <= sqrtN; i++)

{

for (j = 1; j <= sqrtN; j++)

{

if (myid == 0) {

X1 [sqrtN \* (i - 1) + j - 1] = 2 \* i \* Pi;

printf ("X1 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,X1 [sqrtN \* (i - 1) + j - 1]);

X2 [sqrtN \* (i - 1) + j - 1] = (2 \* i - 1) \* Pi;

printf ("X2 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,X2 [sqrtN \* (i - 1) + j - 1]);

X3 [sqrtN \* (i - 1) + j - 1] = (2 \* i - 1) \* Pi;

printf ("X3 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,X3 [sqrtN \* (i - 1) + j - 1]);

}

if (myid == 1) {

Y1 [sqrtN \* (i - 1) + j - 1] = (2 \* j - 1) \* Pi;

printf ("Y1 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,Y1 [sqrtN \* (i - 1) + j - 1]);

Y2 [sqrtN \* (i - 1) + j - 1] = 2 \* j \* Pi;

printf ("Y2 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,Y2 [sqrtN \* (i - 1) + j - 1]);

Y3 [sqrtN \* (i - 1) + j - 1] = (2 \* j - 1) \* Pi;

printf ("Y3 [%d] =%.3f\n",sqrtN \* (i - 1) + j - 1,Y3 [sqrtN \* (i - 1) + j - 1]);

}

}

}

/\*

пересылка результатов вычислений на "головную" машину

\*/

if (myid == 1) {

MPI\_Send (Y1, N, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD);

MPI\_Send (Y2, N, MPI\_DOUBLE, 0, 2, MPI\_COMM\_WORLD);

MPI\_Send (Y3, N, MPI\_DOUBLE, 0, 3, MPI\_COMM\_WORLD);

MPI\_Recv (X1, N, MPI\_DOUBLE, 0, 4, MPI\_COMM\_WORLD,

&status);

MPI\_Recv (X2, N, MPI\_DOUBLE, 0, 5, MPI\_COMM\_WORLD,

&status);

MPI\_Recv (X3, N, MPI\_DOUBLE, 0, 6, MPI\_COMM\_WORLD,

&status);

}

if (myid == 0) {

MPI\_Recv (Y1, N, MPI\_DOUBLE, 1, 1, MPI\_COMM\_WORLD,

&status);

MPI\_Recv (Y2, N, MPI\_DOUBLE, 1, 2, MPI\_COMM\_WORLD,

&status);

MPI\_Recv (Y3, N, MPI\_DOUBLE, 1, 3, MPI\_COMM\_WORLD,

&status);

MPI\_Send (X1, N, MPI\_DOUBLE, 1, 4, MPI\_COMM\_WORLD);

MPI\_Send (X2, N, MPI\_DOUBLE, 1, 5, MPI\_COMM\_WORLD);

MPI\_Send (X3, N, MPI\_DOUBLE, 1, 6, MPI\_COMM\_WORLD);

}

// вывод времени вычисления аппрокс. функций

if (myid == 0) {

endwtime = MPI\_Wtime ();

printf ("\napp func clock time =%f\n", endwtime-startwtime);

}

printf ("\n - --------------- - BEGIN - -----------------\n");

/\*

выделение памяти под массивы коэффициентов ФПЭД

и вычисление их на разных процессах.

\*/

C1 = (double\*) malloc (3 \* N \* 3 \* N \* sizeof (double));

C2 = (double\*) malloc (3 \* N \* 3 \* N \* sizeof (double));

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

// обнуление всех значений для удобства пересылки

C1 [i\*N+j] =0;

C1 [i\*N+N+j] =0;

C1 [i\*N+2\*N+j] =0;

C1 [N+i\*N+j] =0;

C1 [N+i\*N+N+j] =0;

C1 [N+i\*N+2\*N+j] =0;

C1 [2\*N+i\*N+j] =0;

C1 [2\*N+i\*N+N+j] =0;

C1 [2\*N+i\*N+2\*N+j] =0;

if (myid == 0) {

C1 [N\*i+j] =C (1, i,j);

C1 [N\*i+N+j] =C (2, i,j);

C1 [N\*i+2\*N+j] =C (3, i,j);

C1 [N+N\*i+j] =C (4, i,j);

C1 [N+N\*i+N+j] =C (5, i,j);

}

if (myid == 1) {

C1 [N+N\*i+2\*N+j] =C (6, i,j);

C1 [2\*N+N\*i+j] =C (7, i,j);

C1 [2\*N+N\*i+N+j] =C (8, i,j);

C1 [2\*N+N\*i+2\*N+j] =C (9, i,j);

}

}

}

// пересылка массивов на "головную" машину

if (myid == 1) {

MPI\_Send (C1, 3\*N\*3\*N, MPI\_DOUBLE, 0, 7, MPI\_COMM\_WORLD);

}

if (myid == 0) {

MPI\_Recv (C2, 3\*N\*3\*N, MPI\_DOUBLE, 1, 7, MPI\_COMM\_WORLD, &status);

printf ("\n\nC2 [1]%.3f\n",C2 [0]);

}

printf ("\n - --------------- - END - -----------------\n");

if (myid == 0) {

matrix <double> M1 (3\*N,3\*N);

printf ("-------------------- - BEGIN FIRST - -----------------\n");

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

M1. setvalue (i,j,C (1, i,j));

printf ("C1 [%d,%d]: =%.5f ", i,j,C (1, i,j));

M1. setvalue (i,N+j,C (2, i,j));

printf ("C2 [%d,%d]: =%.5f ", i,N+j,C (2, i,j));

M1. setvalue (i,2\*N+j,C (3, i,j));

printf ("C3 [%d,%d]: =%.5f\n", i,2\*N+j,C (3, i,j));

M1. setvalue (N+i,j,C (4, i,j));

printf ("C4 [%d,%d]: =%.5f ",N+i,j,C (4, i,j));

M1. setvalue (N+i,N+j,C (5, i,j));

printf ("C5 [%d,%d]: =%.5f ",N+i,N+j,C (5, i,j));

M1. setvalue (N+i,2\*N+j,C (6, i,j));

printf ("C6 [%d,%d]: =%.5f\n",N+i,2\*N+j,C (6, i,j));

M1. setvalue (2\*N+i,j,C (7, i,j));

printf ("C7 [%d,%d]: =%.5f ",2\*N+i,j,C (7, i,j));

M1. setvalue (2\*N+i,N+j,C (8, i,j));

printf ("C8 [%d,%d]: =%.5f ",2\*N+i,N+j,C (8, i,j));

M1. setvalue (2\*N+i,2\*N+j,C (9, i,j));

printf ("C9 [%d,%d]: =%.5f\n",2\*N+i,2\*N+j,C (9, i,j));

}

}

printf ("-------------------- - END FIRST - -----------------\n");

printf ("-------------------- - BEGIN SECOND - -----------------\n");

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

M1. setvalue (i,j,C1 [N\*i+j]);

printf ("C1 [%d,%d]: =%.5f ", i,j,C1 [N\*i+j]);

M1. setvalue (i,N+j,C1 [N\*i+N+j]);

printf ("C2 [%d,%d]: =%.5f ", i,N+j,C1 [N\*i+N+j]);

M1. setvalue (i,2\*N+j,C1 [N\*i+2\*N+j]);

printf ("C3 [%d,%d]: =%.5f\n", i,2\*N+j,C1 [N\*i+2\*N+j]);

M1. setvalue (N+i,j,C1 [N+i+j]);

printf ("C4 [%d,%d]: =%.5f ",N+i,j,C1 [N+N\*i+j]);

M1. setvalue (N+i,N+j,C1 [N+N\*i+N+j]);

printf ("C5 [%d,%d]: =%.5f ",N+i,N+j,C1 [N+N\*i+N+j]);

M1. setvalue (N+i,2\*N+j,C2 [N+N\*i+2\*N+j]);

printf ("C6 [%d,%d]: =%.5f\n",N+i,2\*N+j,C2 [N+N\*i+2\*N+j]);

M1. setvalue (2\*N+i,j,C2 [2\*N+N\*i+N+j]);

printf ("C7 [%d,%d]: =%.5f ",2\*N+i,j,C2 [2\*N+N\*i+N+j]);

M1. setvalue (2\*N+i,N+j,C2 [2\*N+N\*i+N+j]);

printf ("C8 [%d,%d]: =%.5f ",2\*N+i,N+j,C2 [2\*N+N\*i+N+j]);

M1. setvalue (2\*N+i,2\*N+j,C2 [2\*N+N\*i+2\*N+j]);

printf ("C9 [%d,%d]: =%.5f\n",2\*N+i,2\*N+j,C2 [2\*N+N\*i+2\*N+j]);

}

}

printf ("-------------------- - END SECOND - -----------------\n");

// заполнение массивов свободных членов

matrix <double> Pow (3\*N,3\*N);

for (i=0; i < Pow. getactualsize (); i++)

{

for (j=0; j< Pow. getactualsize (); j++)

{

Pow. setvalue (i,j,0);

}

}

for (i = 0; i < N; i++)

{

Pow. setvalue (i,0,0);

Pow. setvalue (N+i,0,0);

Pow. setvalue (2\*N+i,0,q/E\*C (10, i,0));

}

printf ("\n\n\nM1: ");

for (i=0; i < M1. getactualsize (); i++)

{

printf ("\n%d: ", i);

for (j=0; j< M1. getactualsize (); j++)

{

M1. getvalue (i,j,rv,xyz);

printf ("%.10f ",rv);

}

}

// выделение памяти под матрицы

matrix <double> M2 (3\*N,3\*N);

matrix <double> M3 (3\*N,3\*N);

// копирование матрицы коэффициентов

M2. copymatrix (M1);

// обращение матрицы

M1. invert ();

// произведение матриц

M3. settoproduct (M1,M2);

// сравнение полученной единичной матрицы с эталоном единичной матрицы

M3.comparetoidentity ();

printf ("\n\n\ninverse: ");

for (i=0; i < M3. getactualsize (); i++)

{

printf ("\n%d: ", i);

for (j=0; j< M3. getactualsize (); j++)

{

M3. getvalue (i,j,rv,xyz);

printf ("%.10f ",rv);

}

}

for (i=0; i < M1. getactualsize (); i++)

{

printf ("\n%d: ", i);

for (j=0; j< M1. getactualsize (); j++)

{

M1. getvalue (i,j,rv,xyz);

printf ("%.10f ",rv);

}

}

// выделение памяти для матрицы результатов

matrix <double> Ans (3\*N,3\*N);

Ans. settoproduct (M1,Pow);

printf ("\nanswer \n");

for (i=0; i < Ans. getactualsize (); i++)

{

printf ("%d: ", i);

Ans. getvalue (i,0,rv,xyz);

printf ("%.10f ", rv);

printf ("\n");

}

// вывод результата

W = 0;

for (i = 0; i < N; i++)

{

Ans. getvalue (2\*N+i,0,rv,xyz);

printf ("!!%.10f%.10f%.10f\n", rv, X3 [i], Y3 [i]);

W += rv \* sin\_0 (X3 [i],a/2) \* sin\_0 (Y3 [i],a/2);

}

printf ("W: =%.10f", W);

// вывод времени вычисления программы

endwtime = MPI\_Wtime ();

printf ("\nwall clock time =%f\n", endwtime-startwtime);

fflush (stdout);

}

MPI\_Finalize ();

return 0;

}

Matrix. h:

#ifndef \_\_mjdmatrix\_h

#define \_\_mjdmatrix\_h

template <class D> class matrix{

int maxsize;

int actualsize;

D\* data;

void allocate () {

delete [] data;

data = new D [maxsize\*maxsize] ;

};

matrix () {};

matrix (int newmaxsize) {matrix (newmaxsize,newmaxsize); };

public:

matrix (int newmaxsize, int newactualsize) {

if (newmaxsize <= 0) newmaxsize = 5;

maxsize = newmaxsize;

if ( (newactualsize <= newmaxsize) && (newactualsize>0))

actualsize = newactualsize;

else

actualsize = newmaxsize;

data = 0;

allocate ();

};

~matrix () { delete [] data; };

void settoproduct (matrix& left, matrix& right) {

actualsize = left. getactualsize ();

if (maxsize < left. getactualsize ()) {

maxsize = left. getactualsize ();

allocate ();

}

for (int i = 0; i < actualsize; i++)

for (int j = 0; j < actualsize; j++) {

D sum = 0.0;

D leftvalue, rightvalue;

bool success;

for (int c = 0; c < actualsize; c++) {

left. getvalue (i,c,leftvalue,success);

right. getvalue (c,j,rightvalue,success);

sum += leftvalue \* rightvalue;

}

setvalue (i,j,sum);

}

}

void combine (matrix& left, matrix& right) {

actualsize = left. getactualsize ();

if (maxsize < left. getactualsize ()) {

maxsize = left. getactualsize ();

allocate ();

}

for (int i = 0; i < actualsize; i++)

for (int j = 0; j < actualsize; j++) {

D sum = 0.0;

D leftvalue, rightvalue;

bool success;

left. getvalue (i,j,leftvalue,success);

right. getvalue (i,j,rightvalue,success);

sum = leftvalue + rightvalue;

setvalue (i,j,sum);

}

}

void setactualsize (int newactualsize) {

if (newactualsize > maxsize)

{

maxsize = newactualsize;

allocate ();

}

if (newactualsize >= 0) actualsize = newactualsize;

};

int getactualsize () { return actualsize; };

void getvalue (int row, int column, D& returnvalue, bool& success) {

if ( (row>=maxsize) || (column>=maxsize)

|| (row<0) || (column<0))

{ success = false;

return; }

returnvalue = data [row \* maxsize + column] ;

success = true;

};

bool setvalue (int row, int column, D newvalue) {

if ( (row >= maxsize) || (column >= maxsize)

|| (row<0) || (column<0)) return false;

data [row \* maxsize + column] = newvalue;

return true;

};

void dumpMatrixValues () {

bool xyz;

double rv;

for (int i=0; i < actualsize; i++)

{

std:: cout << "i=" << i << ": ";

for (int j=0; j< actualsize; j++)

{

M. getvalue (i,j,rv,xyz);

std:: cout << rv << " ";

}

std:: cout << std:: endl;

}

};

void comparetoidentity () {

int worstdiagonal = 0;

D maxunitydeviation = 0.0;

D currentunitydeviation;

for (int i = 0; i < actualsize; i++) {

currentunitydeviation = data [i\*maxsize+i] - 1.;

if (currentunitydeviation < 0.0) currentunitydeviation \*= - 1.;

if (currentunitydeviation > maxunitydeviation) {

maxunitydeviation = currentunitydeviation;

worstdiagonal = i;

}

}

int worstoffdiagonalrow = 0;

int worstoffdiagonalcolumn = 0;

D maxzerodeviation = 0.0;

D currentzerodeviation;

for (int i = 0; i < actualsize; i++) {

for (int j = 0; j < actualsize; j++) {

if (i == j) continue;

currentzerodeviation = data [i\*maxsize+j] ;

if (currentzerodeviation < 0.0) currentzerodeviation \*= - 1.0;

if (currentzerodeviation > maxzerodeviation) {

maxzerodeviation = currentzerodeviation;

worstoffdiagonalrow = i;

worstoffdiagonalcolumn = j;

}

}

}

printf ("Worst diagonal value deviation from unity:%0.5f at row/column%0.3f\n", maxunitydeviation, worstdiagonal);

printf ("Worst off-diagonal value deviation from zero:%0.5f at row%0.3f, column%0.3f\n", maxzerodeviation, worstoffdiagonalrow,worstoffdiagonalcolumn);

};

void copymatrix (matrix& source) {

actualsize = source. getactualsize ();

if (maxsize < source. getactualsize ()) {

maxsize = source. getactualsize ();

allocate ();

}

for (int i = 0; i < actualsize; i++)

for (int j = 0; j < actualsize; j++) {

D value;

bool success;

source. getvalue (i,j,value,success);

data [i\*maxsize+j] = value;

}

};

void invert () {

if (actualsize <= 0) return;

if (actualsize == 1) return;

for (int i=1; i < actualsize; i++) data [i] /= data [0] ;

for (int i=1; i < actualsize; i++) {

for (int j=i; j < actualsize; j++) {

D sum = 0.0;

for (int k = 0; k < i; k++)

sum += data [j\*maxsize+k] \* data [k\*maxsize+i] ;

data [j\*maxsize+i] - = sum;

}

if (i == actualsize-1) continue;

for (int j=i+1; j < actualsize; j++) {

D sum = 0.0;

for (int k = 0; k < i; k++)

sum += data [i\*maxsize+k] \*data [k\*maxsize+j] ;

data [i\*maxsize+j] =

(data [i\*maxsize+j] -sum) / data [i\*maxsize+i] ;

}

}

for (int i = 0; i < actualsize; i++) // invert L

for (int j = i; j < actualsize; j++) {

D x = 1.0;

if (i! = j) {

x = 0.0;

for (int k = i; k < j; k++)

x - = data [j\*maxsize+k] \*data [k\*maxsize+i] ;

}

data [j\*maxsize+i] = x / data [j\*maxsize+j] ;

}

for (int i = 0; i < actualsize; i++)

for (int j = i; j < actualsize; j++) {

if (i == j) continue;

D sum = 0.0;

for (int k = i; k < j; k++)

sum += data [k\*maxsize+j] \* ( (i==k)? 1.0: data [i\*maxsize+k]);

data [i\*maxsize+j] = - sum;

}

for (int i = 0; i < actualsize; i++)

for (int j = 0; j < actualsize; j++) {

D sum = 0.0;

for (int k = ( (i>j)? i: j); k < actualsize; k++)

sum += ( (j==k)? 1.0: data [j\*maxsize+k]) \*data [k\*maxsize+i] ;

data [j\*maxsize+i] = sum;

}

};

};

#endif

1. OpenMP (Open Multi-Processing) – набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с единой памятью на языках C, C++ и Fortran. [↑](#footnote-ref-1)
2. GIMPS (Great Internet Mersenne Prime Search) — широкомасштабный проект [распределённых вычислений](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) по поиску [простых](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) [чисел Мерсенна](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE_%D0%9C%D0%B5%D1%80%D1%81%D0%B5%D0%BD%D0%BD%D0%B0). [↑](#footnote-ref-2)
3. Эмуля́ция ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *emulation*) — воспроизведение программными или аппаратными средствами либо их комбинацией работы других программ или устройств. [↑](#footnote-ref-3)
4. Кроссплатформенное программное обеспечение — [программное обеспечение](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), работающее более чем на одной аппаратной платформе и/или [операционной системе](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). [C](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8_%28%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29) — кроссплатформенный язык на уровне компиляции, то есть для этого языка есть компиляторы под различные платформы. [↑](#footnote-ref-4)