МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**Государственное образовательное учреждение
высшего профессионального образования
«Ростовский государственный университет»**

**Механико-математический факультет
Кафедра алгебры и дискретной математики**

Магистерская диссертация

Автоматическая генерация MPI кода в открытой распараллеливающей системе

Студента
2 года магистратуры — Науменко С. А.

Научный руководитель
к.ф.-м.н.,
доц. каф. АДМ мехмата РГУ — Штейнберг Б. Я.

Рецензенты
к.ф.-м.н.,
доц. каф. ИВЭ мехмата РГУ — Савельев В. А.

к.ф.-м.н.,
доц. каф. ВМиМФ мехмата РГУ — Барковский Ю. С.

**Ростов-на-Дону
2005 г.**

Оглавление:

1. Введение. 3
2. Открытая распараллеливающая система. 6
3. Общая информация о библиотеке MPI. 10
4. Базовые функции библиотеки MPI. 14
5. Информационные зависимости в программе. 17
6. Дополнительные оптимизирующие преобразования. 22
6.1. Развёртка программных циклов. 22
6.2. Гнездование программных циклов. 25
7. Автоматическая генерация MPI кода для циклов, не содержащих информационные зависимости. 27
8. Автоматическая генерация MPI кода для циклов, содержащих зависимости. 30
8.1. Циклы, содержащие антизависимости. 30
8.2. Циклы, содержащие входные зависимости. 32
8.3. Циклы, содержащие истинные зависимости. 33
8.4. Циклы, содержащие выходные зависимости. 34
9. Заключение. 35
10. Литература. 36

1. Введение.

В проделанной работе разработан генератор MPI-кода, который автоматически генерирует параллельный программный код на языке C, использующий функции из библиотеки MPI. Результирующий код предназначен для выполнения на параллельных кластерных системах поддерживаемых библиотекой MPI. Автоматическая генерация кода является заключительным этапом обработки исходного программного кода Открытой распарралеливающей системой (далее ОРС). ОРС разрабатывается студентами и аспирантами кафедры алгебры и дискретной математики. Эта система предназначена для автоматического распараллеливания программ с процедурных языков программирования (ФОРТРАН, Паскаль, Си) на параллельные суперкомпьютеры. К суперкомпьютерам относятся и кластерные системы.

Основная идея программирования для кластера с использованием библиотеки MPI заключается в том, что программа выполняется одновременно на всех узлах кластера (компьютерах), но с разными данными. Поэтому последовательные участки программы выполняются на всех узлах, выполняя одну и ту же работу. А вот программные циклы можно выполнять так, чтобы каждый узел делал только свою работу. Как простейший пример, раздать каждому узлу для выполнения по одной итерации цикла. Все информационные зависимости, которые могут быть в таком цикле необходимо разрешать пересылками данных, для чего используются различные функции библиотеки MPI. Естественно, для достижения оптимального быстродействия, нужно стремиться уменьшить количество пересылок, а значит и информационных зависимостей. Для этого применяются различные преобразования из библиотеки преобразований ОРС.

Генератор кода, разработанный в данной работе, по внутреннему представлению выдаёт программу на языке C, использующую библиотеку

MPI. Эта программа может быть скомпилирована любым компилятором языка С, для которого есть библиотека MPI. Таких компиляторов довольно много, например, свободно распространяемый компилятор *gcc* [11]. Библиотека MPI была выбрана исходя из того, что она предоставляет очень удобный интерфейс пересылки сообщений между процессами параллельной программы выполняющейся на кластере, а также она довольно часто используется при программировании для кластерных систем.

Пример 1. Приведём пример работы генератора кода. В качестве входного файла возьмём код программы на языке C, которая в цикле заполняет элементы массива значением полинома в точке равной значению счётчика цикла.

```
int main(int argc, char **argv)
{
      int i, a[10], b[12];
      for(i = 1; i < 10; i++)
      {
            a[i] = b[i+2] * i + i * i;
      }
      return 0;
}
```

На выходе генератора кода получим текст программы на языке C, уже подготовленный для выполнения на кластере.

```
#include <iostream>
#include "mpi.h"

int MPIPRODEX_myid = 0, MPIPRODEX_numprocs = 0;

int main(int argc, char **argv)
{
  int i, a[10], b[12];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &MPIPRODEX_numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &MPIPRODEX_myid);

  if (MPIPRODEX_myid == 0)
  {
    for (i = 1; i < MPIPRODEX_numprocs; i++)
    {
      int MPIPRODEX_left = i * (10 - 1) / MPIPRODEX_numprocs + 2;
```

```
      int MPIPRODEX_count = (i + 1) * (10 - 1) / MPIPRODEX_numprocs + 2 –
          MPIPRODEX_left;
      MPI_Send(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
          MPI_COMM_WORLD);
    }
  }
  else
  {
    int MPIPRODEX_left = MPIPRODEX_myid * (10 - 1) / MPIPRODEX_numprocs + 2;
    int MPIPRODEX_count = (MPIPRODEX_myid + 1)*(10-1) / MPIPRODEX_numprocs +
      2 – MPIPRODEX_left;
    MPI_Recv(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0, MPI_COMM_WORLD,
        0);
  }

  for (i = 1 + MPIPRODEX_myid * (10 - 1) / MPIPRODEX_numprocs;
        i < 1 + (MPIPRODEX_myid + 1) * (10 - 1) / MPIPRODEX_numprocs; i++)
  {
    a[i] = b[i+2] * i + i * i;
  }

  if (MPIPRODEX_myid == 0)
  {
    for (i = 1; i < MPIPRODEX_numprocs; i++)
    {
      int MPIPRODEX_left = i * (10 - 1) / MPIPRODEX_numprocs;
      int MPIPRODEX_count = (i + 1) * (10 - 1) / MPIPRODEX_numprocs –
          MPIPRODEX_left;
      MPI_Recv(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
          MPI_COMM_WORLD, 0);
    }
  }
  else
  {
    int MPIPRODEX_left = MPIPRODEX_myid*(10-1) / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (MPIPRODEX_myid + 1)*(10-1) / MPIPRODEX_numprocs –
        MPIPRODEX_left;
    MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0,
        MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

2. Открытая распараллеливающая система.

Актуальность создания ОРС объясняется тем, что постоянно нарастает необходимость переносить уже использующиеся программы на параллельные суперкомпьютеры. Распараллеливающие компиляторы позволяют довольно быстро, без привлечения специалистов параллельного программирования, это делать.

ОРС состоит из нескольких частей: парсера, библиотеки преобразований и генератора кода. Парсер, используя грамматику входного языка, преобразует исходный текст программы во внутреннее представление. Во внутреннем представлении программа выглядит как набор сложных древовидных структур. Они описывают типы переменных, сами переменные, а также операторы и операции языка. Что характерно для внутреннего представления, его легко можно перевести обратно в исходный текст, а также над ним легко совершать такие редакторские операции как удаление, вставка, копирование и замена. Для разработки ОРС был выбран язык С++.

Библиотека преобразований это группа модулей, каждый из которых представляет некоторое преобразование над внутренним представлением. Преобразования, используя простейшие редакторские операции над внутренним представлением, изменяют программу, либо фрагмент программы. Основная задача преобразований улучшение некоторых качеств программы: либо повышение быстродействия, либо уменьшение объема требуемой памяти, либо повышение точности вычислений, либо получение исходного текста на другом языке программирования (для последующего использования в какой-то программной среде).

Преобразования, задачей которых является выявление и, если это возможно, улучшение параллелизма программы (то есть увеличение эффективности её работы на суперкомпьютере), называются распараллеливающими. Так как существует множество различных семейств суперкомпьютерных платформ, то и распараллеливающие преобразования

бывают специфичными для какого-то одного семейства. В связи с чем, некоторые преобразования выполняют прямо противоположную работу. Например, преобразование разрезания цикла и преобразование слияния циклов являются взаимобратными. По этому целесообразность применения того или иного преобразования может зависеть от целевой платформы.

Наряду с распараллеливающими преобразованиями, существуют преобразования оптимизирующие, которые улучшают быстродействие программы и на обычных последовательных машинах.

Так как все преобразования работают с внутренним представлением, то нет необходимости отслеживать синтаксическую корректность преобразованных фрагментов программы, но в тоже время каждое преобразование должно проверять эквивалентность исходного и преобразованного фрагментов. Для решения этой задачи используется анализ информационных зависимостей в программе. Для описания зависимостей используется граф информационных связей. Он реализован другим разработчиком в виде отдельного класса C++.

На базе ОРС разработана демонстрационная программа, которая позволяет наглядно демонстрировать возможности системы.

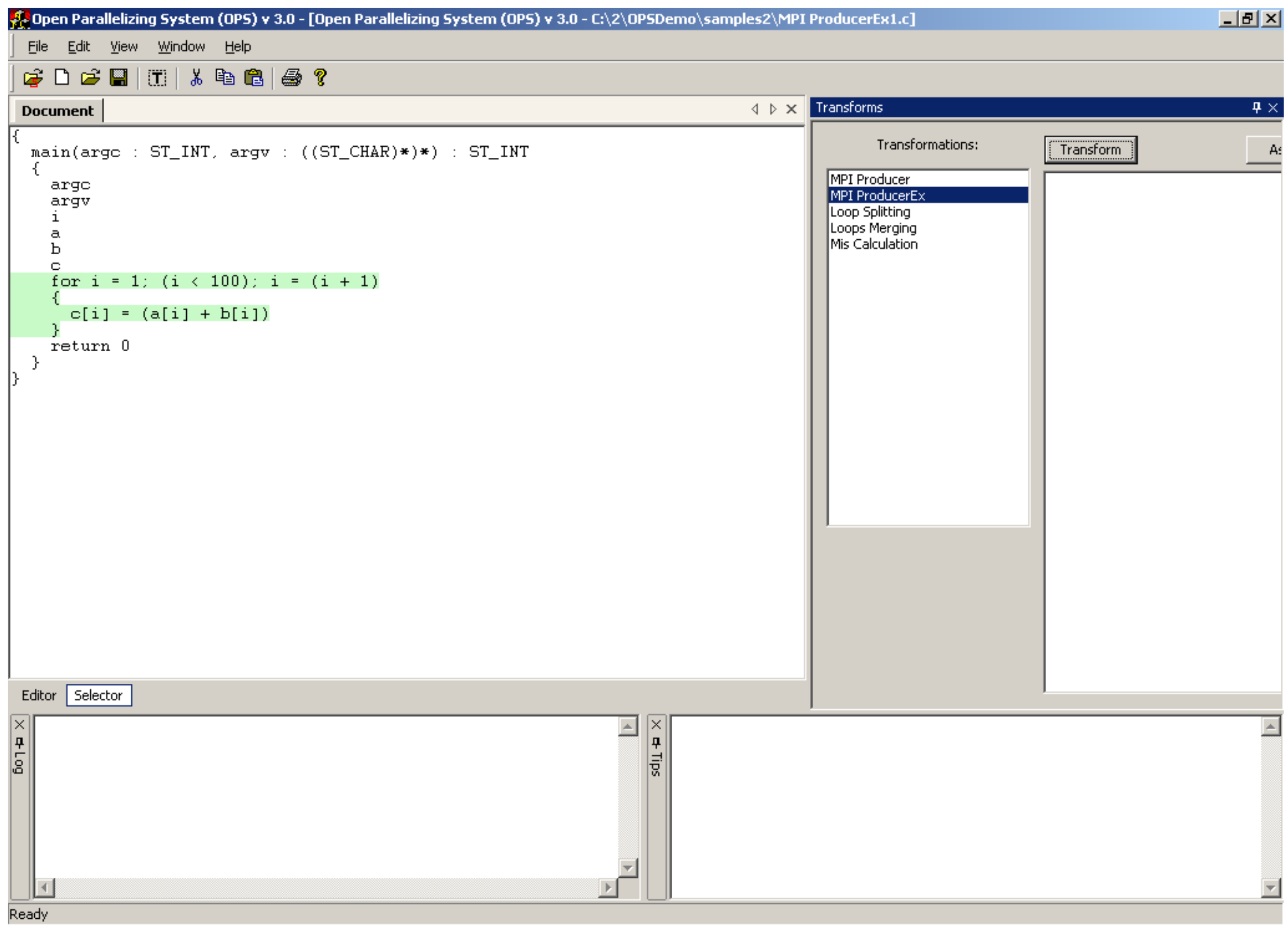


Рис. 1.

На рис.1. показан уже разобранный парсером пример, в котором выбран цикл для осуществления преобразования.

Open Parallelizing System (OPS) v 3.0 - [Open Parallelizing System (OPS) v 3.0 - C:\2\OPSDemo\samples2\MPI ProducerEx1.c]

mpi_out.c - Блокнот

Файл Правка Формат Вид Справка

```
#include <iostream>
#include "mpi.h"

int MPIPRODEX_myid = 0, MPIPRODEX_numprocs = 0;


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


int i;
int a[100];
int b[100];
int c[100];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &MPIPRODEX_numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &MPIPRODEX_myid);

if (MPIPRODEX_myid == 0)
{
MPI_Bcast(&argc, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&i, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 1; i < MPIPRODEX_numprocs; ++i)
{
int MPIPRODEX_left = i*(100-1)/(1*MPIPRODEX_numprocs);
int MPIPRODEX_count = (i+1)*(100-1)/(1*MPIPRODEX_numprocs) - MPIPRODEX_left;
MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i, MPI_COMM_WORLD);
}
for (i = 1; i < MPIPRODEX_numprocs; ++i)
{
int MPIPRODEX_left = i*(100-1)/(1*MPIPRODEX_numprocs);
int MPIPRODEX_count = (i+1)*(100-1)/(1*MPIPRODEX_numprocs) - MPIPRODEX_left;
MPI_Send(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i, MPI_COMM_WORLD);
}
for (i = 1; i < MPIPRODEX_numprocs; ++i)
{
int MPIPRODEX_left = i*(100-1)/(1*MPIPRODEX_numprocs);
int MPIPRODEX_count = (i+1)*(100-1)/(1*MPIPRODEX_numprocs) - MPIPRODEX_left;
MPI_Send(c + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i, MPI_COMM_WORLD);
}
}
else
{
int MPIPRODEX_left = MPIPRODEX_myid*(100-1)/(1*MPIPRODEX_numprocs);
int MPIPRODEX_count = (MPIPRODEX_myid+1)*(100-1)/(1*MPIPRODEX_numprocs) - MPIPRODEX_left;
MPI_Recv(&argc, 1, MPI_INT, 0, MPI_COMM_WORLD, 0);
```

Ready

Рис. 2.

На рис. 2. показан результат выполнения генератора MPI-кода на выбранном цикле.

3. Общая информация о библиотеке MPI.

MPI программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу (не обязательно одну и ту же), написанную на языке C или FORTRAN. Появились реализации MPI для C++, однако разработчики стандарта MPI за них ответственности не несут. Процессы MPI программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Как правило, каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти. MPI не специфицирует модель выполнения процесса - это может быть как последовательный процесс, так и многопотоковый. MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются либо на операционную систему, либо на программиста. В частности, на nCUBE2 используется стандартная команда *xnc*, а на кластерах специальный командный файл (скрипт) *mpirun*, который предполагает, что исполнимые модули уже каким-то образом распределены по компьютерам кластера. MPI не накладывает каких-либо ограничений на то, как процессы будут распределены по процессорам, в частности, возможен запуск MPI программы с несколькими процессами на обычной однопроцессорной системе.

Для идентификации наборов процессов вводится понятие *группы*, объединяющей все или какую-то часть процессов. Каждая группа образует *область связи*, с которой связывается специальный объект - *коммуникатор* области связи. Процессы внутри группы нумеруются целым числом в диапазоне `0..groupsize-1`. Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором. При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор

MPI_COMM_WORLD. В большинстве случаев, на каждом процессоре запускается один отдельный процесс, и тогда термины процесс и процессор становятся синонимами, а величина groupsize становится равной NPROCS – числу процессоров выделенных задаче. В дальнейшем обсуждении мы будем понимать именно такую ситуацию и не будем очень уж строго следить за терминологией.

Итак, если сформулировать коротко, MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Это достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

- функции инициализации и закрытия MPI процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Набор функций библиотеки MPI далеко выходит за рамки набора функций минимально необходимого для поддержки механизма передачи сообщений [1]. Однако сложность этой библиотеки не должна пугать пользователей, поскольку, в конечном итоге, все это множество функций предназначено для облегчения разработки эффективных параллельных программ. В конце концов, пользователю принадлежит право самому решать, какие из предоставляемого арсенала средств использовать, а какие нет. В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций [2].

Каждая из MPI функций характеризуется способом выполнения:

1) Локальная функция – выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.

2) Нелокальная функция - для ее завершения требуется выполнения MPI процедуры другим процессом.
3) Глобальная функция – процедуру должны выполнять все процессы группы. Не соблюдение этого условия может приводить к зависанию задачи.
4) Блокирующая функция – возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.
5) Неблокирующая функция – возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла *mpi.h*. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках. В таблице 1 приведено соответствие предопределенных в MPI типов стандартным типам языка С.

Таблица 1. Соответствие между MPI-типами и типами языка С.

тип MPI	**тип языка С**
MPI_CHAR	signed char
MPI_SHORT	signed short int

MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

В таблице 1 перечислен обязательный минимум поддерживаемых стандартных типов, однако если в базовой системе представлены и другие типы, то их поддержку будет осуществлять и MPI, например, если в системе есть поддержка комплексных переменных двойной точности DOUBLE COMPLEX, то будет присутствовать тип MPI_DOUBLE_COMPLEX. Типы MPI_BYTE и MPI_PACKED используется для передачи двоичной информации без какого-либо преобразования. Кроме того, программисту предоставляются средства создания собственных типов на базе стандартных.

4. Базовые функции библиотеки MPI.

Любая прикладная MPI-программа (приложение) должна начинаться с вызова функции инициализации MPI (функция MPI_Init). В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. Кроме этого создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса.

```
int MPI_Init(int *argc, char ***argv)
```

В программах на С каждому процессу при инициализации передаются аргументы функции main, полученные из командной строки.

Функция завершения MPI программ MPI_Finalize.

```
int MPI_Finalize(void)
```

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Функция определения числа процессов в области связи MPI_Comm_size.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

IN comm - коммуникатор

OUT size - число процессов в области связи коммуникатора comm.

Функция возвращает количество процессов в области связи коммуникатора comm.

До создания явным образом групп и связанных с ними коммуникаторов единственно возможными значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI. Подпрограмма является локальной.

Функция определения номера процесса MPI_Comm_rank.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

IN comm - коммуникатор

OUT rank - номер процесса, вызвавщего функцию.

Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции). Подпрограмма является локальной.

В минимальный набор следует включить также две функции передачи и приема сообщений.

Функция передачи сообщения MPI_Send.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

IN buf - адрес начала расположения пересылаемых данных

IN count - число пересылаемых элементов

IN datatype - тип посылаемых элементов

IN dest - номер процесса-получателя в группе, связанной с коммуникатором comm

IN tag - идентификатор сообщения (аналог типа сообщения функций
nread и nwrite PSE nCUBE2)

IN comm - коммуникатор области связи

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf – это, как правило, массив, или скалярная переменная. В последнем случае значение count = 1.

Функция приема сообщения MPI_Recv.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

OUT buf - адрес начала расположения принимаемого сообщения

IN count - максимальное число принимаемых элементов

IN datatype - тип элементов принимаемого сообщения

IN source - номер процесса-отправителя

IN tag - идентификатор сообщения

IN comm - коммуникатор области связи

OUT status - атрибуты принятого сообщения

Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

5. Информационные зависимости в программе.

В связи с тем, что в работе используется анализ информационных зависимостей, необходимо напомнить понятия информационной зависимости теории оптимизирующих/распараллеливающих преобразований программ [3].

Вхождением переменной, как обычно, будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась. Иногда, не умаляя общности, все переменные, кроме счетчиков циклов, будем считать индексными (т.е. массивами), поскольку всякую безындексную переменную `z` можно заменить переменной `z(1)`. Всякому вхождению при конкретном значении индексного выражения соответствует обращение к некоторой ячейке памяти. Если при этом обращении происходит запись в ячейку памяти (вхождение в левую часть оператора присваивания, не входящее в индексное выражение другого вхождения), то такое вхождение называется генератором. Остальные вхождения называются использованиями.

Пример 2. В следующем операторе присваивания

```
A(I+B(J+2)) = D(I-1, B(I)) + 3*A(1) – I + 5
 1 2 3 4       5 6    7 8       9     10
```

десять вхождений переменных (их номера проставлены снизу) – и только первое является генератором.

Говорят, что два вхождения порождают информационную зависимость, если при некоторых допустимых значениях индексных выражений они обращаются к одной и той же ячейке памяти.

Пример 3.

```
    DO 99 I = 1, 100
99  A(I) = D(I, I-1) + A(I-1)
```

В операторе с меткой 99 вхождения переменной A информационно зависимы, поскольку оба обращаются к ячейке памяти A(10): вхождение A(I) на десятой итерации, а A(I-1) - на одиннадцатой итерации цикла.

Граф информационных связей – это ориентированный граф, вершины которого соответствуют вхождениям, а дуга соединяет пару вершин (v,u), если выполняется одно из следующих условий:

1. Эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение v раньше, чем u и хотя бы одно из этих вхождений является генератором.

2. Вхождение u является генератором, а вхождение v принадлежит этому же оператору присваивания. Такие дуги будем называть тривиальными.

Генератор будем обозначать – out (output), а использование - in (input). Дуги графа информационных связей бывают трех типов в зависимости от типов инцидентных им вершин: out-in - истинная информационная зависимость (true dependence), in-out - антизависимость (antidependence), out-out - выходная зависимость (output dependence) [3, с.95].

Если информационная зависимость связывает два использования, то мы будем говорить об in-in зависимости. Иногда будем рассматривать обобщенный граф информационных связей. Этот граф является смешанным (содержит и дуги и ребра) и представляет собой обычный граф информационных связей, к которому добавлены ребра, соединяющие вершины, соответствующие вхождениям, связанным in-in зависимостью.

Иногда бывает трудно определить, существует или нет информационная зависимость между парой вхождений. При преобразованиях программ в таких случаях предполагают худшее – считают, что такая зависимость существует, и на графе соответствующие вершины соединяют дугой.

Пример 4.

```
    DO 99 I = 1, N
99  X(2*I) = X(2*I+k)
```

В этом примере, если k нечетное, то между обоими вхождениями переменной X нет информационной зависимости; если k четное и отрицательное, то есть истинная информационная зависимость, делающая цикл рекуррентным; если k четное и неотрицательное, то имеет место антизависимость. Если до выполнения программы неизвестно, какое значение примет k к началу выполнения цикла, то и характер зависимости тоже нельзя определить. В этом случае в графе информационных зависимостей должны быть проведены обе дуги между двумя вхождениями переменной X.

Пример 5.

```
     DO 333 J = 2, N
     DO 333 I = 2, N
     A(I-2, J) = X(2*I-1)
c       v1          v2
     X(2*I) = X(2*I+3) + A(I,J-1)
c       v3       v4          v5
     X(2*I+1) = A(J, I-1)
c       v6          v7
333  X(2*I+2) = A(I+K, J) + A(I-2, J)
        v8          v9          v10
```

Выпишем список всех нетривиальных дуг графа информационных связей этого цикла:

```
(v1; v7), (v1; v9), (v1; v10), (v4; v6), (v5; v1),
(v6; v2), (v7; v1), (v8; v3), (v9; v1)
```

Если до выполнения программы неизвестно, какое значение примет K к началу выполнения цикла, то и характер зависимости между вхождениями v1 и v9 нельзя определить. В этом случае в графе информационной

зависимости должны быть проведены обе дуги между этими двумя вхождениями.

Пример 6.

```
     DO 333 I = 1, N
333   A(I+2) = A(N-I)
c       v1        v2
```

Для данного цикла список дуг графа имеет вид:

```
v1 – v2
v2 – v1
```

Пример 7. В следующем цикле дуга графа информационных связей ведет от первого вхождения `A(I)` ко второму, но не наоборот. А вхождения переменной `X` соединяются двумя дугами в обе стороны: дуга антизависимости, поскольку на каждой итерации сначала `X` используется в первом операторе тела цикла, а потом перезаписывается; дуга истинной зависимости, поскольку на каждой итерации, начиная со второй, используется значение `X`, полученное на предыдущей итерации.

```
      DO 99 I = 1, N
      A(I) = B + X
      X = A(I)
 99   CONTINUE
```

Информационная зависимость между вхождениями называется циклически независимой (loop independent dependence), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется циклически порожденной (loop carried dependence) [3, с.96].

В предыдущем примере дуга зависимости между вхождениями `A(I)` циклически независима, дуга зависимости между вхождениями `X`, ведущая сверху вниз тоже циклически независима, а дуга, ведущая снизу вверх – циклически зависима.

Пусть `v1` и `v2` – два вхождения массива `X` в тело цикла со счетчиком `I`, связанные информационной зависимостью. Информационную зависимость между `v1` и `v2` будем называть *регулярной* относительно этого цикла, если разность индексных выражений вхождений `v1` и `v2` не зависит от счетчика цикла `I`.

6. Дополнительные оптимизирующие преобразования.

В работе реализованы также такие оптимизирующие преобразования как: развёртка циклов и гнездование циклов.

6.1. Развёртка программных циклов.

Развёртка цикла кратности `k` – это преобразование программного цикла, при котором вместо цикла с шагом `1` записывается цикл с шагом `k` и телом, состоящим из `k` раз повторённого тела исходного цикла с учётом значения счётчика цикла и синтаксической корректности. Это преобразование эквивалентно всегда. Целесообразность применения этого преобразования следует из того, что оно уменьшает количество итераций цикла, за счёт увеличения его тела. А уменьшение количества итераций означает уменьшение числа пересылок между процессами. Преобразование реализовано в виде отдельного модуля, что позволяет его использовать не только в рамках этой работы. Как и все преобразования, оно работает с внутренним представлением. То есть на вход подаётся указатель на цикл во внутреннем представлении и кратность развёртки, а на выходе то же внутреннее представление, но уже с преобразованным циклом.

Пример 8: Исходная программа:

```
void main()
{
	int i, a[10];

	for(i = 0; i < 10; i = i + 1)
		a[i] = 2 * i;
}
```

После применения преобразования с кратностью развёртки равной 5 получаем:

```
void main()
{
     int i, a[23];

     for(i = 0; i < 19; i = i + 5)
     {
          a[i] = 2 * i;
          a[i+1] = 2 * (i+1);
          a[i+2] = 2 * (i+2);
          a[i+3] = 2 * (i+3);
          a[i+4] = 2 * (i+4);
     }
     a[20] = 2 * 20;
     a[21] = 2 * 21;
     a[22] = 2 * 22;
}
```

На этом примере видно, что в случае, когда число итераций исходного цикла не делится на кратность развёртки, после преобразованного цикла необходимо дописать тело исходного цикла `n` *mod* `k` раз. Где `n` - число итераций исходного цикла, `k` – кратность развёртки, а *mod* – операция взятия остатка от целочисленного деления.

Рассмотрим теперь пример, в котором на этапе выполнения преобразования число итераций исходного цикла неизвестно.

Пример 9: Исходная программа:

```
void main()
{
     int i, a[10], n = 10;

     for(i = 0; i < n; i = i + 1)
          a[i] = i * 3;
}
```

После применения преобразования с кратностью развёртки равной 3:

```
void main()
{
     int i, a[10], n = 10;
     int _RightBound_1;
```

```
        _RightBound_1 = (n / 3) * 3;
        for(i = 0; i < _RightBound_1; i = i + 3)
        {
              a[i] = i * 3;
              a[i+1] = (i+1) * 3;
              a[i+2] = (i+2) * 3;
        }
        for(i = _RightBound_1; i < n; i = i + 1)
              a[i] = i * 3;
}
```

На этом примере видно, что в случае, когда число итераций исходного цикла зависит от переменных и не может быть посчитано на момент выполнения преобразования, после преобразованного цикла дописывается ещё один цикл, в котором итераций меньше чем кратность развёртки. Вводится также новая переменная с уникальным именем, что позволяет сократить количество вычислений правой границы преобразованного цикла.

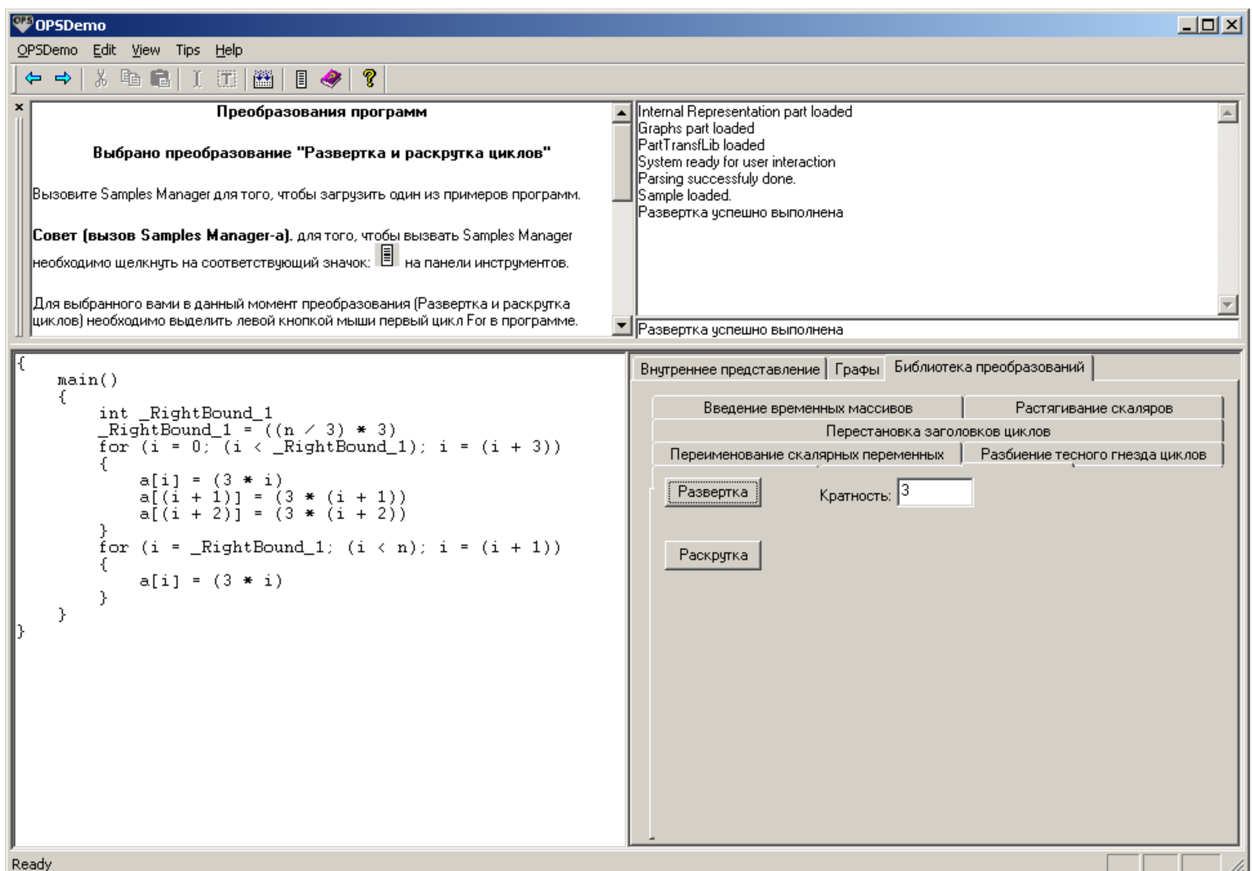


Рис. 3.

6.2. Гнездование программных циклов.

Гнездование циклов это преобразование, которое заменяет цикл

```
for(i = 0; i < N; i = i + 1)
     BODY(i);
```

на следующий фрагмент программы

```
for(j = 0; j < h; j = j + 1)
     for(i = 0; i < (N / h); i = i + 1)
          BODY(i + j * (N / h));
for(i = 0; i < (N - (N / h) * h); i = i + 1)
     BODY((N / h) * h + i);
```

Так как у внешнего цикла, получившегося в результате преобразования гнезда циклов, количество итераций определяется самим преобразованием, а размер внутреннего цикла зависит ещё и от исходной программы, то гнездование целесообразно применять в случае, когда заранее известно число параллельных процессов. Это преобразование не меняет порядок выполнения операций, а значит равносильно.

Пример 10: Исходная программа:

```
void main()
{
     int i, a[100], n = 10;

     for(i = 0; i < n * n; i = i + 1)
          a[i] = 5 * i;
}
```

После применения преобразования при кратности гнездования равной 3.

```
void main()
{
     int i, a[100], n = 10;
     int j1;
```

```
      for(j1 = 0; j1 < 3; j1 = j1 + 1)
            for(i = 0; i < (n * n) / 3; i = i + 1)
                  a[i + j1 * (N / 3)] = 5 * (i + j1 * (N / 3));

      for(i = 0; i < (N - (N / 3) * 3); i = i + 1)
            a[((n * n) / 3) * 3 + i] = 5 * (((n * n) / 3) * 3 + i);
}
```

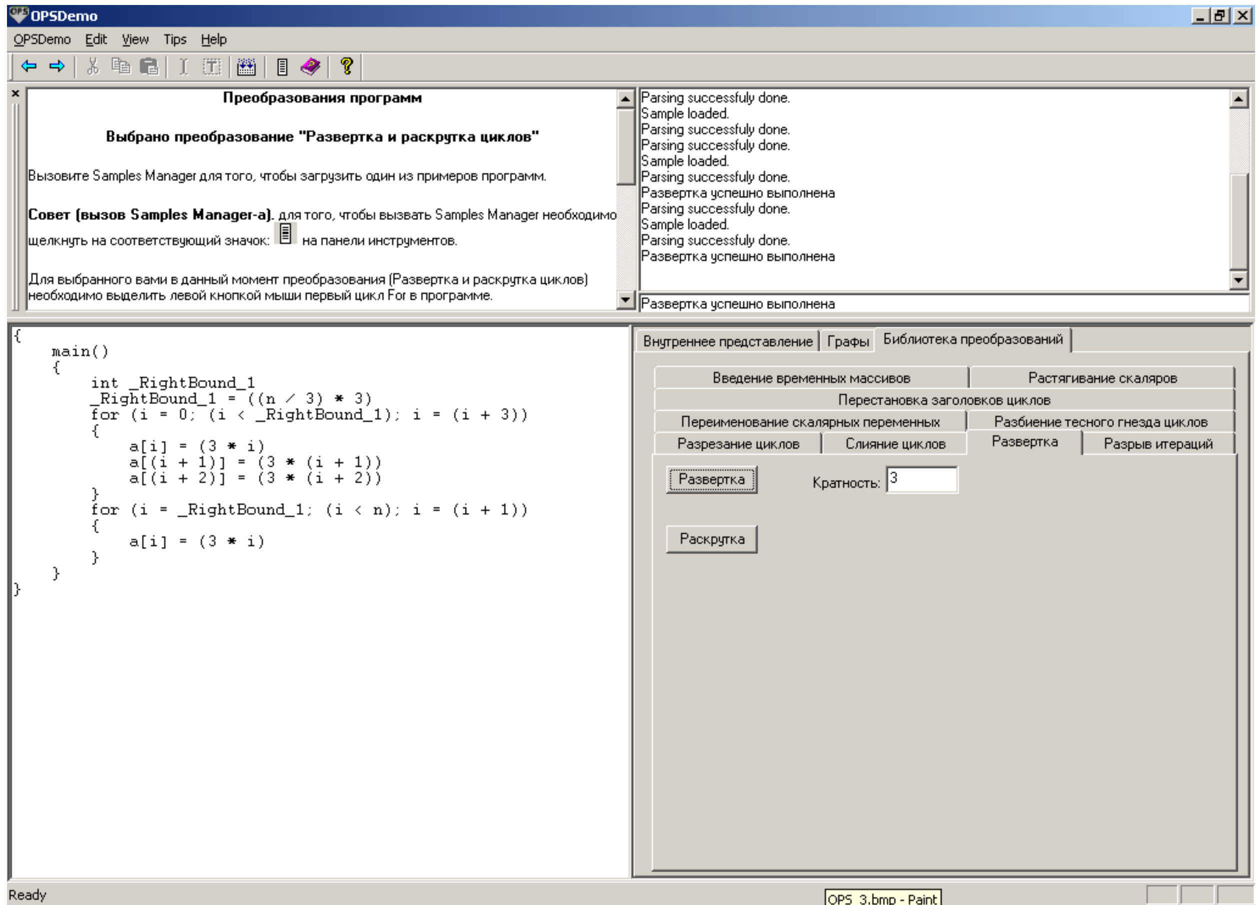


Рис. 4.

7. Автоматическая генерация MPI кода для циклов, не содержащих информационные зависимости.

При распараллеливании цикла необходимо распределить его итерации и входные (по отношению к циклу) данные между параллельными процессами. Предполагается что все данные, используемые циклом, к моменту его выполнения находятся в главном процессе. Также, после работы цикла, данные, которые подвергались изменению в результате его работы, необходимо из всех параллельных процессов переслать обратно в главный процесс. Рассмотрим сначала случай, когда распараллеливаемый цикл не содержит информационных зависимостей. Таким циклом, например, является цикл, вычисляющий поэлементное сложение векторов.

Пример 11:

```
for (i = 0; i < n; i++)
     c[i] = a[i] + b[i];
```

Итерации в генераторе распределяются таким образом, чтобы всем процессам досталось равное количество итераций. Если количество итераций не делится нацело на количество процессов, то некоторым процессам достаётся на одну итерацию больше. Например, в случае когда количество итераций исходного цикла равно 18, а количество процессов равно 4, распределение будет таким: нулевой процесс будет выполнять итерации с 0 по 3, первый процесс с 4 по 7, второй с 8 по 12 и третий с 13 по 17 (номера итераций и номера процессов начинаются с 0). Более наглядно это можно представить на схеме.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><td colspan="4">Процесс 0</td><td colspan="4">Процесс 1</td><td colspan="5">Процесс 2</td><td colspan="5">Процесс 3</td></tr>
</table>

Здесь верхняя строчка представляет номера итераций, а нижняя строчка процессы. Количество итераций исходного цикла и количество процессов не известны генератору на этапе его работы, поэтому все необходимые вычисления вставляются в виде кода в сгенерированную программу.

Распределение данных осуществляется следующим образом. Генератор определяет по индексным выражениям используемых в цикле массивов, какие из их элементов потребуются в том или ином процессе, и в сгенерированную программу помещается код, непосредственно перед циклом, который пересылает эти элементы между процессами используя функции библиотеки MPI.

Из тела цикла в примере 11 видно, что используются массивы `a` и `b`. Причём на `i`-ой итерации используются элементы с индексом `i`. Что позволяет распределить данные между процессами аналогично итерациям. В генераторе единственного оператора стоит массив `c`, который тоже используется с индексным выражением равным счётчику цикла. То есть массив `c` будет распределён точно также как и остальные данные.

Вот фрагмент сгенерированной программы для этого примера.

```
if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs - MPIPRODEX_left;
    MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD);
  }
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs - MPIPRODEX_left;
    MPI_Send(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD);
  }
}
```

```
else
{
  int MPIPRODEX_left = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
  int MPIPRODEX_count = (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs –
      MPIPRODEX_left;
  MPI_Recv(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0,
      MPI_COMM_WORLD, 0);
  MPI_Recv(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0,
      MPI_COMM_WORLD, 0);
}

for (i = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
  i < (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs; i++)
{
  c[i] = a[i] + b[i];
}

if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; ++i)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs - MPIPRODEX_left;
    MPI_Recv(c + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD, 0);
  }
}
else
{
  int MPIPRODEX_left = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
  int MPIPRODEX_count = (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs –
      MPIPRODEX_left;
  MPI_Send(c + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0, MPI_COMM_WORLD);
}
```

В переменной `MPIPRODEX_myid` хранится идентификатор текущего процесса, а в переменной `MPIPRODEX_numprocs` количество процессов. Они инициализируются библиотекой `MPI` в начале программы. Программный код инициализации библиотеки `MPI` вставляется в сгенерированную программу в функцию `main`, если таковая отсутствует, то генератор сигнализирует об этом пользователю.

8. Автоматическая генерация MPI кода для циклов, содержащих зависимости.

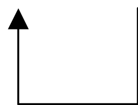
В программах решающих реальные задачи редко встречаются циклы, которые в своём теле не содержат информационных зависимостей. Поэтому генератор должен уметь обнаруживать их и соответствующим образом реагировать. В этой главе рассмотрены все виды информационных зависимостей и то как генератор их обрабатывает.

8.1. Циклы, содержащие антизависимости.

Рассмотрим случай для цикла, содержащего в теле антизависимость. Антизависимость – это `in-out` дуга графа информационных зависимостей (см. 5. Информационные зависимости в программе.4), то есть из одной ячейки памяти сначала производится чтение, а потом в эту же ячейку записывается информация.

Пример 12:

```
for (i = 0; i < n; i++)
    a[i] = a[i+5] * 2;
```



В теле данного цикла присутствует одна антизависимость, обозначенная стрелкой. Видно, что из ячейки памяти соответствующей элементу массива `a[6]` читается информация на итерации 1 и в эту же ячейку памяти записывается информация на итерации 6. То есть в таком цикле используются только те данные, которые ещё не были изменены во время его работы. Значит, достаточно распределить данные также как и в предыдущем

случае, когда зависимостей не было. Отличие заключается в том, что массив `a` будет не только распределяться между процессами до работы цикла, но и будет собираться в главном процессе после окончания работы цикла, так как он изменялся. Причём в использовании и в генераторе стоят разные индексные выражения, а значит, использование и генератор будут распределены по-разному. Например, если некоторый процесс обрабатывает итерации цикла от `i1` по `i2`, то в него будут переданы элементы массива от `a[i1+5]` по `a[i2+5]`. А после окончания работы цикла в главный процесс из этого будут переданы элементы от `a[i1]` по `a[i2]`. Ниже представлен результат работы генератора MPI кода для этого примера.

```
if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs + 5;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs + 5 -
        MPIPRODEX_left;
    MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD);
    }
}
else
{
  int MPIPRODEX_left = MPIPRODEX_myid * n / MPIPRODEX_numprocs + 5;
  int MPIPRODEX_count = (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs + 5 -
      MPIPRODEX_left;
  MPI_Recv(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0,
      MPI_COMM_WORLD, 0);
}

for (i = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
     i < (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs; i++)
{
    a[i] = a[i + 5] * 2;
}

if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs - MPIPRODEX_left;
    MPI_Recv(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD, 0);
    }
}
else
{
  int MPIPRODEX_left = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
  int MPIPRODEX_count = (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs -
```

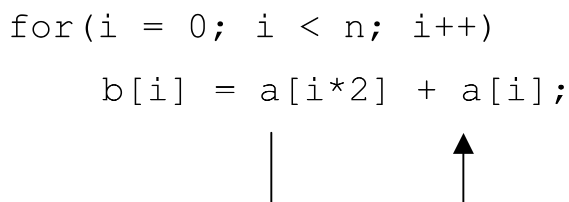
```
      MPIPRODEX_left;
  MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0, MPI_COMM_WORLD);
}
```

8.2. Циклы, содержащие входные зависимости.

Рассмотрим случай для цикла, содержащего в теле входную зависимость. Входная зависимость – это `in-in` дуга графа информационных зависимостей (см. 5. Информационные зависимости в программе.), то есть из одной ячейки памяти производится чтение несколько раз, возможно на разных итерациях цикла.

Пример 13:

```
for(i = 0; i < n; i++)
     b[i] = a[i*2] + a[i];
```



В теле данного цикла присутствует одна входная зависимость, обозначенная стрелкой. Видно, что из ячейки памяти соответствующей элементу массива `a[8]` читается информация на итерациях 4 и 8. То есть в таком цикле используются только те данные, которые ещё не были изменены во время его работы. Значит, достаточно распределить данные также как и в предыдущем случае, когда зависимостей не было. Отличие заключается в том, что массив `a` будет распределяться между процессами с учётом обоих своих вхождений. Ниже представлен результат работы генератора MPI кода для этого примера.

```
if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs * 2;
    MPIPRODEX_left = min(i * n / MPIPRODEX_numprocs, MPIPRODEX_left);

    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs * 2;
    MPIPRODEX_count = max((i + 1) * n / MPIPRODEX_numprocs, MPIPRODEX_count);

    MPIPRODEX_count = MPIPRODEX_count - MPIPRODEX_left;
```

```
    MPI_Send(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD);
    }
}
else
{
  int MPIPRODEX_left = i * n / MPIPRODEX_numprocs * 2;
  MPIPRODEX_left = min(i * n / MPIPRODEX_numprocs, MPIPRODEX_left);

  int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs * 2;
  MPIPRODEX_count = max((i + 1) * n / MPIPRODEX_numprocs, MPIPRODEX_count);

  MPIPRODEX_count = MPIPRODEX_count - MPIPRODEX_left;

  MPI_Recv(a + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0,
      MPI_COMM_WORLD, 0);
}

for (i = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
     i < (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs; i++)
{
    b[i] = a[i * 2] + a[i];
}

if (MPIPRODEX_myid == 0)
{
  for (i = 1; i < MPIPRODEX_numprocs; i++)
  {
    int MPIPRODEX_left = i * n / MPIPRODEX_numprocs;
    int MPIPRODEX_count = (i + 1) * n / MPIPRODEX_numprocs - MPIPRODEX_left;
    MPI_Recv(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, i,
        MPI_COMM_WORLD, 0);
    }
}
else
{
  int MPIPRODEX_left = MPIPRODEX_myid * n / MPIPRODEX_numprocs;
  int MPIPRODEX_count = (MPIPRODEX_myid + 1) * n / MPIPRODEX_numprocs -
      MPIPRODEX_left;
  MPI_Send(b + MPIPRODEX_left, MPIPRODEX_count, MPI_INT, 0, MPI_COMM_WORLD);
}
```

8.3. Циклы, содержащие истинные зависимости.

Зависимости такого типа появляются в рекуррентных циклах.

Пример 14:

```
for(i = 2; i < N; i++)
     a[i] = a[i-1] + a[i-2];
```

В некоторых, частных случаях, возможно приведение рекуррентного вычисления к нерекуррентному. Также в труде [8] авторы рассматривают различные методы неавтоматического распараллеливания таких циклов.

В данной работе истинные информационные зависимости не рассматривались.

8.4. Циклы, содержащие выходные зависимости.

Выходные информационные зависимости представляют собой проблему для распараллеливания. Однако, применяя некоторые вспомогательные преобразования программ, можно попытаться избавиться от зависимостей такого вида. Например, преобразования «переименование переменных» и «разрезание циклов». Преобразование «переименование переменных» помогает избавиться от ложных дуг выходной зависимости. А преобразование «разрезание цикла» позволяет разрезать цикл на несколько таким образом, чтобы часть из получившихся циклов не содержала выходные зависимости. На данный момент такая возможность не реализована в данном генераторе MPI-кода.

Пример 15:

```
for (i = 0; i < N; i++)
{
    a[i] = i;
    a[2*i] = i + 2;
}
```

9. Заключение.

В данной работе разработан генератор параллельного MPI-кода для Открытой распараллеливающей системы.

В дальнейшем планируется провести ряд улучшений и дополнений, связанных с введением оптимизации, использованием вспомогательных преобразований, а также обработкой истинных информационных зависимостей.

10. Литература.

1. Дацюк В.Н., Букатов А.А., Жегуло А.И. Методическое пособие по курсу «Многопроцессорные системы и параллельное программирование» Часть I. Введение в организацию и методы программирования многопроцессорных вычислительных систем. Ростов-на-Дону, 2000.
2. Ian Foster. Designing and Building Parallel Programs. http://www.hensa.ac.uk/parallel/books/addison-wesley/dbpp/index.html
3. Векторизация программ. // Векторизация программ: теория, методы, реализация. / Сборник переводов статей М.: Мир, 1991. С. 246 - 267.
4. Штейнберг Б.Я., Черданцев Д.Н., Науменко С.А., Бутов А.Э., Петренко В.В. Преобразования программ для открытой распараллеливающей системы // Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 3, с. 97-104.
5. Штейнберг Б. Я., Бутов А. Э., Науменко С. А., Петренко В. В., Черданцев Д. Н., Штейнберг Р. Б., Шульженко А. М.. Полуавтоматическое распараллеливание на основе Открытой распараллеливающей системы. Тезисы Всероссийской научно-технической конференции «Параллельные вычисления в задачах математической физики», 21 – 25 июня 2004 г. «ЮГИНФО РГУ», стр. 178 – 185.
6. Штейнберг Б.Я.. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью. Ростов-на-Дону, 2004 г.

7. Штейнберг Б.Я., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М., Бутов А.Э., Науменко С.А., Петренко В.В., Шилов М.В., Гуфан К.Ю., Тузаев А.В., Арутюнян О.Э., Морылёв Р.И. Обучающая распараллеливанию программа на основе ОРС. Тезисы научно-методической конференции «Современные информационные технологии в образовании: Южный Федеральный округ», 13 - 14 мая 2004 г., Ростовский государственный университет, издательство «ЦВВР», стр. 248 - 250.
8. Аветисян А.И., Гайсарян С.С., Самоваров О.И. Возможности оптимального выполнения параллельных программ, содержащих простые и итерированные циклы, на неоднородных параллельных вычислительных системах с распределенной памятью // Программирование, 2002, № 1, с. 38-54.
9. Антонов А.С. Параллельное программирование с использованием технологии MPI.// М.: Изд-во МГУ, 2004.
10. Воеводин В.В. Воеводин Вл.В. Параллельные вычисления, С-Петербург «БХВ-Петербург», 2002.
11. GNU Project – Free Software Foundation http://gcc.gnu.org