УДК 681.3

СРЕДСТВА СИНТЕЗА ПАРАЛЛЕЛЬНЫХ MPI-ПРОГРАММ

А.Е. Дорошенко, Е.А. Яценко

Институт программных систем НАН Украины,
03187, Киев, проспект Академика Глушкова, 40.
Тел.: 526 1538, e-mail: dor@isofts.kiev.ua, aiyat@i.com.ua

К.А. Жереб
Киевское отделение Московского физико-технического института,
03650, Киев, проспект Глушкова, 40/1.
E-mail: cotan@mail.ru

Предложен метод проектирования и синтеза MPI-программ, который основывается на алгеброалгоритмическом подходе. Разработана методика применения системы переписывающих правил для автоматизированной трансформации последовательных схем алгоритмов в параллельные, ориентированные на передачу сообщений. Проведен эксперимент по выполнению синтезированной параллельной программы на кластерной мультипроцессорной архитектуре.

The method for designing and synthesis of MPI programs based on algebraic-algorithmic approach is proposed. The technique of rewriting terms system application for automated transformation of serial schemes of algorithms to parallel ones, which are oriented on message passing, is developed. The experiment of execution of synthesized parallel program on cluster multiprocessor architecture is carried out.

Введение

В работах [1–3] рассмотрен разработанный авторами интегрированный инструментарий проектирования и синтеза программ («ИПС»), основывающийся на системах алгоритмических алгебр (САА) В.М. Глушкова и методе диалогового конструирования синтаксически правильных программ [4, 5]. Его особенность состоит в совместном использовании трех представлений алгоритма при его конструировании: аналитического (формула в алгебре алгоритмов), естественно-лингвистического (текстового) и графового (граф-схемы). Данный инструментарий применен для синтеза многопоточных программ по схемам асинхронных алгоритмов для выполнения на многоядерной архитектуре. Рассматривалось также совместное использование «ИПС» и системы переписывания правил TermWare для автоматизации трансформации многопоточных программ с общей памятью. В данной работе предложено развитие инструментария «ИПС» для проектирования и синтеза MPI-программ для выполнения на кластерной мультипроцессорной архитектуре.

Материал работы состоит из разделов. В разделе 1 описан разработанный интегрированный инструментарий и предложен метод проектирования и синтеза программ на языке C, использующих MPI. В разделе 2 предложена методика применения системы TermWare совместно с «ИПС» для трансформации последовательных схем алгоритмов в параллельные схемы, ориентированные на передачу сообщений между процессами. Применение разработанного подхода показано на примере разработки параллельной программы нахождения простых чисел. Приведены результаты эксперимента по выполнению данной программы на кластере Института кибернетики им. В.М. Глушкова НАН Украины [6].

1. Интегрированный инструментарий проектирования и синтеза программ и его настройка на разработку MPI-программ

Основными компонентами разработанного интегрированного инструментария [1, 5] являются диалоговый конструктор синтаксически правильных программ (ДСП-конструктор), редактор граф-схем, трансформатор схем, генератор САА-схем и среда конструирования алгеброалгоритмических описаний (СКАО).

В ДСП-конструкторе осуществляется проектирование алгоритмов в естественно-лингвистической (САА-схема) и алгебраической (регулярная схема) формах. При этом конструирование алгоритма выполняется по уровням сверху-вниз путем суперпозиции операторных и логических операций САА и представляется в виде дерева. В процессе конструирования автоматически генерируется текст схемы алгоритма в алгебраической и текстовой формах. Просмотр и изменение графового представления алгоритма могут быть осуществлены в редакторе граф-схем. По полученному в процессе конструирования алгоритма дереву, реализациям операций САА, элементарных операторов и условий на целевом языке программирования (Java, C, C++) ДСП-конструктор выполняет синтез программы.

СКАО необходима для хранения, ввода и изменения элементов схем, алгоритмов и правил их конструирования. Данная среда содержит: стратегии обработки, описывающие классы алгоритмов; метаправила проектирования схем (свертку, развертку, трансформацию, переориентацию) [4, 5], необходимые

для конструирования новых алгоритмических знаний; операции САА и базисные элементы схем (предикаты и операторы), представленные в трех формах (естественно-лингвистической, алгебраической и граф-схемной), а также их программные реализации, используемые в процессе генерации программ; последовательные и параллельные схемы алгоритмов символьной обработки (поиск в файлах, сортировка массивов). Базисные элементы схем необходимы для доступа к данным, обрабатываемыми алгоритмами. Текст базисных предикатов в САА-схемах указывается в одинарных кавычках, базисных операторов – в двойных.

К операциям САА, хранящимся в СКАО, относятся логические операции (дизъюнкция, конъюнкция, отрицание), последовательные операторные операции (композиция, альтернатива, цикл), а также специализированные операторные конструкции, ориентированные на параллельные вычисления (асинхронная дизъюнкция, синхронизатор, контрольная точка) [1, 2, 5]. В работе [1] рассмотрены программные реализации параллельных операций САА, ориентированные на генерацию многопоточных программ на языке Java. В работах [2, 3] рассматриваются алгоритмы, по которым были сгенерированы многопоточные программы на языке C++.

С целью ориентации ДСП-конструктора на проектирование и синтез программ для кластерной архитектуры, в СКАО были включены дополнительные конструкции, ориентированные на реализацию в MPI. Коммуникационная библиотека MPI является общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений [7, 8]. 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. 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. Количество процессов устанавливается во время инициализации программы. Каждый процесс идентифицируется по его относительному номеру в группе, т.е. последовательными целыми числами в диапазоне 0, …, *groupsize*–1.

К настоящему времени в СКАО интегрированного инструментария были включены следующие конструкции, предназначенные для проектирования MPI-программ: операция *m*-местной асинхронной MPI-дизъюнкции; базисные операторы передачи и приема сообщений между отдельными процессами; базисный оператор, возвращающий количество полученных в сообщении данных; базисный оператор сборки данных от нескольких процессов; базисные операторы начала отсчета времени и вывода информации о времени, истекшем с начала выполнения программы.

Операция *m*-местной асинхронной MPI-дизъюнкции состоит в параллельном выполнении *m* параллельных процессов. Ее естественно-лингвистическое представление имеет вид:

```
ПАРАЛЛЕЛЬНО_MPI(i = 0, ..., m-1)
(
   "оператор1"
)
```

где “оператор1” – операторная переменная, вместо которой в процессе проектирования указываются операторы, выполняемые в рамках отдельного процесса; *i*, *m* – параметры операции.

Отображение на языке программирования C для данной операции, хранящееся в СКАО, является таким:

```
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Get_processor_name(processor_name, &namelen);
^оператор1^
MPI_Finalize();
```

Приведенный фрагмент программной реализации содержит вызовы таких функций MPI: функции инициализации MPI_Init (которой передаются аргументы функции main, полученные из командной строки); функции MPI_Comm_size определения размера proc_num группы процессов; функции MPI_Comm_rank определения номера myrank текущего процесса; функции MPI_Get_processor_name определения идентификатора процессора, а также функции завершения MPI_Finalize. В приведенном программном шаблоне указана также строка “^оператор1^”, вместо которой в процессе генерации будет подставлен код реализации операнда асинхронной MPI-дизъюнкции, а именно, операторы, осуществляющие параллельную обработку. Предполагается, что переменные proc_num и myrank определены в блоке глобальных переменных схемы алгоритма.

Естественно-лингвистическая форма базисного оператора передачи данных процессу имеет вид:

```
"Послать данные (buf) в количестве (count) типа (datatype) процессу (proc_no)".
```

Данный базисный оператор соответствует MPI-функции блокирующей передачи сообщения MPI_Send. Напомним [8], что в языке С данная функция имеет спецификацию

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

где buf – адрес начала расположения пересылаемых данных; count – число пересылаемых элементов; datatype – тип посылаемых элементов; dest – номер процесса-получателя в группе, связанной с коммуникатором comm; tag – идентификатор сообщения; comm – коммуникатор области связи. Функция MPI_Send выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm.

Шаблон на языке программирования С для приведенного базисного оператора передачи в СКАО такой:

```
MPI_Send(%1, %2, %3, %4, myrank + 98, MPI_COMM_WORLD),
```

где MPI_COMM_WORLD – предопределенный идентификатор группы, состоящей их всех процессов приложения.

Для облегчения использования базисного оператора по сравнению с функцией MPI_Send в нем присутствуют лишь три параметра. Имена остальных указаны в фрагменте отображения на языке программирования и являются фиксированными. Приведенная естественно-лингвистическая форма описания базисного элемента содержит в себе имена формальных параметров, указанные в скобках (например, var_name). Признаком формального параметра в реализации является символ “%”, за которым следует порядковый номер параметра в тексте базисного оператора (например, %1). В процессе генерации программы значения параметров подставляются вместо соответствующих формальных параметров в тексте отображения на языке программирования С. Например, оператор

```
"Послать данные (Primes) в количестве (PrimesCount) типа (MPI_UNSIGNED_LONG) процессу (0)"
```

будет отображен в фрагмент программной реализации

```
MPI_Send(Primes, PrimesCount, MPI_UNSIGNED_LONG, 0, myrank + 98, MPI_COMM_WORLD).
```

Естественно-лингвистическая форма базисного оператора приема данных от процесса является следующей:

```
"Получить данные (buf) в количестве (count) типа (datatype) от процесса (proc_no)".
```

Данный базисный оператор соответствует MPI-функции блокирующего приема MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status) [8]. Данная функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm; status – переменная для хранения атрибутов принятого сообщения. Предполагается, что переменная status определена в блоке глобальных описаний схемы алгоритма.

Отображение на языке программирования С для базисного оператора приема является следующим:

```
MPI_Recv(%1, %2, %3, %4, %4 + 98, MPI_COMM_WORLD, &status).
```

Для определения числа элементов, фактически полученных в сообщении (с помощью вышеприведенного оператора приема), в СКАО введен базисный оператор, присваивающий переменной var_name количество полученных данных

```
"Присвоить переменной (var_name) количество принятых в сообщении элементов типа (mpi_type)".
```

Данный оператор соответствует функции MPI_Get_count. Его реализация на языке С в СКАО имеет вид

```
MPI_Get_count(&status, %2, &%1).
```

Вышерассмотренные базисные операторы передачи и приема сообщений соответствуют функциям, реализующим коммуникационные операции типа точка-точка. В СКАО введен также базисный оператор осуществляющий коллективную операцию – сборки данных от нескольких процессов

```
"Собрать данные (sendbuf)(datatype) длиной (count) от всех процессов в переменной (recvbuf) процесса (root_proc)",
```

который соответствует функции сбора данных MPI_Gather. Его отображение на языке С является следующим:

```
MPI_Gather(%1, %3, %2, %4, %3, %2, %5, MPI_COMM_WORLD).
```

При выполнении операции MPI_Gather все процессы посылают содержимое своего блока данных sendbuf процессу с номером root_proc. Длина блоков (count) предполагается одинаковой. Процесс с номером root_proc получает сообщения, располагая их в буфере recvbuf в порядке возрастания номеров процессов.

СКАО содержит также базисный оператор начала отсчета времени “Засечь время” и оператор “Вывести информацию о времени, истекшем с начала обработки”. В программных реализациях данных операторов использована MPI-функция отсчета времени MPI_Wtime [8], которая возвращает астрономическое время в сек., прошедшее с некоторого момента в прошлом (точки отсчета).

Аналогичным образом в СКАО могут быть введены и другие базисные операторы, соответствующие MPI-операциям, например, операторы неблокирующей передачи и приема сообщений и операторы коллективных взаимодействий процессов [7, 8].

Пример 1. Далее приведена параллельная схема нахождения простых чисел на отрезке от 1 до MaxNumbers, предназначенная для реализации в MPI. Схема состоит из блока определения глобальных данных, основного составного оператора (который соответствует функции main в языке программирования С), составного оператора “Поиск(ProcessNum)”, используемого каждым процессом для поиска простых чисел в своем блоке и сборки данных от процессов “Собрать_данные”. Операторы, входящие в “ОсновнойСоставнойОператор” включены в вышерассмотренную операцию асинхронной MPI-дизъюнкции.

```
СХЕМА НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ_MPI

"ОпределениеГлобальныхДанных"
==== "Определить массив (Primes) тип (unsigned long)";
     "Определить переменную (MaxNumbers) типа (unsigned long)";
     "Определить переменную (PrimesCount) типа (unsigned long)";
     "Определить массив (PrimesRcv) тип (unsigned long)";
     "Определить переменную (BlockSize) типа (unsigned long)";
     "Определить переменную (proc_num) типа (int)";
     "Определить переменную (myrank) типа (int)";
     "Определить переменную (status) типа (MPI_Status)"

"ОсновнойСоставнойОператор"
==== ПАРАЛЛЕЛЬНО_MPI(myrank = 0, ..., proc_num-1)
     (
        "Чтение входных параметров программы"
        ЗАТЕМ
        (BlockSize := MaxNumbers / proc_num)
        ЗАТЕМ
        "Инициализация данных"
        ЗАТЕМ
        ЕСЛИ 'Номер процесса = (0)'
        ТО "Засечь время"
           ЗАТЕМ
           "Вывести сообщение (Finding prime numbers from 1 to) и целое (MaxNumbers)"
        КОНЕЦ ЕСЛИ
        ЗАТЕМ
        "(PrimesCount) := (0)"
        ЗАТЕМ
        "Поиск(myrank)"
        ЗАТЕМ
        "Собрать_данные"
        ЗАТЕМ
        ЕСЛИ 'Номер процесса = (0)'
        ТО "Вывести сообщение (Found prime numbers:) и целое (PrimesCount)"
           ЗАТЕМ
           "Вывести информацию о времени, истекшем с начала обработки"
           ЗАТЕМ
           "Записать информацию о результатах выполнения программы"
        КОНЕЦ ЕСЛИ
     )


"Собрать_данные"
==== ЛОКАЛЬНЫЕ_ПЕРЕМЕННЫЕ
     (
       "Определить переменную (i) типа (int)";
```

```
       "Определить переменную (count) типа (int)";
       "Определить переменную (k) типа (unsigned long)";
       "Определить переменную (j) типа (unsigned long)"
     )
     ЗАТЕМ
     ЕСЛИ 'Номер процесса = (0)'
     ТО "(k) := (PrimesCount)"
        ЗАТЕМ
        ДЛЯ '(i) от (1) до (proc_num - 1)'
        ЦИКЛ
          "Получить данные (PrimesRcv) в количестве (BlockSize / 2 + 1) типа
(MPI_UNSIGNED_LONG) от процесса (i)"
          ЗАТЕМ
          "Присвоить переменной (count) количество принятых в сообщении элементов типа
(MPI_UNSIGNED_LONG)"
          ЗАТЕМ
          ЕСЛИ '(count) > (0)'
          ТО
            ДЛЯ '(j) от (0) до (count - 1)'
            ЦИКЛ
              "(Primes[k]) := (PrimesRcv[j])"
              ЗАТЕМ
              "Увеличить (k) на (1)"
            КОНЕЦ ЦИКЛА
          КОНЕЦ ЕСЛИ
        КОНЕЦ ЦИКЛА
        ЗАТЕМ
        "(PrimesCount) := (k)"
     ИНАЧЕ "Послать данные (Primes) в количестве (PrimesCount) типа (MPI_UNSIGNED_LONG)
процессу (0)"
     КОНЕЦ ЕСЛИ

"Поиск(ProcessNum)"
==== ЛОКАЛЬНЫЕ_ПЕРЕМЕННЫЕ
     (
       "Определить переменную (number) типа (unsigned long)";
       "Определить переменную (start) типа (unsigned long)";
       "Определить переменную (end) типа (unsigned long)";
       "Определить переменную (stride) типа (unsigned long)";
       "Определить переменную (factor) типа (unsigned long)"
     )
     ЗАТЕМ
     ЕСЛИ 'Номер процесса = (0)'
     ТО "(Primes[PrimesCount]) := (2)"
        ЗАТЕМ
        "Увеличить (PrimesCount) на (1)"
     КОНЕЦ ЕСЛИ
     ЗАТЕМ
     "(start) := (2 * ProcessNum + 1)"
     ЗАТЕМ
     "(end) := (MaxNumbers)"
     ЗАТЕМ
     "(stride) := (2 * proc_num)"
     ЗАТЕМ
     ЕСЛИ '(start) = (1)'
     ТО "(start) := (start) + (stride)"
     КОНЕЦ ЕСЛИ
     ЗАТЕМ
     "(number) := (start)"
     ЗАТЕМ
     ПОКА НЕ '(number) >= (end)'
     ЦИКЛ
        "(factor) := (3)"
        ЗАТЕМ
        ПОКА НЕ 'Остаток от деления (number) на (factor) = (0)'
        ЦИКЛ
           "(factor) := (factor) + (2)"
        КОНЕЦ ЦИКЛА
        ЗАТЕМ
        ЕСЛИ '(factor) = (number)'
        ТО "(Primes[PrimesCount]) := (number)"
```

```
            ЗАТЕМ
            "(PrimesCount) := (PrimesCount) + (1)"
         КОНЕЦ ЕСЛИ
         ЗАТЕМ
         "(number) := (number) + (stride)"
      КОНЕЦ ЦИКЛА

КОНЕЦ СХЕМЫ НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ_MPI
```

Приведенный алгоритм состоит в параллельном выполнении proc_num процессов, каждый из которых выполняет поиск простых чисел на своем отрезке (блоке) с помощью оператора “Поиск(ProcessNum)” и заносит их в массив Primes. В данном операторе осуществляется проверка каждого нечетного числа из отрезка, является ли оно нацело делимым на меньшие нечетные числа. Начало и конец отрезка задается в переменных start и end, а шаг, с которым обрабатываются числа – в переменной stride. Эти переменные устанавливаются таким образом, чтобы все процессы выполняли примерно одинаковый объем работ. Обрабатываемые блоки чисел являются перемежающимися, т.е. каждый процесс работает как на больших, так и на малых числах [2]. После того, как обработка отрезка завершена, каждый процесс (кроме нулевого), отсылает массив найденных чисел Primes корневому процессу с помощью вышерассмотренного базисного оператора передачи данных. В свою очередь, нулевой процесс получает данные массивы в переменной PrimesRcv и последовательно переписывает каждый массив в результирующий массив Primes. Количество чисел, которое будет получено от каждого процесса, заранее неизвестно, поэтому для получения информации о количестве фактически полученных данных от каждого процесса используется базисный оператор, соответствующий функции MPI_Get_count, также вышерассмотренный. Результаты выполнения программы с указанием времени выполнения записываются в файл.

По рассмотренной параллельной схеме в интегрированном инструментарии была сгенерирована MPI-программа на языке С. В процессе синтеза управляющие конструкции САА-схемы алгоритма автоматически отображены ДСП-конструктором в соответствующие операторы языка программирования, а вместо базисных операторов и предикатов подставлены их реализации на том же языке из СКАО. Составные операторы в тексте программы представлены как подпрограммы (функции). Результаты выполнения данной программы на вычислительном кластере Института кибернетики НАН Украины рассмотрены в разделе 2.

2. Применение системы TermWare для трансформации последовательных программ в параллельные

В работах [2, 3] рассмотрено использование системы символьных вычислений TermWare [9] совместно с инструментарием «ИПС» для автоматизированного выполнения преобразований (трансформаций) многопоточных алгоритмов, направленного, в частности, на их оптимизацию по времени выполнения. К данным преобразованиям относятся – трансформация последовательных программ в многопоточные и исключение накладных расходов на синхронизацию в параллельных программах. Основой функционирования системы TermWare в рамках «ИПС» является создание и применение систем правил вида

$$f(x_1, \ldots, x_n) \rightarrow g(x_1, \ldots, x_n),$$

где f и g – термы, зависящие от переменных $x_1, x_2, \ldots, x_n$. Каждая система правил соответствует некоторому преобразованию схемы алгоритма. Трансформация алгоритма с помощью TermWare требует предварительного преобразования дерева конструирования алгоритма, построенного в ДСП-конструкторе (см. раздел 1, а также [1, 5]), в терм.

Пример 2. Рассмотрим применение системы TermWare для преобразования последовательного алгоритма нахождения простых чисел в параллельную схему, ориентированную на MPI вычисления, приведенную в примере 1. Естественно-лингвистическая форма последовательного алгоритма имеет вид:

```
СХЕМА НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ_ПОСЛЕДОВАТЕЛЬНОЕ

"ОпределениеГлобальныхДанных"
==== "Определить массив (Primes) тип (unsigned long)";
     "Определить переменную (MaxNumbers) типа (unsigned long)";
     "Определить переменную (PrimesCount) типа (unsigned long)"

"ОсновнойСоставнойОператор"
==== "Чтение входных параметров программы"
     ЗАТЕМ
     "Инициализация данных"
     ЗАТЕМ
     "Засечь время"
     ЗАТЕМ
     "Вывести сообщение (Finding prime numbers from 1 to) и целое (MaxNumbers)"
```

```
     ЗАТЕМ
     "(PrimesCount) := (0)"
     ЗАТЕМ
     "Поиск"
     ЗАТЕМ
     "Вывести сообщение (Found prime numbers:) и целое (PrimesCount)"
     ЗАТЕМ
     "Вывести информацию о времени, истекшем с начала обработки"
     ЗАТЕМ
     "Записать информацию о результатах выполнения программы"

"Поиск"
==== ЛОКАЛЬНЫЕ_ПЕРЕМЕННЫЕ
     (
       "Определить переменную (number) типа (unsigned long)";
       "Определить переменную (start) типа (unsigned long)";
       "Определить переменную (end) типа (unsigned long)";
       "Определить переменную (stride) типа (unsigned long)";
       "Определить переменную (factor) типа (unsigned long)"
     )
     ЗАТЕМ
     "(Primes[PrimesCount]) := (2)"
     ЗАТЕМ
     "Увеличить (PrimesCount) на (1)"
     ЗАТЕМ
     "(start) := (3)"
     ЗАТЕМ
     "(end) := (MaxNumbers)"
     ЗАТЕМ
     "(stride) := (2)"
     ЗАТЕМ
     "(number) := (start)"
     ЗАТЕМ
     ПОКА НЕ '(number) >= (end)'
     ЦИКЛ
        "(factor) := (3)"
        ЗАТЕМ
        ПОКА НЕ 'Остаток от деления (number) на (factor) = (0)'
        ЦИКЛ
           "(factor) := (factor) + (2)"
        КОНЕЦ ЦИКЛА
        ЗАТЕМ
        ЕСЛИ '(factor) = (number)'
        ТО "(Primes[PrimesCount]) := (number)"
           ЗАТЕМ
           "(PrimesCount) := (PrimesCount) + (1)"
        КОНЕЦ ЕСЛИ
        ЗАТЕМ
        "(number) := (number) + (stride)"
     КОНЕЦ ЦИКЛА

КОНЕЦ СХЕМЫ НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ_ПОСЛЕДОВАТЕЛЬНОЕ
```

Приведенная схема, как и алгоритм из примера 1, осуществляет поиск простых чисел на отрезке от 1 до MaxNumbers с помощью составного оператора “Поиск” и заносит результаты в массив Primes.

Для выполнения трансформаций данного алгоритма в TermWare, его САА-схема записана в виде терма:

```
ROOT
(
   Globals (
     THEN (
     Array(Primes, unsigned_long),
     THEN (
     Variable(MaxNumbers, unsigned_long),
     Variable(PrimesCount, unsigned_long)
   ))),

   MAIN (
       THEN (
       THEN (
```

```
        ReadInputParameters,
        THEN (
        DataInitialization,
        THEN (
        NoteTime,
        THEN (
        OutputMessage(string_1, MaxNumbers),
        THEN (
        Assignment(PrimesCount, 0),
        THEN (
        CALL(Search),
        THEN (
        OutputMessage(string_2, PrimesCount),
        THEN (
        OutputExpiredTime,
        WriteExecutionResults
        )))))))),
        END(MAIN))
    ),

    Search
    (
      Search_realization
    )
)
```

Составные операторы схемы (“ОпределениеГлобальныхДанных”, “ОсновнойСоставнойОператор”, “Поиск”) представлены в виде подтермов терма Root (Globals, Main, Search). Строки сообщений, выдаваемых в программе, обозначены string_1 и string_2. В данном примере подробно рассматриваются только преобразования термов Root, Globals и Main, поэтому детализация терма Search не приведена.

Рассмотрим трансформации, которые необходимо применить к упомянутым термам для преобразования последовательного алгоритма в параллельный. При этом применяется такая совокупность правил:

1. `THEN($0, Variable (PrimesCount, unsigned_long)) → THEN($0,`
   `THEN(Variable (PrimesCount, unsigned_long),`
   `THEN(Array(PrimesRcv, unsigned_long),`
   `THEN(Variable(BlockSize, unsigned_long),`
   `THEN(Variable(proc_num, int),`
   `THEN(Variable(myrank, int),`
   `THEN(Variable(status, MPI_Status))))))))`
2. `MAIN(THEN(THEN($0, $1), $2)) → MAIN(THEN(`
   `Parallel_MPI(Parameters(myrank, 0, minus(proc_num, 1)), THEN($0, $1)), $2))`
3. `THEN(ReadInputParameters, THEN($0, $1)) → THEN(THEN(ReadInputParameters,`
   `Assignment(BlockSize, divide(MaxNumbers, proc_num))), THEN($0, $1))`
4. `THEN (NoteTime, THEN (OutputMessage(string_1, MaxNumbers), THEN ($0, $1))) →`
   `THEN (THEN (NoteTime, OutputMessage(string_1, MaxNumbers)), THEN ($0, $1));`

   `THEN(THEN(NoteTime, OutputMessage(string_1,MaxNumbers)), $0) →`
   `THEN(IF(eq(myrank, 0), THEN(NoteTime, OutputMessage(string_1,MaxNumbers))), $0)`
5. `Search(Search_realization) → Search(Parameters(ProcessNum), Search_realization)`
6. `THEN(CALL(Search), $0) → THEN(THEN(CALL(Search(myrank)), CALL(GatherData)), $0)`
7. `THEN($0, THEN(OutputMessage(string_2, PrimesCount),`
   `THEN(OutputExpiredTime, WriteExecutionResults))) →`
   `THEN($0, IF(eq(myrank, 0), THEN(OutputMessage(string_2, PrimesCount),`
   `THEN(OutputExpiredTime, WriteExecutionResults))))`
8. `ROOT($0, $1, $2) → ROOT($0, $1, $2, GatherData(GatherData_realization))`

В приведенных правилах знаком “$” обозначены идентификаторы пропозициональных переменных. Правило 1 добавляет необходимые в MPI-программе глобальные определения, а именно: массив PrimesRcv, переменные BlockSize, proc_num, myrank, status (см. раздел 1). Правило 2 включает операторы терма Main в операцию асинхронной MPI-дизъюнкции. Правило 3 добавляет после базисного оператора ReadInputParameters чтения входных параметров программы оператор, вычисляющий размер BlockSize блока, обрабатываемого отдельным процессом. Два правила, сгруппированные под номером 4, необходимы для включения двух операторов – засечки времени NoteTime и вывода сообщения OutputMessage, в блок условного оператора ЕСЛИ ‘Номер процесса = (0)’ ТО … КОНЕЦ ЕСЛИ. Необходимость ввода данного условного оператора связана с тем, что указанные операторы должны выполняться только в корневом процессе. Правило 5 добавляет

в функцию Search параметр ProcessNum (номер процесса). Правило 6 добавляет в вызов функции Search формальный параметр myrank, а также включает после вызова функции Search вызов функции GatherData, предназначенной для сборки результатов, полученных от процессов. Правило 7 аналогично правилу 4, включает три оператора – OutputMessage вывода сообщения о результатах выполнения программы, OutputExpiredTime вывода информации о времени, истекшем с начала обработки, WriteExecutionResults записи информации о результатах выполнения программы, в блок условного оператора с проверкой условия ‘Номер процесса = (0)’. По окончанию правило 8 добавляет в схему составной оператор “Собрать_данные” (GatherData), где GatherData_realization – детализация (подтермы) терма GatherData.

В результате применения этих правил исходный терм преобразуется в следующий:

```
ROOT(
   Globals(
      THEN(Array(Primes, unsigned_long),
      THEN(Variable(MaxNumbers, unsigned_long),
      THEN(Variable(PrimesCount, unsigned_long),
      THEN(Array(PrimesRcv, unsigned_long),
      THEN(Variable(BlockSize, unsigned_long),
      THEN(Variable(proc_num, int),
      THEN(Variable(myrank, int),
      THEN(Variable(status, MPI_Status)))))))))
   ),

   MAIN(
      THEN(
      Parallel_MPI(Parameters(myrank, 0, minus(proc_num, 1)),
         THEN(
         THEN(ReadInputParameters,
         Assignment(BlockSize, divide(MaxNumbers, proc_num))),
         THEN(DataInitialization,
         THEN(
         IF(eq(myrank, 0),
            THEN(NoteTime,
            OutputMessage(string_1, MaxNumbers))
         ),
         THEN(Assignment(PrimesCount, 0),
         THEN(
         THEN(CALL(Search(myrank)),
         CALL(GatherData)),
         IF(eq(myrank, 0),
            THEN(OutputMessage(string_2, PrimesCount),
            THEN(OutputExpiredTime,
            WriteExecutionResults)))
         )))))
      ),
      END(MAIN))
   ),

   Search(
      Parameters(ProcessNum),
      Search_realization
   ),

   GatherData(
      GatherData_realization
   )
)
```

К составному оператору “Поиск” также должно быть применено несколько правил. Так, базисные операторы “(Primes[PrimesCount]) := (2)” и “Увеличить (PrimesCount) на (1)” должны быть включены в блок условного оператора с проверкой номера процесса, аналогично применению вышерассмотренного правила 7. Значения, присваиваемые переменным start, end и stride также должны быть соответствующим образом изменены (см. параллельную САА-схему нахождения простых чисел).

В результате выполнения всех указанных трансформаций и обратного преобразования терма в САА-схему была получена параллельная схема НАХОЖДЕНИЕ_ПРОСТЫХ_ЧИСЕЛ_MPI (см. раздел 1). На рисунке показаны результаты выполнения MPI-программы, сгенерированной по данной схеме, на вычислительном кластере СКИТ-3 Института кибернетики НАН Украины. СКИТ-3 представляет собой 127-узловой кластер на многоядерных процессорах – 75 узлов на 2-ядерных процессорах Intel Xeon 5160 и 52 узла на 4-ядерных

процессорах Xeon 5345 [6]. Полученные результаты показывают хорошую степень распараллеливаемости вычислений.

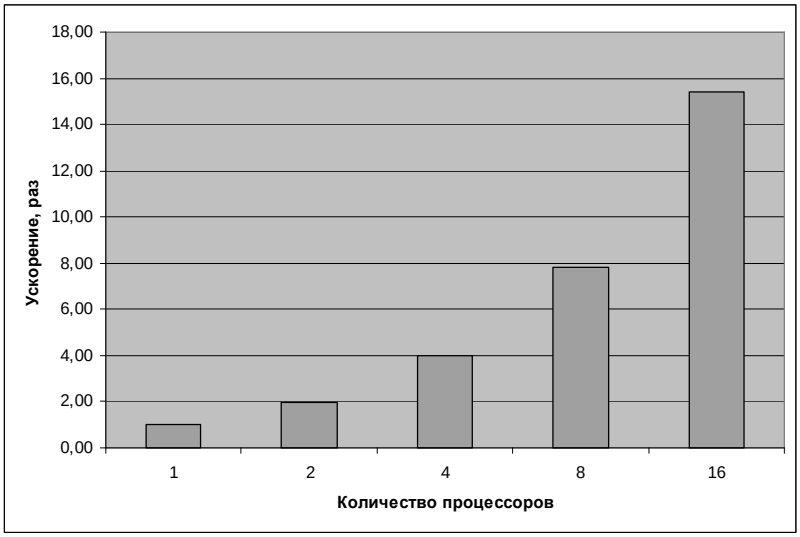

Рисунок. Зависимость мультипроцессорного ускорения от количества используемых процессоров (300000 чисел)

Выводы

Предложен метод проектирования MPI-программ на основе алгеброалгоритмических спецификаций и подход к генерации данных программ в разработанном интегрированном инструментарии «ИПС». Разработана методика применения системы перезаписи термов TermWare для автоматизированной трансформации последовательных схем алгоритмов в параллельные, ориентированные на передачу сообщений. Проведен эксперимент по выполнению синтезированной в интегрированном инструментарии параллельной программы на кластерной мультипроцессорной архитектуре, результаты которого показывают хорошую степень распараллеливаемости вычислений.

1. *Дорошенко Е.А., Яценко Е.А.* О синтезе программ на языке Java по алгеброалгоритмическим спецификациям // Проблемы программирования. – 2006. – № 4. – С. 58–70.
2. *Дорошенко Е.А., Жереб К.А., Яценко Е.А.* Формализованное проектирование эффективных многопоточных программ // Проблемы программирования. – 2007. – № 1. – С. 17–30.
3. *Дорошенко Е.А., Жереб К.А., Яценко Е.А.* Об оценке сложности и координации вычислений в многопоточных программах // Проблемы программирования. – 2007. – № 2. – С. 41–55.
4. *Цейтлин Г.Е.* Введение в алгоритмику. – Киев: Сфера, 1998. – 311 с.
5. *Андон Ф.И., Дорошенко Е.А., Цейтлин Г.Е., Яценко Е.А.* Алгеброалгоритмические модели и методы параллельного программирования. – Киев: Академпериодика, 2007. – 631 с.
6. *Суперкомпьютеры* Института кибернетики им. В.М. Глушкова НАН Украины. О кластерах. – http://cluster.icyb.kiev.ua/about.html.
7. *Шпаковский Г.И., Серикова Н.В.* Программирование для многопроцессорных систем в стандарте MPI. – Минск: БГУ, 2002. – 323 с.
8. *MPI*: Стандарт интерфейса передачи сообщений (MPI: A Message-Passing Interface Standard): Пер. с англ. / Под ред. Г.И. Шпаковского. – Минск: БГУ, 2001. – 246 с.
9. *Дорошенко А.Е., Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблемы программирования. – 2005. – № 4. – С. 718–727.