ИНФОРМАТИКА

2005 **июль-сентябрь** **№ 3**

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ И РАСПРЕДЕЛЕННАЯ ОБРАБОТКА

УДК 519.68, 519.7

Н.Р. Торопов

ПАРАЛЛЕЛЬНЫЕ ЛОГИКО-КОМБИНАТОРНЫЕ ВЫЧИСЛЕНИЯ В СРЕДЕ MPI

Задачи, решаемые на мультипроцессорных системах, разделяются на два характерных класса. Для каждого из них предлагается программная модель выполнения параллельных вычислений на суперкомпьютере семейства СКИФ в среде MPI с базовым языком C++. Для организации эффективного информационного обмена между процессами при параллельном решении логико-комбинаторных задач разработан специальный класс операций над сообщениями, обеспечивающий компактную упаковку логических объектов в посылаемые сообщения и распаковку их из принятых сообщений.

Введение

Большинство логических задач, возникающих в различных областях науки и техники (таких, как логическое проектирование, техническая и медицинская диагностика, распознавание образов, криптография, лингвистика, искусственный интеллект и т. п.), носит комбинаторный характер.

Отличительной особенностью логико-комбинаторных задач является то, что основными объектами преобразований для них являются не числа, как для задач численного анализа, а логические объекты (логические переменные, булевы и k-значные векторы, булевы и k-значные матрицы, графы и т. д.). Для задач рассматриваемого класса характерно также то, что объемы перерабатываемой информации при поиске решений относительно невелики, а сам процесс переработки достаточно сложен, так как зачастую сопряжен с необходимостью перебора и анализа значительного числа вариантов промежуточных решений. Поскольку число перебираемых вариантов растет экспоненциально с возрастанием размерности параметров решаемых задач, становится проблематично решать такие задачи большой размерности за приемлемое время даже с использованием самых быстродействующих персональных компьютеров. Между тем трудоемкость порождаемых практикой задач постоянно возрастает, причем не только «количественно» (за счет увеличения размерности параметров), но и «качественно» (за счет увеличения алгоритмической сложности).

В данной ситуации могут помочь или разработка принципиально новых методов решения таких задач, или использование естественного параллелизма в известных методах при реализации их на мультипроцессорных системах, тем более что оба названных выше свойства логико-комбинаторных задач делают последние потенциально пригодными для их эффективного параллельного решения, если перебираемые варианты промежуточных решений допускают независимую параллельную обработку.

Далее речь пойдет о применении мультипроцессорных систем семейства СКИФ [1] для решения логико-комбинаторных задач в среде библиотеки MPI (Message Passing Interface) [2] с базовым языком программирования C++. В разд. 1 приводится классификация задач, решаемых на мультипроцессорных системах. Для организации эффективного информационного взаимодействия между процессами при параллельном решении логико-комбинаторных задач в разд. 2 предлагается класс **CBUF** операций над сообщениями, обеспечивающий образование приемо-передающих буферов для сообщений заданной емкости, а также компактную упаковку логических объектов в посылаемое сообщение и распаковку их из принятого сообщения. Класс **CBUF** базируется на следующих дружественных ему классах логических объектов: булевых векторах (**CBV**) и матрицах (**CBM**) [3], а также троичных векторах (**CTV**) и матрицах (**CTM**) [4].

В разд. 3 приводится минимальный перечень MPI-операций, достаточный для разработки простых программ параллельных вычислений, и предлагаются две программные MPI-модели,

которые могут служить шаблонами, помогающими создавать из разработанных на языке C++ модулей параллельные программы для решения логико-комбинаторных задач из рассматриваемых в разд. 1 классов.

1. Типы параллельно решаемых задач

Для параллельного решения любой задачи на мультипроцессорной системе семейства СКИФ выделяется *группа* из затребованного количества N процессов. Хотя процессы внутри группы, идентифицируемые своими номерами 0, 1, …, $N-1$, принципиально не отличаются друг от друга, тем не менее процесс номер 0 часто выделяется особо и считается *корневым*, а остальные $N-1$ процессов – *подчиненными*.

Корневой процесс получает из внешней среды исходные данные для решаемой задачи, декомпозирует ее на $N-1$ частных подзадач и распределяет их между подчиненными процессами, которые решают каждый свою задачу параллельно и посылают результаты решений корневому. Из полученных частичных решений корневой процесс формирует итоговое решение задачи в целом.

Все задачи по способам их декомпозиции на частные подзадачи, решаемые подчиненными процессами, и по правилам формирования из частичных решений итогового решения исходной задачи распадаются на два следующих класса.

Задачи класса *J* – это задачи, которые решаются *совместно* (*jointly*) $N-1$ процессами. Каждый из них решает свою *частную задачу* над своими исходными данными до конца, а корневой процесс *собирает* результаты *частичных решений* и формирует из них *итог решения* общей задачи. Поскольку время решения задачи из класса *J* определяется временем «*самого медленного*» (последнего из закончивших работу) процесса, то при декомпозиции исходной задачи на частные подзадачи следует стремиться к наиболее равномерному распределению нагрузки между процессами.

Задачи класса *C* отличаются от задач класса *J* тем, что решение любой частной задачи, полученное любым из процессов в ходе *соревнования* (*competition*) по времени с другими, является решением общей задачи. Когда какой-либо из процессов находит решение, все остальные должны прекратить работу, т. е. время решения задачи класса *C* определяется «*самым быстрым*» процессом.

Естественно, что любая декомпозиция исходной задачи на подзадачи не должна приводить к потере решений, удовлетворяющих заданным критериям. Если же в пространстве поиска, определенном по заданным исходным условиям задачи, таких решений не оказывается, то задача из класса *C* переходит в класс *J*, так как для доказательства отсутствия решений каждый из подчиненных процессов должен обследовать все свое подпространство до конца.

К классу *C* относятся, например, задачи, решаемые *конкурирующими* алгоритмами, когда подчиненные процессы решают одну и ту же исходную задачу, каждый своим алгоритмом. Пополняют этот класс также задачи, для решения которых используются рандомизированные и приближенные алгоритмы с конкурирующими эвристиками.

Следует заметить, что мультипроцессорные структуры с индивидуальной памятью (к каким относится семейство СКИФ) ориентированы в основном на задачи класса *J* (реализованы эффективные операции по сбору данных корневым процессом у всех подчиненных, имеются операции синхронизации процессов и пр.). В то же время при решении задач класса *C* приходится изыскивать специальные приемы для прерывания активных процессов в тот момент, когда один из процессов нашел решение.

Одним из таких приемов может служить вставка во внутренние циклы программ, выполняющихся в подчиненных процессах, операции *MPI_IProbe*, реагирующей на факт посылки корневым процессом некоторого сигнала прерывания. В случае «положительной пробы» (когда к моменту проверки корневой процесс уже послал сигнал прерывания) подчиненный процесс должен обеспечить прием этого сигнала и прекратить работу. Место вставки выбирается с та-

ким расчетом, чтобы обеспечить оптимальную скорость реакции подчиненных процессов на сигнал прерывания при допустимых накладных расходах.

2. Операции над сообщениями с логическими объектами

Параллельное решение задачи на мультипроцессорной системе предполагает взаимодействие между процессами, участвующими в решении. В системах с индивидуальной памятью, к каким относится семейство СКИФ, информационное взаимодействие осуществляется через обмен сообщениями.

Сообщение – это некоторая совокупность в общем случае разнотипных объектов, которыми обмениваются параллельные процессы при совместном решении задачи. Если в сообщении представляются объекты, тип которых не является элементарным для базового языка программирования (в нашем случае C++), то для организации такого обмена необходимо предусмотреть специальные *буфера* (передающие и приемные) и обеспечить упаковку посылаемых объектов в передающие буфера и распаковку полученных объектов из приемных буферов.

Во многих случаях для организации обменов в параллельных вычислениях достаточно образовать по одному буферу на каждый процесс, используя его как для приема, так и для передачи сообщений.

Далее предлагается описание класса **CBUF**, обеспечивающего образование в памяти процессов буферов указанной емкости, а также компактную упаковку в буфер и распаковку из него таких объектов, как целое число, булев или троичный вектор, булева или троичная матрица, массив троичных матриц или содержимое другого буфера. При этом классы **CBV** (булев вектор), **CTV** (троичный вектор), **CBM** (булева матрица), **CTM** (троичная матрица), **CMTM** (массив троичных матриц) объявлены как дружественные. Заметим, что для представления перечисленных типов объектов в буфере используются те же формы, что приняты для представления объектов соответствующих классов в памяти [3, 4]. Например, n-мерный булев вектор располагается в k соседних байтах памяти, где $k =]n/8[$ – ближайшее к $n/8$ сверху целое число.

Атрибуты. В качестве атрибутов класса **CBUF** выступают **m_nBuf** – указатель (адрес начала) области памяти (**char***), в которой располагается буфер; **m_nIndex** – индекс (номер) обрабатываемого байта в сообщении; **m_nSize** – длина сообщения (в байтах), упакованного в буфере; **m_nMaxSize** – емкость буфера (максимально допустимая длина сообщения в байтах).

Конструкторы. Для образования объектов класса «буфер» определены следующие конструкторы: **CBUF**() – конструктор «пустого буфера», **CBUF** (*BufMaxSize*) – конструктор буфера указанной емкости *BufMaxSize* и **CBUF** (*Buf*) – конструктор копии некоторого другого буфера *Buf*.

Операции-члены класса CBUF распадаются на следующие группы.

Определение значений атрибутов **GetIndex** (текущего индекса), **GetSize** (длины сообщения), **GetMaxSize** (емкости буфера).

Установка заданных значений атрибутов **SetIndex** (текущего индекса) и **SetSize** (длины сообщения).

Упаковка **Set** в начале буфера объектов указанного в операнде типа (**Int**, **CBV**, **CTV**, пары (**CTV**, **CBV**)**, CBM**, **CTM**, **CMTM**, **CBUF**).

Добавление **Add** к концу представленного в буфере сообщения объектов указанного в операнде типа (**Int**, **CBV**, **CTV**, пары (**CTV**, **CBV**), **CBM**, **CTM**, **CMTM**, **CBUF**).

Все операции двух последних групп после упаковки указанного операнда возвращают новую длину **m_nSize** представленного в буфере сообщения. Фактически любая операция из группы **Set** функционально эквивалентна соответствующей паре операций **SetSize(0)** и **Add**. Например, операция **Set** (*Bm*) равносильна последовательности операций **Set** (0), **Add** (*Bm*).

Распаковка объекта (указанного в имени операции типа) из буфера, начиная с текущего значения индекса **m_nIndex** (или с нового его значения, заданного в операнде): **GetInt**, **GetBv**, **GetTv**, **GetTvBv**, **GetBm**, **GetTm**, **GetMtm**, **GetBuf**. Возвращают эти операции значения распа-

кованных из буфера объектов соответствующих типов, кроме операции **GetTvBv**, которая возвращает значение **m_nIndex**, а первый и второй операнды получают значения распакованных троичного и булева векторов. Заметим, что операции, имеющие дело с векторами, требуют явного задания их размерностей в соответствующих операндах.

В рамках класса **CBUF** реализована перезагрузка оператора присваивания (=), при этом в качестве операнда выступает другой буфер.

3. Параллельные C++ вычисления в среде MPI

Заметим, что в мультипроцессорных системах с индивидуальной памятью, к каким относится семейство СКИФ, текст всей программы перед ее исполнением чаще всего копируется в памяти каждого из *N* процессов группы. Все они включаются в работу одновременно и работают параллельно. При этом каждый из процессов должен безошибочно «распознавать» свои участки программы и выполнять только их, при необходимости обмениваясь сообщениями с другими процессами.

Все участки параллельной программы по субъектам их исполнения делятся на три типа: *общие*, исполняемые всеми процессами; *корневые*, исполняемые только корневым процессом, и *подчиненные*, исполняемые только подчиненными процессами.

Параллельные программы, при исполнении которых подчиненные процессы функционально неразличимы между собой, т. е. все они исполняют одни и те же участки программы, будем называть *однородными* в отличие от *неоднородных* программ, в которых подчиненные участки распадаются на несколько различных подтипов для отдельных функционально различимых подгрупп подчиненных процессов.

3.1. Минимальный набор MPI-операций для логико-комбинаторных задач

Любая MPI-программа с начинкой из модулей, разработанных на языке C++, начинается операцией инициализации MPI-среды

MPI_Init (&argc, &argv)

и заканчивается операцией завершения работы MPI

MPI_Finalize ().

Обе операции принадлежат общей части программы, т. е. исполняются всеми процессами. В начале общей части любой программы, после операций декларации некоторых общих переменных, в том числе **int** *NumProcs* (число процессов в группе) и **int** *myid* (собственный номер процесса), должна присутствовать операция определения числа процессов в группе

MPI_Comm_size (**MPI_COMM_WORLD**, &*NumProcs*)

и операция определения собственного номера, идентифицирующего данный процесс,

MPI_Comm_rank (**MPI_COMM_WORLD**, &*myid*).

Здесь и далее через **MPI_COMM_WORLD** обозначен коммуникатор, описывающий состав процессов данной группы и связи между ними. Для облегчения чтения служебные слова в базовом языке C++, а также в MPI-операциях и в операциях прикладных классов печатаются прямым жирным шрифтом, а идентификаторы операндов – курсивом.

Значение переменной *myid* является инструментом, помогающим «распараллелить» программу путем разбиения ее текста на отдельные блоки, исполняемые параллельно различными процессами, например:

if (!*myid*) {/* блок, исполняемый только корневым процессом */};

if (*myid*) {/* блок, исполняемый всеми подчиненными процессами */};

if (*myid* > 3) {/* блок, исполняемый *i*-м подчиненным процессом, если *i* > 3 */}.

Связь между процессами, решающими параллельно общую задачу, обеспечивается путем обмена сообщениями. Любой обмен является парным, так как предполагает наличие двух участвующих в нем субъектов. При исполнении любой корректно составленной программы все обмены должны быть завершены, т. е. посланное отправителем сообщение обязательно должно быть принято адресатом.

Если корневой процесс в состоянии оценить на основании исходных данных решаемой задачи значение *BufMaxSize* максимальной емкости буфера обмена сообщениями, то он посылает это значение всем процессам операцией «широковещательного обмена»

MPI_Bcast (&*BufMaxSize*, 1, **MPI_INT**, 0, **MPI_COMM_WORLD**),

где &*BufMaxSize* – адрес приемопередающего буфера корневого и подчиненных процессов; 1 – число элементов в посылаемом сообщении; **MPI_INT** – тип (в данном случае целое число) посылаемых в буфере элементов; 0 – номер процесса, передающего сообщение.

Заметим, что обращение к операции **MPI_Bcast**, исполняемой всеми процессами одновременно, должно располагаться в общем блоке программы. Если же оценка максимальной емкости буфера обмена невозможна, то каждый из процессов инициирует ее значение некоторой определенной константой по умолчанию

int *BufMaxSize* = MAX_SIZE_BUF.

Напомним, что в классе **CBUF**, описанном в разд. 2, имеется конструктор, обеспечивающий образование буфера *Buf* указанной емкости *BufMaxSize*, –

CBUF *Buf* (*BufMaxSize*).

Рассмотрим еще три MPI-операции, которые вместе с уже приведенными составят минимальное подмножество, достаточное для написания широкого круга программ параллельных логико-комбинаторных вычислений при решении задач классов *C* и *I*, базируясь на C++ модулях, в том числе разработанных в среде Windows.

Следующие две операции чаще других используются для организации парных обменов сообщениями в параллельных вычислениях.

MPI_Send (*address*, *count*, *datatype*, *destination*, *tag*, **MPI_COMM_WORLD**) – послать из буфера с адресом *address* сообщение из *count* элементов типа *datatype* с описателем *tag* (целое число из диапазона [0 ÷ 32767], приписанное к данному сообщению) процессу номер *destination* из группы, определяемой коммуникатором **MPI_COMM_WORLD**.

MPI_Recv (*address*, *maxsize*, *datatype*, *source*, *tag*, **MPI_COMM_WORLD**, &*status*) – принять в буфер с адресом *address* сообщение, представленное не более чем *maxsize* элементами типа *datatype* с описателем *tag*, от процесса номер *source* из группы, определяемой коммуникатором **MPI_COMM_WORLD**. Длина и другие характеристики принятого сообщения представляются значением операнда *status*, декларированного предварительно операцией

MPI_Status *status*.

Действительную длину *count* принятого сообщения в байтах можно определить, воспользовавшись операций

MPI_Get_count (&*status*, **MPI_BYTE**, &*count*).

Если в операции **MPI_Recv** номер *source* процесса-источника сообщения не определен явно (т. е. на месте четвертого операнда стоит **MPI_ANY_SOURCE**), то это означает, что ожидается сообщение от любого процесса из данной группы. Номер *source* процесса, пославшего принятое сообщение, можно определить операцией

source = *status*.**MPI_SOURCE**.

Аналогично, если в операции **MPI_Recv** на месте *tag* стоит **MPI_ANY_TAG**, то значение *tag* посланного сообщения определяется операцией

tag = *status*.**MPI_TAG**.

Еще одна важная, уже упоминавшаяся выше операция, вставляемая во внутренний цикл программы, которая выполняется в подчиненных процессах при решении задач класса *C*, –

MPI_Iprobe (*source*, *tag*, **MPI_COMM_WORLD**, &*Break*, &*status*).

Эта операция проверяет, не послал ли уже процесс номер *source* сообщение с заданным значением *tag*. Если проба положительна (т. е. во входном потоке есть посланное указанным процессом сообщение с заданным значением *tag*), то переменной *Break* присваивается отличное от нуля значение, а иначе *Break* = 0. Заметим, что в этой операции, так же, как и в операции **MPI_Recv**, на местах операндов *source* и *tag* могут стоять **MPI_ANY_SOURCE** и **MPI_ANY_TAG**. Это позволяет опробовать сообщения из произвольного источника и / или с произвольным значением *tag*.

3.2. Программная модель для решения задач класса J

Предлагается модель однородной параллельной программы для решения задач класса *J* с одним приемопередающим буфером обмена в каждом процессе.

```
      /* Общий блок */
{MPI_Init (&argc, &argv);                                   // Инициализация MPI
 MPI_Status status;                                         // Декларация объекта status
 int i, p, NumProcs, myid, BufMaxSize, size, count, tag;    // Декларация общих объектов
 CBV Pact (0, NumProcs, 1);          // Вектор Pact, «единицы» которого отмечают активные процессы
 MPI_Comm_size (MPI_COMM_WORLD,&NumProcs);                  // Определение числа процессов
 MPI_Comm_rank (MPI_COMM_WORLD, &myid);      // Определение идентификатора процесса


      /* Корневой блок */
 if (!myid)
 { /* Корневой процесс вводит исходные данные и по возможности определяет значение BufMaxSize
емкости буфера Buf. */}

      /* Общий блок */
/* Если значение BufMaxSize определено, то выполняется пара следующих операций. */
 MPI_Bcast (&BufMaxSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
 CBUF Buf (BufMaxSize);                                     // Образование буфера Buf в каждом из процессов
/* Если значение BufMaxSize не было определено, то оно задается по умолчанию некоторой определенной
константой MAX_SIZE_BUF.*/
 CBUF Buf (BufMaxSize = MAX_SIZE_BUF);

      /* Корневой блок */
 if (!myid)
 { /* Корневой процесс декомпозирует исходную задачу на p ≤ NumProcs частных задач и рассылает их
подчиненным процессам. При этом NumProc – p процессам посылаются «пустые» (длиной 0 байт) сооб-
щения с tag = 0. */
  for (i = 0; i < NumProcs; i ++)
  {tag = size = 0;
   if (i <= p)
   { /* Объекты, представляющие i-ю частную задачу, упаковываются в буфер Buf, при этом вычисляется общая
длина size подготовленного сообщения в байтах, i-й компоненте Pact присваивается значение 1 и tag = 1.*/}
   MPI_Send (Buf.m_nBuf, size, MPI_BYTE, i, tag, MPI_COMM_WORLD);
  }
/*Корневой процесс принимает частичные решения, полученные подчиненными процессами.*/
  while (!Pact.IsZero ())                   // Пока от всех процессов не будут получены решения
  {MPI_Recv (Buf.m_nBuf, BufMaxSize, MPI_BYTE, MPI_ANY_SOURCE, 2,
                MPI_COMM_WORLD, &status);
   i = status.MPI_SOURCE;                           // Определение процесса-источника сообщения
   Pact.SetBitAt (i, 0);                            // Отметка процесса, приславшего сообщение
   MPI_Get_count (&status, MPI_BYTE, &count);       // Определение длины сообщения
   Buf.SetSize (count);
/* По принятому от i-го процесса сообщению в буфере Buf формируются результаты частичного решения
i-й частной задачи. */
  }
/* Корневой процесс выдает общее решение задачи, сформированное из частичных решений. */
 }

      /* Блок подчиненных процессов */
 else
 {MPI_Recv (Buf.m_nBuf, BufMaxSize, MPI_BYTE, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);           // Получают сообщение с заданием
```

```
  if (status.MPI_TAG)                    // Если tag > 0, выполняются следующие операции
  {MPI_Get_count (&status, MPI_BYTE, &count);
   Buf.SetSize (count);
/* По принятому от корневого процесса сообщению в буфере Buf формируются объекты, представляющие исходные данные для частной задачи данного процесса, и инициируется выполнение модуля, решающего эту задачу. Результаты решения упаковываются в буфер Buf, вычисляется длина size сообщения, которое посылается корневому процессу с tag = 2. /*
   MPI_Send (Buf.m_nBuf, size, MPI_BYTE, 0, 2, MPI_COMM_WORLD);
  }
 }
      /* Общий блок */
 MPI_Finalize ();
}
```

Поскольку решение исходной задачи класса J в целом определяется по частичным решениям, полученным от всех $N-1$ подчиненных процессов, то итоговое «параллельное время» t_p определяется суммой

$$t_p = t_0 + max\ \{t_i\}\ \ (i \in \{1, 3, \ldots, N-1\}),$$

где t_0 – время, затраченное корневым процессом на предварительную декомпозицию исходной задачи и на распределение нагрузки между подчиненными процессами, а t_i – время работы i-го подчиненного процесса.

3.3. Программная модель для решения задач класса C

Предлагается модель однородной параллельной программы для решения задач класса C с одним приемопередающим буфером обмена в каждом процессе.

```
      /* Общий блок */
{MPI_Init (&argc, &argv);                          // Инициализация MPI
 MPI_Status status;                                // Декларация объекта status
 int i, j, p, NumProcs, myid, BufMaxSize, size, count, tag;   // Декларация общих объектов
 CBV Pact (0, NumProcs, 1);          // Вектор Pact, «единицы» которого отмечают активные процессы
 MPI_Comm_size (MPI_COMM_WORLD,&NumProcs);          // Определение числа процессов
 MPI_Comm_rank (MPI_COMM_WORLD, &myid);     // Определение идентификатора процесса

      /* Корневой блок */
 if (!myid)
 { /* Корневой процесс вводит исходные данные и по возможности определяет значение BufMaxSize емкости буфера Buf. */}

      /* Общий блок */
/* Если значение BufMaxSize определено, то выполняется пара следующих операций. */
 MPI_Bcast (&BufMaxSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
 CBUF Buf (BufMaxSize);                  // Образование буфера Buf в каждом из процессов
/* Если значение BufMaxSize не было определено, то оно задается по умолчанию определенной константой MAX_SIZE_BUF.*/
 CBUF Buf (BufMaxSize = MAX_SIZE_BUF);

      /* Корневой блок */
 if (!myid)
 { /* Корневой процесс декомпозирует исходную задачу на p ≤ NumProcs частных задач и рассылает их подчиненным процессам. При этом NumProcs – p процессам посылаются «пустые» (длиной 0 байт) сообщения с tag = 0. */
  for (i = 0; i < NumProcs; i ++)
  {tag = size = 0;
   if (i <= p)
```

```
  { /* Объекты, представляющие i-ю частную задачу, упаковываются в буфер Buf, при этом вычисляется общая
длина size подготовленного сообщения в байтах, tag = 1, и i-й компоненте Pact присваивается значение 1.*/ }
    MPI_Send (Buf.m_nBuf, size, MPI_BYTE, i, tag, MPI_COMM_WORLD);
   }
   if (!Pact.IsZero ())                    // Прием сообщения от процесса, нашедшего решение первым
   {MPI_Recv (Buf.m_nBuf, BufMaxSize, MPI_BYTE, MPI_ANY_SOURCE, 2,
                    MPI_COMM_WORLD, &status);
    i = status.MPI_SOURCE;                          // Определение процесса-источника сообщения
    MPI_Get_count (&status, MPI_BYTE, &count);
    Buf.SetSize (count);
/* По принятому сообщению формируется и выдается решение исходной задачи. */
   }
   j = 0;
   while ((j = Pact.LeftOne (j)) ! = -1)
/* Посылается сигнал конца работы (пустое сообщение с tag = 0) всем активным процессам.*/
   MPI_Send (Buf.m_nBuf, 0, MPI_BYTE, j, 0, MPI_COMM_WORLD);
   Pact.SetBitAt (i, 0);
   while (!Pact.IsZero ())
/* Принимаются сообщения от всех процессов, отмеченных в Pact «единицами». */
   {MPI_Recv (Buf.m_nBuf, BufMaxSize, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &status);
   i = status.MPI_SOURCE;
   Pact.SetBitAt (i, 0);
  }
 }

        /* Блок подчиненных процессов */
 else
 while (1)
 {MPI_Recv (Buf.m_nBuf, BufMaxSize, MPI_BYTE, 0, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status);
  if (!status.MPI_TAG) break;
/* Если данному процессу было послано сообщение с tag = 0, то он прекращает свою работу, а иначе
выполняет следующие операции */
  MPI_Get_count (&status, MPI_BYTE, &count);
  Buf.SetSize (count);
/* По принятому от корневого процесса сообщению в буфере Buf формируются объекты, представляю-
щие исходные данные для частной задачи данного процесса, и инициируется выполнение модуля, ре-
шающего эту задачу. Результаты решения упаковываются в буфер Buf, вычисляется длина сообщения
size, которое посылается корневому процессу с tag = 2. /*
  MPI_Send (Buf.m_nBuf, size, MPI_BYTE, 0, 2, MPI_COMM_WORLD);
 }

        /* Общий блок */
 MPI_Finalize ();
}
```

Поскольку время t_p решения исходной задачи класса C определяется по часам i-го процесса, быстрее всех нашедшего это решение, то

$$t_p = t_0 + min\ \{t_i\}\ \ (i \in \{1, 3, \ldots, N-1\}),$$

где t_0 – время, затраченное корневым процессом на предварительную декомпозицию исходной задачи и на распределение нагрузки между подчиненными процессами, а t_i – время работы i-го подчиненного процесса.

Заключение

Предлагаемые здесь программные модели могут оказаться полезными в качестве шаблонов при разработке параллельных программ начинающими разработчиками, владеющими языком C++, но досконально не знакомыми с библиотекой MPI. Обе модели были опробованы в качестве шаблонов при разработке программы параллельной проверки троичной матрицы на вырожденность [5]. Решаемая при этом задача, в зависимости от типа проверяемой исходной матрицы, является представителем того или иного класса рассматриваемых здесь задач.

При доказательстве вырожденности матрицы решается задача класса *J*, время решения которой определяется «самым медленным» процессом, доказавшим, что последний из непроверенных еще миноров также оказался вырожденным. При обнаружении вектора, ортогонального всем строкам матрицы, решается задача из класса *C*, время решения которой определяется «самым быстрым» процессом, первым обнаружившим такой вектор.

Проведенные эксперименты показали, что *k*-кратное ускорение параллельной проверки троичной матрицы на вырожденность с использованием *k* подчиненных процессов, по сравнению с однопроцессорным вариантом, вполне достижимо при надлежащем распределении нагрузки между процессами.

Эффективный информационный обмен между процессами при параллельных логико-комбинаторных вычислениях поможет организовать специально разработанный класс **CBUF** операций над сообщениями с логическими объектами.

Список литературы

1. Принципы построения суперкомпьютеров семейства «СКИФ» и их реализация / С.М. Абрамов, Н.Н. Парамонов, В.В. Анищенко, С.В. Абламейко // Информатика. – 2004. – № 1. – С. 89–106.

2. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. – Мн.: БГУ, 2002. – 323 с.

3. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message Passing Interface // MIT Press. – 1995.

4. Романов В.И., Василькова И.В. Булевы векторы и матрицы в C++ // Логическое проектирование. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 1997. – С. 150–158.

5. Черемисинов Д.И., Черемисинова Л.Д. Троичные векторы и матрицы // Логическое проектирование. – Мн.: Ин-т техн. кибернетики НАН Беларуси, 1998. – С. 146–155.

6. Торопов Н.Р. Параллельная проверка ДНФ на тавтологию // Информатика. – 2005. – № 2. – С. 35–42.

Поступила 21.04.05

Объединенный институт проблем информатики НАН Беларуси, Минск, Сурганова, 6

N.R. Toropov

PARALLEL LOGIC-COMBINATORIAL CALCULATIONS WITH MPI

The calculation problems in multi-processor systems are divided into two typical classes. For each class the program model for executing the parallel calculations on family SKIF supercomputer with MPI and a basic language C++ is suggested. A special class of communication operations (compact packing of logical objects in communications to be sent and their unpacking from received communications) for organizing the effective information exchange between parallel processors is worked out.