Эффективность динамического децентрализованного управления для параллельных алгоритмов обращения матриц

Е.А. Ильченко

Тамбовский государственный университет

В работе обсуждаются некоторые подходы к распараллеливанию рекурсивных древовидных алгебраических алгоритмов. Для таких алгоритмов рассматривается несколько схем управления параллельным вычислительным процессом, реализующих концепцию динамического децентрализованного распараллеливания алгоритмов с использованием MPI. Сравнивается эффективность таких схем. Приводятся результаты экспериментов по обращению и вычислению ступенчатой формы матриц на кластере МВС-10П. Дается сравнительный анализ экспериментов для матриц с рациональными коэффициентами и для матриц в конечных полях.

Ключевые слова: динамическое децентрализованное управление, DDP, adjoint, МВС-10П, MathPartner, MPI, многопоточность, POSIX treads, C++, GMP.

1. Введение

Технологии написания параллельных программ, реализующих точные символьные алгоритмы, активно развиваются [1–4]. Одной из актуальных проблем является написание эффективной реализации блочно-рекурсивных алгебраических алгоритмов, работающих с неоднородными данными.

Одним из решений проблемы неоднородности данных служит стратегия, использующая диспетчера, выполняющего функции динамической загрузки вычислительных узлов супер-ЭВМ. В данной статье развиваются идеи, предложенные в работе [5], когда на каждом задействованном вычислительном узле создается свой диспетчерский поток.

Рассматриваемая парадигма носит название *DDP – dynamic decentralized parallelization.* Современные кластерные системы, построенные на линейках процессоров Intel Xeon или AMD Opteron, предполагают высокий уровень многоядерности. Поэтому оказывается выгодным запускать на каждом вычислительном узле кластера, имеющем m ядер, единственный MPI-процесс, порождающий один диспетчерский и $m-1$ счетных потоков, которые будут использовать общее адресное пространство.

Работа алгоритма начинается с того, что некоторый из MPI-процессов имеет у себя стартовую задачу, а номера остальных MPI-процессов находятся у него в специальном списке. В процессе счета начальной задачи будут создаваться подзадачи, которые будут по возможности передаваться другим MPI-процессам. Если некоторый узел создает дочерние, впоследствии он может отсылать им равное количество номеров еще не занятых вычислениями MPI-процессов. Такое поведение одинаково для всех.

Пример организации параллельных вычислений при использовании DDP-схемы показан на рисунке 1. Пусть для работы алгоритма было создано 10 MPI-процессов. Вершины деревьев – это диспетчеры MPI-процессов, они пронумерованы. Связи – это отношения родитель–потомок. Числа в прямоугольниках показывают, какими номерами свободных MPI-процессов располагает процесс на данный момент. Пусть было запущено 10 экземпляров параллельной программы с номерами $0\ldots9$. На рисунке 1(а) показан начальный момент времени. Задача запускается на процессе с номером 0, который располагает номерами $1\ldots9$. При вычислении имеющейся задачи создается несколько подзадач, часть из которых была оставлена на процессе с номером 0, а две других были переданы процессам с номерами 1

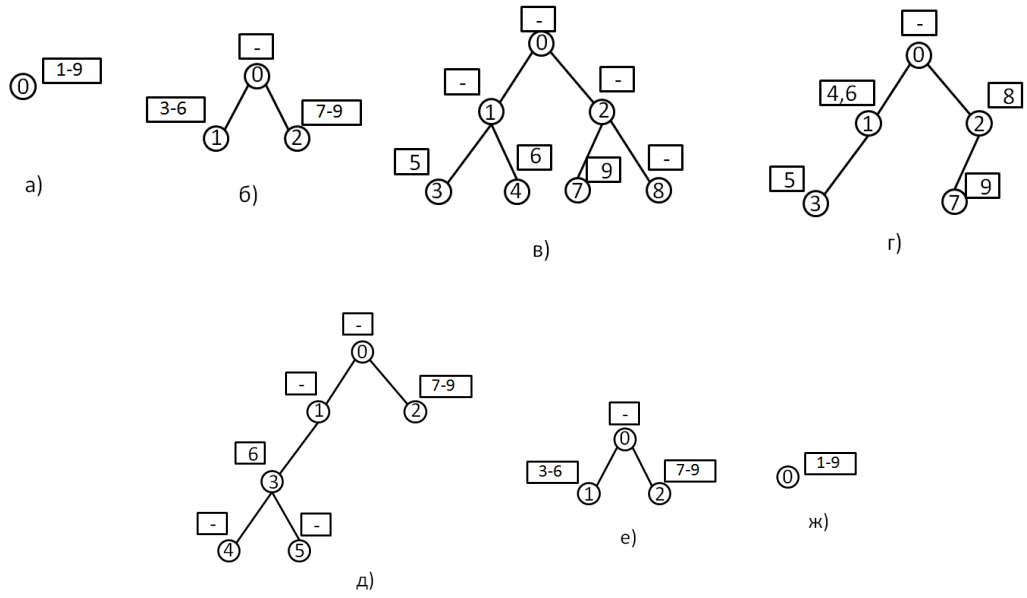


Рис. 1. *Пример организации параллельных вычислений при использовании DDP-схемы.*

и 2, также им были отданы все имеющееся номера свободных процессов. Диспетчеры 1 и 2 также делят свои задачи и часть из них пересылают другим диспетчерскам (рис.1(в)). На следующем рисунке процессы 6 и 8 посчитали свои задачи, вернули результат своим родителям. К списку свободных узлов процесса 1 добавились 4 и 6, к списку процесса 2 —- число 8. На рисунке 1(д) диспетчерский поток 3 получил от родителя новые номера свободных MPI-процессов, одну из подзадач передал уже имевшемуся дочернему процессу 5, другую передал только что полученному четвертому узлу. На рисунке 1(е,ж) показана завершающая стадия работы алгоритма.

2. Описание реализации DDP

2.1. Четыре основных режима работы диспетчерского потока.

Работа диспетчерского потока реализована в виде цикла, на каждой итерации которого в зависимости от состояния вычисления имеющейся на узле задачи принимаются те или иные действия. Диспетчер имеет четыре основных состояния (режима работы):

- `MODE_TASK_WAIT` — ожидание задачи. Изначально в этом режиме находятся диспетчерские потоки всех узлов, кроме корневого узла (корневым считается узел с номером 0, если для работы DDP был выделен глобальный коммуникатор `MPI_COMM_WORLD`, либо берется первый узел из списка, который был передан в конструктор класса `DispThread`).
- `MODE_RECEIVING_TASK_DATA` — десериализация исходных данных полученной задачи. Этот режим является промежуточным между ожиданием задачи и ее непосредственной обработкой.
- `MODE_PROC_TASK` — вычисление полученной задачи. Задача считается посчитанной сразу же в момент получения результата, до завершения постобработки.
- `MODE_FINISH_CUR_TASK` — завершение обработки имеющейся задачи. В этом режиме диспетчерский поток сообщает родительскому узлу о завершении работы и отсылает ему все имеющиеся локальные номера свободных процессоров.

2.2. Принцип отложенной постобработки.

Любая из вершин некоторого графа алгоритма имеет четыре процедуры обработки: инициализацию, счет, сборку (если вершина считалась параллельно и разбивалась на подзадачи) и постобработку (может присутствовать не у всех вершин). DDP использует *принцип отложенной постобработки*. Он заключается в том, что если MPI-процесс закончил обработку вершины, для которой нужна постобработка, но данные для нее еще не были получены, то эта вершина отправляется в специальный контейнер для ожидания постданных. После перемещения вершины в контейнер ожидания постобработки MPI-процесс сообщает родителю (MPI-процессу, приславшему эту задачу) что работа закончена, возвращает ему весь набор свободных процессоров (включая себя). Как только дочерний узел возвращает список номеров свободных MPI-процессов, он может получить новую задачу. Нужно заметить, что вершин, ожидающих своей постобработки может быть несколько на одном MPI-процессе, более того, у всех этих вершин могут быть различные родительские MPI-процессы. Таким образом обеспечивается динамическая загрузка узлов вычислительной системы.

2.3. Реализация диспетчерского потока.

В процессе работы диспетчерского потока происходит обмен сообщениями с диспетчерами других узлов вычислительного кластера. Все эти сообщения можно разделить на две группы.

1. Сообщения с данными для обработки задач — исходные данные задачи, данные для постобработки и результат задачи. Размер этих сообщений может достигать нескольких ГБ, поэтому одно сообщение может быть разбито на несколько частей, что приводит к необходимости использования некоторого протокола обмена сообщениями данного типа. Для отправки сегментов сообщений используется процедура `MPI_Isend`, для соответствующего приема используется `MPI_Irecv`. В обоих случаях выбраны неблокирующие процедуры, поскольку прием или посылка данных большого объема может потребовать значительное время, что может привести к «зависанию» цикла диспетчерского потока на одном шаге.

2. Служебные сообщения — номера свободных процессоров, сигналы о завершении работы и вспомогательные коммуникации для осуществления пересылки сообщений из первой группы. Размер служебных сообщений невелик (в худшем случае несколько КБ), для пересылки используется процедура `MPI_Isend` (выбран неблокирующий вариант, чтобы предотвратить возникновение тупиковой ситуации и исключить простаивание локального диспетчерского потока в ожидании инициализации соответствующей процедуры приема). Для приема служебных сообщений используется блокирующая процедура `MPI_Recv` (перед любым непосредственным приемом служебного сообщения выполняется проверка на его наличие при помощи процедуры `MPI_Iprobe`, что исключает «зависание» диспетчерского потока на одном шаге).

Рассмотрим протокол обмена сообщениями первой группы, содержащими данные для счета. Пусть на каждом узле кластера, задействованном для работы DDP, имеется счетчик отправленных сообщений (с учетом сегментации) из первой группы (`int sendingIdentsCounter`). Тогда любая задача, отданная на счет другому узлу, может быть идентифицирована парой целых чисел — номером узла, с которого она была отправлена и значением счетчика `sendingIdentsCounter` на этом узле в момент отправки сообщения. Поскольку сообщение из первой группы может быть разбито на несколько частей, перед непосредственной отправкой сегментов получателю отслылается служебное сообщение, содержащее дескриптор отправляемых данных. Дескриптор включает в себя:

- тип сообщения (`TYPE_TASK_DATA`, `TYPE_TASK_POSTPROC_DATA` или `TYPE_TASK_RESULT`);
- размер сообщения (число байт);
- значение локального счетчика `sendingIdentsCounter` на момент отправки этого сообщения;
- некоторые дополнительные данные `ExtraData*`, специфичные для каждого из трех типов сообщений. Для исходных данных задачи поле `ExtraData*` будет содержать тип графа родительской вершины, номер отсылаемой задачи в этом графе, уровень рекурсии этой задачи (задача, с которой начинался счет имела на корневом узле кластера имела уровень рекурсии 0, а каждая создаваемая подзадача имеет уровень рекурсии на единицу больший, чем родительская) и флаг (`true/false`), сигнализирующий о том, что для отсылаемой задачи необходимо использовать многопоточный алгоритм, работающий над общей памятью (кустовая вершина). Для сообщений с постданными задачи или посчитанным результатом поле `ExtraData*` будет содержать единственное целое число – значение счетчика `sendingIdentsCounter`, которое использовалось для отправки данных задачи (таким образом, с помощью поля `ExtraData*` выполняется привязка постданных или результата к исходных данным задачи).

После отправки дескриптора и сериализации необходимых данных выполняется отправка большого сообщения. Пусть сообщение было разделено на n сегментов, тогда для их отправки будут использованы теги $[sendingIdentsCounter + 0, sendingIdentsCounter + 1, ..., sendingIdentsCounter + n - 1]$, после чего счетчик `sendingIdentsCounter` увеличивается на n. Поскольку перед отправкой сегментов выполняется пересылка служебного дескриптора для этого сообщения, получатель однозначно идентифицирует принимаемые данные. Десериализация принятых данных выполняется только после получения всех сегментов сообщения.

Работу цикла диспетчерского потока удобно представлять в последовательном выполнении некоторого набора процедур (шагов), которые можно разбить на две группы: служебные процедуры, которые не задействуют счетные потоки, и процедуры, при выполнении которых некоторые из счетных потоков получают задания (псевдокод показан на рисунках 2,3). Рассмотрим список процедур из первой группы:

- выполнить прием номеров свободных узлов от родительского узла (шаг 1);
- выполнить прием номеров свободных узлов от дочерних узлов(шаг 2);
- выполнить прием всех сообщений с данными для постобработки (шаг 3);
- выполнить проверку завершенности приема сообщений с постданными (процедура `MPI_Test` для каждого сегмента, шаг 4);
- вернуть результаты посчитанных задач (с выполненной постобработкой) отправителям (выполнить процедуру `MPI_Isend` для каждого сегмента данных, шаг 5);
- проверить наличие сообщений с данными большого объема (выполнить процедуру `MPI_Irecv` для каждого сегмента, шаг 9);
- проверить наличие сообщения с дескриптором исходных данных задачи (шаг 10);
- проверить завершенность приема всех сегментов исходных данных задачи (процедура `MPI_Test` для каждого сегмента, шаг 11);
- проверить наличие сообщения с дескриптором результата задачи(шаг 13);

```
//пока значение флага, сигнализирующего о выходе из главного цикла,
//не было изменено на true
while (!flForExitFromMainLoop){
   //прием номеров свободных MPI-процессов и прием постданных
   •выполняем процедуры 1-4;
   if (mode!=MODE_FINISH_CUR_TASK{
      //необходимо зарезервировать часть номеров свободных процессоров,
      //чтобы в дальнейшем была возможность создать из них новые дочерние узлы.
      //В противном случае они могут быть все отданы уже имеющимся дочерним узлам.
      •резервируем часть номеров свободных процессоров;
   }
   •выбираем все имеющиеся незанятые счетные потоки;
   •выполняем действия 5-9;
   //если текущий режим - ожидание задачи для счета
   if(mode==MODE_TASK_WAIT{
      //проверим, пришла ли задача (шаг 10)
      int checkRes=checkIncomingTask(&parent, externTasksToSendingIdentAndNode);
      //если был получен сигнал к завершению работы узла
      if(checkRes==INC_TASK_COMMAND_TO_FINISH){
         //изменим флаг выхода из главного цикла
         flForExitFromMainLoop=true;
      }
      else{
         //если на узел пришла задача
         if (checkRes==INC_TASK_RECV){
            //отметим, что данные внешней задачи еще не были десериализованы
            isStartTaskDeserializationStart=false;
            //переведем главный цикл в режим получения данных задачи
            mode=MODE_RECEIVING_TASK_DATA;
         }
      }
   }
   if (mode==MODE_RECEIVING_TASK_DATA){
      //проверим завершенность приема данных задачи
      •выполняем шаг 11;
   }
```

Рис. 2. *Псевдокод главного цикла диспетчерского потока (часть 1).*

- проверить завершенность приема всех сегментов результата посчитанной задачи (процедура `MPI_Test` для каждого сегмента, шаг 14);
- отправить сериализованные постданные задач получателям (выполнить процедуру `MPI_Isend` для каждого сегмента данных, шаг 15);
- отправить исходные данные задач получателям, тем самым создав новые дочерние узлы (выполнить процедуру `MPI_Isend` для каждого сегмента данных, шаг 22);
- выполнить отправку имеющихся номеров свободных процессов дочерним узлам(шаг 25);
- проверить наличие сообщений от дочерних узлов с запросом на завершение (при наличии таких сообщения отправителю возвращается номер последнего свободного процесса, который был ему отправлен, что позволяет дочернему узлу корректно завершить свою работу, шаг 26);
- проверить наличие сообщения с номером последнего свободного MPI-процесса, отправленного этому дочернему узлу (используется для корректного завершения работы текущего дочернего узла, шаг 27);

```
   if (mode==MODE_PROC_TASK && !flForExitFromMainLoop){
      •выполняем шаги 12-19;
      //если внешняя задача посчитана и мы находимся на корневом узле
      //(исходная задача, с которой начинается выполнение алгоритма,
      //также помечается как внешняя на корневом узле
      if (isExternTaskProcessed() && isRootRank()){
         //подождем сообщения с номерами свободных процессоров от дочерних узлов
         //(они могут прийти позже, чем результат задачи)
         waitDaughterNodes(lastFreeNode,freeNodes);
         //отправим сообщение о завершении работы всем узлам
         sendMessageAboutExitToAllNodes();
         //завершим работу главного диспетчерского цикла
         flForExitFromMainLoop=true;
      }
      //если работа диспетчерского цикла не была завершена в теле предыдущего условия
      if (!flForExitFromMainLoop){
         //если внешняя задача посчитана и текущий узел не является корневым
         if (isExternTaskProcessed() && !isRootRank()){
            //сбросим флаг сигнализирующий о том, что внешняя задача посчитана
            isExternTaskProcessedFl = false;
            //подождем сообщения с номерами свободных процессоров от дочерних узлов
            waitDaughterNodes(lastFreeNode, freeNodes);
            //сообщим родительской вершине о завершении работы
            tellParentAboutFinish(parent);
            •переместим все номера зарезервированных узлов в массив свободных;
            //перейдем в режим завершения работы узла
            mode = MODE_FINISH_CUR_TASK;
         }
         //если работа диспетчерского цикла не была завершена в теле предыдущего условия
         if (mode != MODE_FINISH_CUR_TASK){
            •выполняем действия 20-26;
         }
      }
   }
   //если не было получен сигнал к завершению работы цикла
   if (!flForExitFromMainLoop){
      //если цикл перешел в режим завершения работы узла
      if (mode == MODE_FINISH_CUR_TASK){
         //проверим, пришло ли сообщение с номером последней отправленной
         //вершины от родительского узла
         •выполним шаг 27;
         //если полученный номер последнего отправленного MPI-процесса
         //присутствует в локальном списке
         if (tryFinishCurrentJob(parent, freeNodes)){
            //если процедура завершения работы узла выполнена успешно,
            //переводим главный цикл диспетчерского потока в режим ожидания задачи
            mode = MODE_TASK_WAIT;
         }
      }
      //проверим завершенность неблокирующих посылок
      •выполним шаг 28;
   }
}
```

Рис. 3. *Псевдокод главного цикла диспетчерского потока (часть 2).*

- проверить завершенность выполнения всех вызванных неблокирующих процедур `MPI_Isend` (вызвать процедуру `MPI_Test` для каждого объекта `MPI_Request`, возвращенного неблокирующими посылками; в случае успешного завершения выполняется очистка задействованных данных, шаг 28).

Ко второму набору процедур (шагов), выполняемых диспетчером в процессе своей работы, относятся такие, при завершении которых счетные потоки могут получить некоторые задания:

- сериализовать результаты посчитанных задач (если для вершины требуется постобработка, то на момент сериализации она должна быть выполнена, шаг 6);
- выполнить постобработку вершин (шаг 7);
- сделать десериализацию постданных вершин (шаг 8);
- проверить наличие кустовой вершины и при необходимости выполнить ее счет в многопоточном режиме (шаг 12);
- сериализовать доступные постданные задач, которые были отправлены другим MPI-процессам (шаг 16);
- выполнить десереализацию данных задачи, принятой MPI-процессом (выполняется для нового дочернего узла, шаг 17);
- десериализовать принятые результаты счета от дочерних узлов (шаг 18);
- выполнить сборку графов задач (шаг 19);
- сериализовать данные задач (для создания новых дочерних узлов, шаг 20);
- выполнить инициализацию данных вершин (шаг 21);
- выполнить счет «легких» вершин (шаг 23);

2.4. Реализация счетных потоков

Всего счетный поток может принимать 12 заданий, часть из которых выполняют подготовку данных к отправке либо их восстановление после приема, а другая часть выполняет обработку вершин графов задач (псевдокод цикла обработки заданий счетным потоком показан на рисунке 4).

1. Выполнить сериализацию данных задачи.
2. Выполнить сериализацию данных для постобработки некоторой тяжелой вершины.
3. Выполнить сериализацию результата задачи.
4. Выполнить десериализацию данных задачи.
5. Выполнить десериализацию данных для постобработки некоторой тяжелой вершины.
6. Выполнить десериализацию результата задачи.
7. Выполнить инициализацию некоторой тяжелой вершины графа задачи.
8. Выполнить счет некоторой тяжелой вершины графа задачи.
9. Выполнить счет некоторой легкой вершины графа задачи.
10. Выполнить постобработку некоторой тяжелой вершины графа задачи.
11. Выполнить завершающую сборку некоторого графа задачи.
12. Выполнить счет кустовой задачи (в многопоточном режиме).

```
//пока не было получено сообщение от диспетчера, что работу потока необходимо прервать
while (!isThreadStopped()){
   //если получено задание сериализовать данные некоторой задачи
   if (isRecvSerializationTaskDataJob()){
      //выполняем действия по сериализации данных задачи
      •••
      //помещаем задачу в соответствующий контейнер с задачами,
      //данные которых сериализованы
      gCont→addSerializedTaskForSending(taskDescriptor);
      //ставим метку, что задание по сериализации данных отсутствует
      setAbsentSerializationTaskDataJob();
      //добавляем текущий поток в массив свободных потоков диспетчера
      disp→addFreeThread(this);
   }
   else{
      //если получено задание сериализовать постданные некоторой вершины
      if(isRecvSerializationPostProcDataJob()){
         //выполняем действия по сериализации постданных вершины
         •••
         //помещаем задачу в соответствующий контейнер с задачами,
         //постданные которых сериализованы
         gCont→addSerializedTaskPPDataForSending(taskDescriptor);
         //ставим метку, что задание по сериализации постданных отсутствует
         setAbsentSerializationPostProcDataJob();
         //добавляем текущий поток в массив свободных потоков диспетчера
         disp→addFreeThread(this);
      }
      else{
         //подобным образом обрабатываем остальные 10 заданий,
         //выполняемых счетным потоком
         •••
      }
   }

}
```

Рис. 4. *Псевдокод главного цикла счетных потоков.*

3. Объединение статического управления и DDP на примере алгоритма модулярного обращения матриц

Пусть для запуска параллельной программы выделено N узлов супер-ЭВМ, m — количество ядер на каждом узле. Во время работы программмы допускается поочередный запуск нескольких параллельных алгоритмов, реализованных с помощью фреймворка DDP. Кроме того, можно выполнить разбиение N узлов на несколько подмножеств N_i' и произвести параллельный запуск DDP на каждом из этих подмножеств. Это дает возможность эффективно применять DDP для реализации параллельных алгоритмов, работающих в кольцах простых модулей.

Для точных алгоритмов компьтерной алгебры зачастую известна только верхняя граница необходимого количества простых модулей для восстановления по Китайской теореме об остатках [13], но точное их количество определяется только в процессе работы программы. Хорошим примером является блочный рекурсивный алгоритм нахождения расширенной присоединенной матрицы [10–12] (параллельная реализация с использованием фреймворка DDP показана в работе [5]).

Пусть M — исходная матрица, K — количество простых модулей, произведение которых не меньше чем $2det(M) + 1$ (верхняя оценка количества необходимых простых моду-

```
//пока значение флага, сигнализирующего о том, что после восстановления ответ был
//изменен по сравнению с предыдущим, не было изменено на false
while (isAnswerChangedFl){
   //проверяем первый случай
   if (k<5){
       //выполняем процедуру счета для первого случая
       runSingleDDPForAllNodes(...);
       k=k-1;
   }
   else{
       if (k<N){
           //выполняем процедуру счета для второго случая,
           //процедура возвращает найденное значение k' (см. описание)
           k=k-runMultipleDDPForAllNodes(...);
       }
       else{
           if (k<Nm){
               //выполняем процедуру счета для третьего случая
               runMTAdjointForEveryNode(...);
               k=k-N;
           }
           else{
               //выполняем процедуру счета для третьего случая
               runSTAdjointForEveryCore(...);
               k=k-Nm;
           }
       }
   }
   //выполним процедуру перераспределения строк полученных матриц между
   //всеми узлами вычислительного кластера
   allToAllVZp32Set(...);
   //выполним процедуру продолжнения восстановления для новых
   //полученных данных, процедура вернет true, если хотя бы один элемент
   //локально восстанавливаемого диапазона строк матрицы-результата был изменен
   bool localChangedFl=continueRecovingMatrixSByNewtonMT(...);
   //сообщим узлу с номером 0 информацию о локальных изменениях,
   //который примет решение о продолжении работы алгоритма и вернет
   //всем MPI-процессам true, если хотя бы на одном произошло
   //изменение ответа после восстановления
   if (myRank!=0){
       sendChangesInfoToRootRank(localChangedFl);
       isAnswerChangedFl=recvFinishAlgoInfoFromRootRank();
   }
}
```

Рис. 5. *Псевдокод алгоритма пошагового выбора модулей для счета в кольцах Zp32.*

лей для восстановления присоединенной матрицы определяется неравенством Адамара). Вычисления будут производиться в несколько этапов, на каждом шаге будет выбираться некоторый диапазон модулей. После каждого шага продолжается восстановление ответа относительно предыдущего, используя новые данные (на первом шаге продолжение восстановления начинается с нулевой матрицы), с помощью Схемы Ньютона. Разобьем множество целых неотрицательных чисел на четыре подмножества: $Z_A = [0 \dots 5), Z_B = [5 \dots N), Z_C = [N \dots N \cdot m), Z_D = [N \cdot m \dots + \infty)$. Обозначим через q количество модулей, по которым уже был произведен счет и последующее восстановление после некоторого шага. Пусть $k = K - q$, тогда возможны четыре варианта выбора количества модулей для текущего шага.

1. Наиболее простым является случай, когда для текущего шага берется единственный модуль, и на всех N узлах вычислительного кластера запускается DDP для счета по

этому модулю. Такая ситуация возникает, когда $k < 5$.

2. Если $k \notin Z_A$, тогда следующим проверяется тот факт, что для оставшегося количества модулей для восстановления выполняется неравенство: $k < N$. Найдем такое $\sqrt{k} \leq k' \leq \frac{4}{3}\sqrt{k}$, чтобы при этом k' делило нацело N. Если такого k' найти не удалось, выбирается то, которое даст наименьший остаток при делении числа N. Пусть $d = \lfloor N/k' \rfloor, r = N \ mod \ k'$. Значит будет произведено k' запусков DDP с порядковыми номерами $D_i = [0..k'-1]$, каждое из которых задействует $d + x$ узлов, где

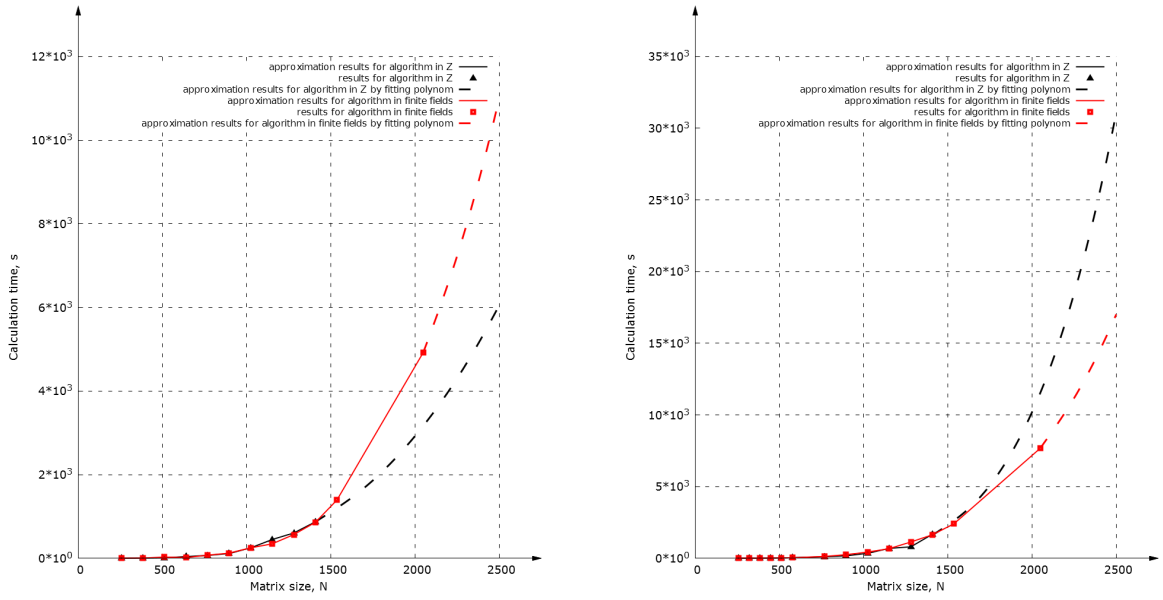
$$x = \begin{cases} 1, & \text{если } i < r \\ 0, & \text{если } i \geq r. \end{cases}$$

3. Следующим шагом выполняется проверка принадлежности k множеству Z_C, когда $N \leq k < Nm$. В этом случае на каждом из N узлов будет запущен свой экземпляр DDP, таким образом производится параллельный счет по N различным модулям.

4. Последним проверяется неравенсвто $k \geq Nm$. В этом случае происходит одновременный запуск Nm однопоточных реализаций алгоритма нахождения присоединенной матрицы, по одной для каждого ядра.

Независимо от выбранного количества модулей для текущей итерации, после завершения счета по этому набору модулей присходит сравнение нового полученного ответа с предыдущим. Если ответы совпадают – значит с большой долей вероятности был получен верный ответ, и алгоритм закончил свою работу.

4. Анализ результатов вычислительных экспериментов на кластере МВС-10П



а) б)

Рис. 6. *Зависимость времени работы программы от размера исходной матрицы. Коэффициенами матриц являются 15-битные числа. Плотности исходных матриц составляют 1%(а) и 100%(б).*

Для исследования алгоритмов прямого (в множестве целых чисел $\mathbb{Z}$) и модулярного (в конечных полях) вычислений расширенной присоединенной матрицы была написана

программа на языке C++. Для реализации многопоточности использовалась библиотека PThreads, арифметика многократной точности реализована с использованием GMP. Для всех экспериментов использовалось 20 узлов кластера, по 8 задействованных ядер на каждом.

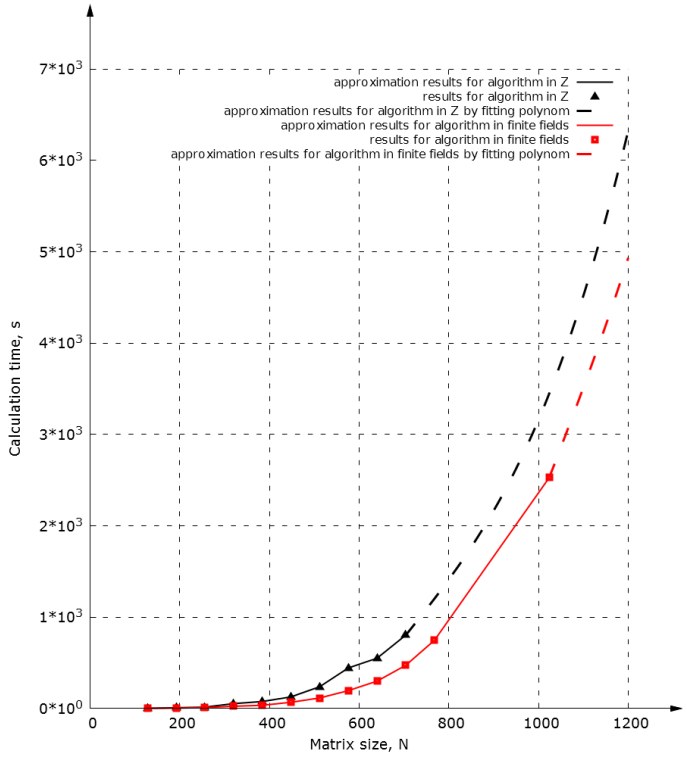


Рис. 7. *Зависимость времени работы программы от размера исходной матрицы. Коэффициенами матриц являются 100-битные числа, плотность матриц 100%.*

На рисунке 6(a) показано сравнение времени работы прямого и модулярного алгоритма для разреженных исходных матриц плотностью 1%, коэффициенты матриц занимали не более 15 бит в двоичном представлении. Для каждой серии экспериментов была выполнена аппроксимация результатов приближающим полиномом в системе компьютерной алгебры MathPartner [14], результаты аппроксимации показаны пунктиром. Эксперименты для матриц с 1% плотности и 15-битными коэффициентами дают следующие приближающие полиномы для оценки времени работы программы:

$$FitZ^a = 4.52 * 10^{-7} n^3 - 1.02 * 10^{-4} n^2 - 0.16n + 46, \quad \mathrm{Err}(Z^a) = 8.5\%;$$

$$FitZp^a = 1.596 * 10^{-6} n^3 - 2.886 * 10^{-3} n^2 + 1.768n - 308, \quad \mathrm{Err}(Zp^a) = 8.4\%.$$

В правой части формул показана среднеквадратичная ошибка, которая находилась по формуле:

$$\mathrm{Err} = 100 \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\frac{A_i^E - A_i^P}{A_i^E} \right)^2},$$

где n – количество точек с результатами экспериментов, A_i^E – экспериментальное время работы программы для текущей точки, A_i^P – оценочное время работы программы, которое получено с помощью приближающего полинома.Формулы для оценки времени работы программы можно записать более наглядным образом, если измерять размер входной матрицы

в тысячах ($n = 1000N$):

$$FitZ_N^a = 452N^3 - 102N^2 - 164N + 46,$$
$$FitZp_N^a = 1596N^3 - 2886N^2 + 1768N - 308.$$

Эксперименты с матрицами 100% плотности и элементами, имеющими не более 15 бит в двоичном представлении (рисунок 6(б)) дают более высокую степень приближающих полиномов для обеих версий программы:

$$FitZ^b = 1.837 * 10^{-9}n^4 - 3.514 * 10^{-6}n^3 + 2.487 * 10^{-3}n^2 - 0.519n,$$
$$FitZp^b = 3.76 * 10^{-10}n^4 + 2.92 * 10^{-7}n^3 - 4.13 * 10^{-4}n^2 + 0.131n,$$
$$\mathrm{Err}(Z^b) = 11.9\%, \quad \mathrm{Err}(Zp^b) = 5.7, \%,$$
$$FitZ_N^b = 1837N^4 - 3514N^3 + 2487N^2 - 519N,$$
$$FitZp_N^b = 376N^4 + 292N^3 - 413N^2 + 131N.$$

Если зафиксировать плотность матриц на 100%, но увеличить размер входных коэффициентов до 100 бит, приближающие полиномы также имеют четвертую степень, но другие коэффициенты (рисунок 7):

$$FitZ^c = 2.439 * 10^{-9}n^4 + 1.083 * 10^{-6}n^3 - 4.36 * 10^{-4}n^2 + 6.26 * 10^{-2}n,$$
$$FitZp^c = 1.078 * 10^{-9}n^4 + 3.425 * 10^{-6}n^3 - 2.819 * 10^{-3}n^2 + 0.61n,$$
$$\mathrm{Err}(Z^c) = 12.3\%, \quad \mathrm{Err}(Zp^c) = 7.1\%,$$
$$FitZ_N^c = 2439N^4 + 1083N^3 - 436N^2 + 63N,$$
$$FitZp_N^c = 1078N^4 + 3425N^3 - 2819N^2 + 606N.$$

Проведенные серии экспериментов показывают, что программа для нахождения расширенной присоединенной матрицы, полностью работающая в целых числах $\mathbb{Z}$, может давать выигрыш для разреженных матриц с небольшими начальными коэффициентами, но с увеличением плотности оказывается выгодным использование алгоритмов, работающих в конечных полях, что согласуется с теорией.

Литература

1. Jean-Guillaume Dumasa, Thierry Gautierb, Clement Pernet, Jean-Louis Rochb, Ziad Sultana. Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination // Parallel Computing, Volume 57, September 2016, Pages 235-249.

2. A. R. Benson and G. Ballard. A framework for practical parallel fast matrix multiplication // In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 42-53, New York, NY, USA, 2015. ACM.

3. P. D'alberto, M. Bodrato, and A. Nicolau. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation // ACM Trans. Math. Softw., 38(1):2:1-2:30, Dec. 2011.

4. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages // ACM Trans. on Mathematical Software(TOMS), 35(3):1-42, 2008.

5. Ильченко Е.А. Об эффективном методе распараллеливания блочных рекурсивных алгоритмов // Вестник Тамбовского университета. Серия: Естественные и технические науки. Выпуск 5, том 20, 2015.

6. Ильченко Е.А. Инструменты математического сервиса «MathPartner» для выполнения параллельных вычислений на кластере // Труды ИСП РАН, 2016, том 1, вып. 3, с. 173-188.

7. Ильченко Е.А. Об одном алгоритме управления параллельными вычислениями с децентрализованным управлением. // Вестник Тамбовского университета. Серия: Естественные и технические науки. Выпуск 4-1, том 18, 2013.

8. Малашонок Г.И. Управление параллельным вычислительным процессом // Вестник Тамбовского университета. Сер. Естественные и технические науки. Тамбов, 2009. Том 14. Вып. 1. С. 269-274.

9. Strassen V. Gaussian Elimination is not optimal // Numerische Mathematik. 1969. **13**, 354-356.

10. Малашонок Г.И. Матричные методы вычислений в коммутативных кольцах. Тамбов: Изд-во Тамбовского университета, 2002.

11. Малашонок Г.И. О вычислении ядра оператора действующего в модуле // Вестник Тамбовского университета. Сер. Естественные и технические науки. Тамбов, 2008. Том 13, вып. 1. С. 129-131

12. Malaschonok G.I. Effective Matrix Methods in Commutative Domains // Formal Power Series and Algebraic Combinatorics. Berlin: Springer, 2000. P. 506-517.

13. https://ru.wikipedia.org/wiki/Китайская_теорема_об_остатках

14. http://mathpar.cloud.unihub.ru

The efficiency of dynamic decentralized control for parallel matrix inversion algorithms

E.A. Ilchenko

Tambov State University

We discuss different approaches to the parallelization of tree-like recursive algebraic algorithms. For such algorithms we suggest several schemes for management of parallel computing processes, which implement the concept of decentralized dynamic parallelization and which using MPI. We compare effectiveness of such schemes. The results of experiments with parallel program are shown for matrix inversion and calculations of echelon forms on the cluster MVS-10P. The comparative analysis of experiments are presented for matrices in the field of rational numbers and matrices in the final fields.

Keywords: dynamic decentralized control, DDP, adjoint, MVS-10P, MathPartner, MPI, multithreading, POSIX treads, C++, GMP.

References

1. Jean-Guillaume Dumasa, Thierry Gautierb, Clement Pernet, Jean-Louis Rochb, Ziad Sultana. Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination // Parallel Computing, Volume 57, September 2016, Pages 235-249.

2. A. R. Benson and G. Ballard. A framework for practical parallel fast matrix multiplication // In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 42-53, New York, NY, USA, 2015. ACM.

3. P. D'alberto, M. Bodrato, and A. Nicolau. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation // ACM Trans. Math. Softw., 38(1):2:1-2:30, Dec. 2011.

4. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages // ACM Trans. on Mathematical Software(TOMS), 35(3):1-42, 2008.

5. Ilchenko E.A. About effective methods parallelizing block recursive algorithms // Vestnik Tambovskogo universiteta. Ser. Estestvennye i tekhnicheskie nauki [Tambov University Reports. Series: Natural and Technical Sciences], 2015, vol. 20, issue 5 (in Russian).

6. Ilchenko E.A. Tools of mathematical service "MathPartner" for parallel computations on a cluster // Trudy ISP RAN [Proc. ISP RAS], 2016, vol. 1, issue 3 (in Russian).

7. Ilchenko E.A. An algorithm for the decentralized control of parallel computing process // Vestnik Tambovskogo universiteta. Ser. Estestvennye i tekhnicheskie nauki [Tambov University Reports. Series: Natural and Technical Sciences], 2013, vol. 18, issue 4 (in Russian).

8. Malaschonok G. Managing of the parallel computational process // Vestnik Tambovskogo universiteta. Ser. Estestvennye i tekhnicheskie nauki [Tambov University Reports. Series: Natural and Technical Sciences], 2009, vol. 14, issue 1, pp. 269-274 (in Russian).

9. Strassen V. Gaussian Elimination is not optimal // Numerische Mathematik. 1969. **13**, 354-356.

10. Malaschonok G.I. Matrix computational methods in commutative rings. Tambov, TSU, 2002, 213 p (in Russian).

11. Malaschonok G.I. On computation of kernel of operator acting in a module // Vestnik Tambovskogo universiteta. Ser. Estestvennye i tekhnicheskie nauki [Tambov University Reports. Series: Natural and Technical Sciences], vol. 13, issue 1, 2008. P. 129-131.

12. Malaschonok G.I. Effective Matrix Methods in Commutative Domains // Formal Power Series and Algebraic Combinatorics. Berlin: Springer, 2000. P. 506-517.

13. https://ru.wikipedia.org/wiki/Китайская_теорема_об_остатках

14. http://mathpar.cloud.unihub.ru