Параллельное программирование в среде *Java* для систем с распределенной памятью. Объектные модели параллельного выполнения

С.С. Гайсарян, М.В. Домрачев, В.Ф. Еч,
О.И. Самоваров, А.И. Аветисян

1. Введение

Цель настоящей работы[*] состоит в исследовании возможностей включения в стандартную среду языка программирования *Java* [1] средств, поддерживающих разработку параллельных программ в рамках модели *SPMD* для параллельных вычислительных систем с распределенной памятью. При этом имеется в виду два класса таких систем:

- многопроцессорные параллельные компьютеры с распределенной памятью;
- локальные сети рабочих станций.

Многопроцессорные параллельные компьютеры с распределенной памятью обычно состоят из нескольких десятков (а иногда и сотен) процессоров (узлов), объединенных в сеть быстрыми коммутирующими устройствами. Примером такой вычислительной системы может служить компьютер *FatNode* фирмы *SUN*, содержащий 64 узла *UltraSparc*. Быстрые системы коммутации, применяемые для объединения компьютеров в локальную сеть, позволяют, объединяя несколько *FatNode,* получать системы с большим числом узлов.

Локальные сети рабочих станций представляют особый класс вычислительных систем с распределенной памятью. В отличие от распределенных компьютеров, сети рабочих станций, как правило, неоднородны (т.е. содержат компьютеры различных типов и с разной производительностью), а это приводит к дополнительным накладным расходам, которые не только замедляют выполнение параллельной программы, но и влияют на такие существенные ее свойства, как масштабируемость. Поэтому параллельные программы, отлаженные на сети рабочих станций, вообще говоря, не будут сохранять своих свойств при переносе на параллельный компьютер.

В модели *SPMD* (*Single Program, Multiple Data*) [2] параллельная программа рассматривается как набор функционально одинаковых компонент (последовательных программ). Каждая компонента выполняется независимо на отдельном процессоре параллельной вычислительной системы, обрабатывая ту часть распределенных данных, которая попала на этот процессор. Если при этом требуются данные с других процессоров, то они пересылаются по сети. Модель *SPMD* используется во многих современных системах параллельного программирования (см., например, [3 – 6]).

Язык программирования *Java*, впервые опубликованный в 1995 году фирмой *SUN*, быстро завоевал популярность и стал одним из наиболее распространенных языков программирования. Это объясняется не только тем, что фирма *SUN* затратила огромные средства на внедрение *Java,* но и многими достоинствами языка *Java* и его окружения. Отметим наиболее существенные из них:

1. *Полная переносимость программ* – программа на языке *Java* работает на любой программно-аппаратной платформе, поддерживающей среду *Java*, не требуя каких-либо модификаций или преобразований. Это свойство позволяет избежать процесса установки программы на конкретную платформу для оценочного запуска. Как следствие, при распределенном и параллельном программировании снимаются проблемы неоднородности вычислительной среды, требующие достаточно больших накладных расходов во время выполнения.
2. *Простота объектной модели*, реализованной в системе *Java*, позволяет легко разбираться в исходной программе, облегчает внесение в нее изменений и дополнений во время отладки и сопровождения, упрощает документирование программы. При этом система программирования *Java* отвечает всем требованиям к современным системам программирования: наличие механизма исключительных ситуаций, поддержка легковесных процессов (тредов), потоковый

[*] Работа поддержана грантом РФФИ 99-01-00206

ввод/вывод, поддержка средств сетевого программирования (библиотека сокетов), графический интерфейс пользователя, системы обеспечения безопасности при пересылке данных и байт-кода по сети и т.д.

3. *Детерминированность получаемого кода* – свойство языка, не позволяющее программисту написать программу, в которой возможны несанкционированные действия, такие как обращение к объектам с помощью "висячих" указателей или преобразование случайного участка памяти в экземпляр объекта в результате ошибки в адресной арифметике.

Переносимость программы на *Java* позволяет пользователю получать исходный код (или байт-код) программы и использовать его на своей платформе без каких бы то ни было доработок. Введение в среду *Java* возможности организации параллельных вычислений позволит разрабатывать параллельные программы, переносимые на различные параллельные платформы. Это существенно упростит распространение параллельных программ (например, через *Internet*).

Отметим также, что средства написания параллельных программ на *Java* дадут возможность пользователям, ранее не имевшим доступа к параллельным вычислительным ресурсам, использовать параллельные высокопроизводительные вычислительные комплексы, доступные через *Internet*. Такая организация доступа к параллельным вычислительным ресурсам позволит эффективно выполнять разовые научные или инженерные расчеты, требующие большого объема вычислений, без необходимости установки специального программного обеспечения и без дополнительной компиляции исходной программы для конкретной параллельной архитектуры.

Однако существуют обстоятельства, препятствующие использованию среды языка *Java* для разработки параллельных программ. Они связаны со следующими особенностями системы программирования *Java:*

1. *Java* – интерпретируемый язык: объектным кодом системы программирования *Java* является не код процессора, на котором выполняется *Java*-программа, а байт-код (*JavaBC*), язык интерпретатора *Java*-программ (*JavaVM*). Интерпретация байт-кода, естественно, требует дополнительных накладных расходов и приводит к замедлению выполнения *Java*-программ. Разработчики системы *Java* утверждают, что накладные расходы, связанные с интерпретацией, приводят к замедлению в 2 – 2,5 раза, однако на практике было выяснено, что замедление может быть и более существенным (до 10 раз).
2. В системе *Java* запрещены какие бы то ни было изменения и расширения. Запрещено вводить новые конструкции в язык *Java* и в *JavaBC*, запрещено менять *JavaVM,* запрещено использовать препроцессоры. Единственным разрешенным способом расширять язык *Java* является разработка новых классов и библиотек классов.

Запрещение любых способов внесения изменений в язык *Java* или в его интерпретатор, что дало бы возможность обойти ограничения, следует рассматривать как цену, которую необходимо платить за удобство программирование на *Java,* за надежность и переносимость *Java*-программ, за возможность их выполнения в произвольной компьютерной сети. В данной работе исследуется возможность введения параллельных средств в язык *Java* средствами самого языка *Java*, а именно с помощью библиотек интерфейсов и классов. Эти интерфейсы и классы в своей совокупности могут рассматриваться как объектная модель системы параллельного программирования.

В настоящее время работы по организации параллельных вычислений на *Java* вызывают большой интерес среди различных групп, работающих над созданием систем параллельного программирования. Краткий обзор работ по параллельным расширениям *Java* содержится в разделе 2.

В разделе 3 рассматриваются два способа организации параллельных вычислений в рамках модели *SPMD* (их можно рассматривать как конкретизацию этой модели): модель *DPJ* (*Data Parallel Java*) и модель *DVM* (*Distributed Virtual Machine*). Сравнение моделей *DPJ* и *DVM* показывает, что модель *DPJ* лучше учитывает особенности программирования в объектно-ориентированном окружении *Java*.

В разделе 4 рассматриваются примеры параллельных программ, использующих библиотеки *DPJ* и *JavaDVM,* и анализируются способы повышения их производительности. Там же указывается адрес указанных библиотек классов на сайте *Internet*.

За рамками статьи осталось описание библиотеки, реализующей основные функции стандарта передачи сообщений *MPI* [7] на языке *Java*. Эта библиотека дает возможность пользоваться библиотеками *DPJ* и *DVM*, не выходя за рамки среды *Java*, т.е. существенно упрощает доступ к этим библиотекам для пользователей системы *Java*.

2. Краткий обзор работ по параллельному расширению языков программирования и обоснование выбранного подхода к параллельному расширению языка Java

Как было отмечено во введении, в настоящее время работы по организации параллельных вычислений на *Java* вызывают большой интерес среди различных групп, работающих над созданием систем параллельного программирования. Об этом можно судить по регулярным публикациям по этому вопросу в периодических изданиях и на *Internet* (см., например, [3]).

В настоящее время применяется несколько парадигм параллельного программирования:

- использование низкоуровневых библиотек передачи сообщений для обычных языков программирования;
- специальные указания: аннотации, прагмы, или параллельные операторы, – как правило, реализованные в виде комментариев, – которые вставляются в последовательную программу и помогают компилятору, который учитывает эти указания, сгенерировать соответствующую параллельную программу;
- новые параллельные языки (в том числе расширения последовательных языков новыми конструкциями для описания параллельных вычислений).

Использование низкоуровневых библиотек передачи сообщений для обычных языков программирования предполагает, что программист, используя один из стандартных последовательных языков, пишет узловые программы, запоминает, на каких узлах расположены какие данные, определяет источник и приемник данных для каждого коммуникационного акта, вызывает функции преобразования данных (если в параллельной программе используются процессоры различной архитектуры), "вручную" осуществляет балансировку вычислительной и коммуникационной нагрузки узловых программ. Оптимизация коммуникаций и отслеживание масштабируемости программы также ложится на программиста. Зачастую эти операции являются противоречивыми, например, программа, хорошо сбалансированная для четырех процессоров, нуждается в повторной балансировке для восьми процессоров, т.е. возникают проблемы, связанные с масштабированием программы. При большом количестве процессоров в параллельном компьютере написание эффективной программы становится очень сложным и трудоемким занятием. Отметим, что такой способ разработки параллельных программ поддерживался системой программирования *Occam* [8], разработанной в конце 80-х годов для транспьютерных сетей.

Сказанное можно пояснить с помощью простого примера параллельной программы на языке *C* с использованием стандартной библиотеки передачи сообщений *MPI* (рис. 1). В примере выделяется группа из двух процессоров из вычислительного пространства и от первого ко второму передается массив целых чисел.

```
#include "mpi.h"
#define BUFLEN 512
#define GRP_SIZE 2

int main(int argc, char* argv[])
{
  int i, myid, numprocs, rc;
  int buffer[BUFLEN];
  MPI_Status status;
  MPI_Group MPI_GROUP_WORLD, grp1;
  MPI_Comm newComm;
  int hosts[3] = {0,1,2};

  MPI_Init(&argc,&argv);
  MPI_Comm_group(MPI_COMM_WORLD,
&MPI_GROUP_WORLD);

MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  If( numprocs < GRP_SIZE ) exit(1);
  MPI_Group_incl(MPI_GROUP_WORLD, 3,
hosts, &grp1);
  MPI_Comm_create(MPI_COMM_WORLD, gr2,
&newComm);

  MPI_Comm_rank(newComm, &myid);
  if(newComm != MPI_COMM_NULL) {
    if (myid == 0) {
      // Инициализация буфера опущена
      . . .
      MPI_Send(buffer, BUFLEN, MPI_INT,
1, 99, MPI_COMM_WORLD);
    } else {
      MPI_Recv(buffer, BUFLEN, MPI_INT,
0, 99, MPI_COMM_WORLD,
      &status);
    }
    MPI_Comm_free(&newComm);
  }
  MPI_Group_free(&grp1);
  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Finalize();
}
```

Рис. 1. Простая параллельная программа на языке C с использованием пакета MPI для передачи данных между процессорами

Как видно из примера, даже несложные действия влекут за собой большой размер получаемого кода, что способствует возникновению ошибок, причем ошибки эти достаточно трудно найти и исправить, так как необходимо одновременно следить за состоянием многих работающих программ.

Кроме того, хотя большая часть библиотек, реализующих передачу сообщений, и реализована практически на всех известных платформах, сохраняются проблемы переноса программы на обычном процедурном языке с платформы на платформу. Свойства масштабируемости такой программы также меняются при переносе на другую платформу, и ее приходится заново оптимизировать и отлаживать или частично переписывать.

Таким образом, программист, использующий эту парадигму, сталкивается со сложными проблемами на всех циклах разработки программы: при ее написании, отладке, оптимизации и сопровождении.

Тем не менее, приходится констатировать, что в настоящее время подавляющее большинство прикладных параллельных программ разрабатывается именно таким способом. В качестве стандартных языков программирования обычно используются *Fortran 77, Fortran 90,* или *C.* Передача данных между узлами параллельного компьютера обычно реализуется с помощью библиотек *PVM* [9], или *MPI* [7]. Существуют многочисленные реализации этих библиотек, как коммерческие, так и свободно доступные (в том числе, через *Internet*).

Вторая парадигма связана с использованием аннотаций, или прагм, добавленных в стандартный последовательный язык программирования в форме специальных комментариев. Программа, написанная на таком языке, может проходить и через стандартный последовательный компилятор с этого языка, и через специальный

распараллеливающий компилятор, который понимает специальные комментарии и использует информацию, содержащуюся в них, для генерации параллельной программы. Обычно аннотации содержат сведения о распределении данных по узлам параллельного компьютера. Распределение вычислений и синхронизация передачи данных осуществляются компилятором автоматически.

Наиболее известным языком, реализующим рассматриваемую парадигму, является *HPF* (*High Performance Fortran*) [4].Пример, рассмотренный выше, на *HPF* может быть представлен следующим образом (рис. 2).

```
      C$HPF PROCESSORS P(2)
        INTEGER A(512*2)
      C Инициализация A(1..512)
      . . . . . . . . . . . . . . .
      C$HPF DISTRIBUTE A(512) ON P
        DO 1 I=1,512
        A(I+512) = A(I)
  1     CONTINUE
      STOP
```

Рис. 2. Параллельная программа с рис.1, записанная на языке HPF

В приведенном примере C является признаком строки комментария, а C$HPF – признаком специального комментария, в котором задаются аннотации *HPF*. По информации, содержащейся в строках программы, начинающихся с символов C$HPF, компилятор *HPF* должен выяснить, где происходит реальная пересылка данных, и выбрать правильную стратегию их передачи (например, использовать буферизацию). При использовании *HPF* сбалансированность нагрузки, оптимальность коммуникаций и свойства масштабируемости программы практически целиком зависят от того, какие алгоритмы и эвристики реализованы в конкретном компиляторе и его системе поддержки. Программист практически не влияет на процесс оптимизации параллельной программы. Этим объясняется сравнительно редкое использование *HPF* при разработке реальных прикладных программ. Появление новой версии *HPF* – языка *HPF-2*, обеспечивающего ряд дополнительных возможностей, не увеличило популярности *HPF* среди разработчиков прикладных параллельных программ. Это привело к закрытию исследований по *HPF* и официальному признанию бесперспективности направления.

Тем не менее, другие системы параллельного программирования, в которых используется подход, близкий к *HPF*, продолжают разрабатываться. Одной из таких работ является модель *DVM*, рассматриваемая в данной работе. Эта модель идеологически очень близка к модели, лежащей в основе *HPF*. Однако в отличие от *HPF,* основу системы *DVM* составляет не возможность более просто описывать распределение данных по сети с помощью аннотаций, а библиотеки инструментальных программ, использующие аннотации для трассировки и мониторинга параллельных программ, что помогает их отлаживать и улучшать степень их распараллеленности (своеобразная диалоговая оптимизация). Инструментальные библиотеки системы *DVM* обеспечили ее практическое использование при разработке прикладных параллельных программ. Другой работой, идеологически близкой к *HPF*, является разработка языка и системы программирования *Linda* [10]. В этой системе программирования реализована модель *VSM* (*Virtual Shared Memory*). http://www.qpsf.edu.au/workshop/linda

Третье решение – создать новый язык программирования с высокоуровневыми абстракциями для параллельного программирования и явными параллельными конструкциями. Примером такого языка может служить язык *mpC* [11], разработанный и реализованный в ИСП РАН. Рассматриваемый пример запишется на *mpC* следующим образом (рис. 3).

```
#include "mpc.h"

#define BUFLEN 512
nettype SimpleNet(n) { coord I=n; }

void [*]main() {
  net SimleNet(2)n;
  int [n]a[BUFLEN];
  [n:I==0] { /* Инициализация буфера a
*/ }
  [n:I==1]a[] = [n:I==0]a[];
}
```

Рис. 3. Параллельная программа с рис.1, записанная на языке mpC

По этому тексту компилятор *mpC* автоматически сгенерирует программу на *C* с вызовами функций из библиотеки *MPI*, похожую на программу, приведенную на рис. 1. Легко видеть, что исходная программа получилась значительно проще, чем в случае непосредственного использования *MPI*.

Среди работ, близких к *mpC,* можно отметить еще одно параллельное расширение *C* – язык *Charm/Charm++* [12]. Как и *mpC, Charm* представляет собой систему параллельного программирования, причем *Charm++* является расширением языка *C++* (т.е. объектно-ориентированным языком параллельного программирования).

К сожалению, разработчики новых языков параллельного программирования, как показывает анализ их языков, имеют весьма скромный опыт разработки оригинальных прикладных параллельных программ. Они плохо представляют, какие этапы разработки параллельной программы требуют автоматизации в первую очередь. И они совершенно игнорируют то обстоятельство, что внедрение нового языка программирования требует больших затрат, связанных с необходимостью обеспечения возможности использования в новом языковом окружении внешних библиотек и прикладных программ, разработанных в других организациях.

Между тем, анализ неудачи, постигшей *HPF*, показал, что при разработке параллельных

программ прикладным программистам удобнее в явном виде задавать пересылку данных между процессорами (ведь правильная организация пересылки данных – единственная возможность обеспечить высокую скорость вычислений). Несмотря на свою громоздкость, программа, приведенная на рис. 1, понятнее для своего разработчика, чем программы на рис. 2 и 3. Это особенно важно при отладке параллельной программы и при оптимизации ее параллельных свойств. Тут уместно вспомнить, что когда основная память компьютеров была достаточно скромных размеров (например, на БЭСМ-6 было всего 6 Mb), разработчики программ решения систем алгебраических уравнений не удовлетворялись системной виртуальной памятью, а писали свои собственные системы замещения страниц, что позволяло им существенно ускорить свои программы.

Таким образом, можно утверждать, что высокоуровневые универсальные модели параллельных вычислений, лежащие в основе языков параллельного программирования высокого уровня, не были приняты программистским сообществом. Следовательно, основные усилия разработчиков систем параллельного программирования необходимо направить не на разработку новой модели параллельного программирования и ее представления в виде нового языка параллельного программирования, а на инструментальные средства, помогающие разрабатывать, оптимизировать и отлаживать параллельные программы в рамках старой, хорошо понятной прикладным программистам модели передачи сообщений. Что касается повышения уровня параллельного программирования, то от универсальных моделей следует перейти к специализированным (многие такие модели, например для задач линейной алгебры уже разработаны [13]). А для разработки специализированных моделей, как известно, наиболее удобны объектно-ориентированные языки программирования.

Сейчас наиболее популярны два языка объектно-ориентированного программирования – *C++* и *Java.* Каждый из них имеет свои достоинства и недостатки. Так, *C++* более приближен к компьютеру, на нем можно писать столь же эффективные программы, как и на *C*, расширением которого он является. Поэтому большинство объектно-ориентированных языков параллельного программирования базируется на *C++*. *Java* – интерпретируемый язык: *Java*-программы выполняются на интерпретаторе *Java* – *JavaVM,* что, естественно, замедляет их выполнение. Но зато *Java*-программы являются полностью переносимыми, так как окружение *Java* одинаково на всех компьютерах. И если первое обстоятельство заставляет воздержаться от применения *Java* в параллельном программировании, то независимость системы программирования *Java* от особенностей аппаратуры компьютера, несомненно, полезна для параллельных программ.

В модели *SPMD* на всех узлах (процессорах) компьютерной сети выполняется одна и та же *последовательная* программа. Следовательно, для описания параллельных вычислений в рамках этой модели нет необходимости вводить какие-либо новые операторы языка, достаточно ввести средства описания распределения данных по узлам и операции обмена данными между узлами. В объектно-ориентированном языке обе эти проблемы можно решить с помощью подходящих библиотек классов. Для обеспечения эффективности необходимо, чтобы эти библиотеки классов опирались непосредственно на новую систему поддержки выполнения скомпилированной программы (*run-time system*), являющуюся расширением соответствующей системы исходного объектно-ориентированного языка. От эффективности расширенной системы поддержки выполнения существенно зависит эффективность параллельных программ, разрабатываемых в ее рамках.

При этом если расширяемым объектно-ориентированным языком является язык *Java*, необходимо, чтобы расширение системы поддержки выполнения не затрагивало *Java VM,* т.е. было внешним по отношению к *Java VM.* Если это условие не выполняется (как, например, в работе [14]), то вместо стандартной *Java VM,* получается расширенная *Java VM,* байт-код которой содержит дополнительные инструкции, отражающие специфику параллельного выполнения. Конечно, такое решение позволяет повысить эффективность выполнения (интерпретации) параллельной *Java*-программы, но оно противоречит требованиям системы программирования *Java.* Параллельные программы, написанные на таком расширении, могут выполняться лишь на тех параллельных вычислительных комплексах, на которых установлена расширенная *Java VM*. При реализации расширения системы поддержки выполнения вне рамок стандартной *Java VM* параллельная программа может выполняться на всех процессорах, на которых установлена система программирования *Java.* При этом достаточно высокая эффективность параллельной программы может быть достигнута за счет реализации подпрограмм, расширяющих стандартную *Java VM,* на языке ассемблера соответствующего процессора (*Java VM* позволяет вызывать и выполнять подпрограммы на языке процессора, на котором она установлена).

В данной работе рассматриваются расширения системы программирования *Java* с помощью системных библиотек классов. Такие расширения не требуют для своей работы ни расширения *Java VM*, ни специальных компиляторов или препроцессоров. Только такой подход позволяет разрабатывать параллельные программы, переносимые как на новую аппаратно-программную платформу, так и на новую версию системы программирования *Java*. Пример параллельной программы с рис. 1 на языке *Java* с

использованием библиотеки классов *DPJ* приведен на рис. 4.

В основе предлагаемых библиотек классов лежат объектные модели параллельных вычислений. Как уже отмечалось, в работе рассматривается две такие модели: модель *DPJ* (п. 3.1) и модель *DVM* (п. 3.2).

3. Объектные модели параллельных вычислений

3.1 Модель *DPJ*

Модель *DPJ* [15] относится к классу моделей *SPMD*, в котором параллельная программа рассматривается как набор функционально одинаковых компонент (последовательных программ). В модели *DPJ* каждая компонента представляет собой последовательную *Java*-программу, выполняемую на отдельной *JavaVM*, причем все компоненты и их *JavaVM* работают одновременно на процессорах параллельного компьютера.

Для организации взаимодействия и обмена данными между компонентами используется пакет передачи сообщений *MPI* (*Message Passing Interface*) [7]. Пакет *MPI* позволяет скрыть физическую структуру процессоров комплекса, используемых сетевых средств, а также особенности операционных систем, представляя набор примитивов для управления передачей сообщений. Одним из важных примитивов пакета *MPI* является процесс. Процесс – это единица исполнения программы, причем обмен сообщениями в *MPI* производится только между процессами. *MPI* позволяет запускать потенциально бесконечное множество процессов на параллельном вычислительном комплексе. В работе каждой компоненте параллельной *Java*-программы ставится в соответствие один процесс *MPI.* Таким образом, параллельная *Java*-программа отображается на множество процессов *MPI.* Это множество процессов *MPI* удобно называть сетью, а каждый процесс – узлом этой сети. Сеть концептуально создается с помощью специальных классов запуска ее множества узлов, определяющих ее мощность и параметры запуска параллельной программы.

В рамках принятой модели компоненту параллельной программы можно ассоциировать с узлом, на котором она выполняется. Поскольку на каждом узле выполняется *Java*-программа, в ней можно применять средства параллелизма, определенные в языке *Java* (классы Thread, ThreadGroup), использующие модель параллелизма на общей памяти. В работе рассматривается только модель параллелизма на распределенной памяти, т.е. предполагается, что каждая компонента параллельной программы работает в своем отдельном адресном пространстве, а трэды действуют в рамках одного узла, причем реальный параллелизм может достигаться, если сам узел имеет сложную структуру (т.е. процесс *MPI* выполняется на многопроцессорной архитектуре с поддержкой распределения трэдов по различным процессорам).

Определение подсетей. Выделение подсети некоторой сети или подсети осуществляется с помощью создания экземпляра объекта специального класса Subnet. Каждая подсеть имеет родительскую сеть или подсеть и может иметь произвольное количество сыновних подсетей. В текущей реализации библиотеки создание сети или подсети приводит к созданию соответствующих группы и коммуникатора *MPI*. Создание коммуникатора *MPI* требует одного акта коллективной коммуникации по соответствующей надсети.

Распределенные контейнеры. В языке *Java*, как и в других объектно-ориентированных языках, доступ к наборам однотипных объектов удобно представлять с помощью контейнерных объектов (*контейнеров*). Примерами контейнеров являются массивы, векторы, списки, множества, множества ключей и др. Каждый объект, доступ к которому организуется через контейнер, называется *элементом* этого *контейнера*. *Итератор* контейнера – это объект, который обеспечивает доступ к элементам контейнера. В последовательной программе в произвольный момент времени итератор обеспечивает доступ не более чем к одному элементу. Итератор удобно использовать для организации циклов обработки элементов контейнера.

Для реализации распараллеливания по данным введем *распределенный контейнер*, который размещает свои элементы по одному на каждом узле некоторой подсети *N*. Подсеть *N* указывается при определении (порождении) соответствующего распределенного контейнера.

Как и в обычном контейнере, каждый элемент распределенного контейнера может иметь ссылки на другие элементы этого контейнера: в этом случае ссылка указывает узел, на котором располагается элемент, адресуемый этой ссылкой. Над распределенным контейнером могут быть определены *операции типа свертки*, выполняемые параллельно с максимальной эффективностью над всеми его элементами. Примерами таких операций являются: подсчет количества элементов в контейнере, получение индекса элемента, сложение всех элементов контейнера (например, распределенного массива), нахождение максимума и минимума, и т.п.

Строгая типизация элементов контейнера может осуществляться с помощью типизированных итераторов и адаптеров. В настоящее время в библиотеку *DPJ* включены следующие распределенные контейнеры:

1. *Распределенный массив* – это распределенный контейнер с фиксированным количеством элементов (узлов), не имеющих связей между собой. К распределенным массивам применяются итераторы с произвольным доступом. Количество узлов в массиве задается при его определении и остается постоянным в процессе выполнения программы.

2. *Распределенный вектор* – отличается от распределенного массива лишь тем, что количество узлов в контейнере может изменяться в процессе выполнения программы.
3. *Распределенный список* – это распределенный контейнер, у которого заданы: головной узел, промежуточные узлы и хвостовой узел. Если список состоит из одного узла, то этот узел является головным и хвостовым одновременно. К распределенным спискам применяются итераторы с последовательным доступом. Количество элементов в списке может изменяться в процессе выполнения программы.
4. *Распределенное дерево* – это распределенный контейнер, у которого имеются корневой узел, нетерминальные узлы и листовые узлы. Если дерево состоит из одного узла, то этот узел является корневым и листовым одновременно, причем у него нет ни сыновнего, ни родительского узлов. В остальных случаях корневой узел не имеет родительского узла, но имеет несколько сыновних. Листовой узел не имеет сыновнего узла, но имеет один родительский. Все нетерминальные узлы имеют один родительский и несколько сыновних узлов, причем эти узлы являются различными. К распределенным деревьям применяются итераторы с последовательным доступом. Количество элементов в дереве может изменяться в процессе выполнения программы.

Итераторы распределенных контейнеров. *Итератор распределенного контейнера* – это объект, распределенный по той же подсети, что и контейнер, и обеспечивающий доступ одновременно ко всем элементам одного из подмножеств узлов этого контейнера. Это подмножество узлов контейнера называется *значением итератора*. Множество всех значений итератора образует покрытие множества узлов контейнера (с пересечениями или без пересечений). У итератора определены две операции: присваивание и переприсваивание его значения. С помощью этих операций итератор может попеременно принимать все свои значения, в совокупности составляющие покрытие множества всех узлов контейнера. Существует специальное значение итератора, множество узлов которого является пустым, которое используется для завершения итерационного процесса.

Итераторами распределенных контейнеров могут служить объекты классов, в которых реализован хотя бы один из интерфейсов, определенных в библиотеке.

- *Множественный итератор* (DMultiIterator) – это итераторы, значения которых есть подмножества множества узлов контейнера. Мощность этих подмножеств может меняться от значения к значению.
- *Унарный итератор* (DUnaryIterator) – это итератор, представляющий работу с множеством, состоящим из одного узла контейнера.
- *Полный итератор* (DAllIterator) – это итератор, единственным значением которого является множество всех узлов контейнера.

Параллельные алгоритмы. *Параллельный алгоритм* – это распределенный контейнер или итератор, специально оптимизированный для параллельного выполнения алгоритмов. В библиотеке *DPJ* содержатся стандартные классы, реализующие часто используемые параллельные алгоритмы для работы с распределенными контейнерами. Все библиотечные параллельные алгоритмы реализуют интерфейс ParallelAlgorithm, содержащий следующие методы:

- Запустить (остановить, или приостановить) метод run(), runTop(), или runBottom().
- Сами методы run(), runTop()и runBottom().

В настоящее время реализованы следующие параллельные алгоритмы:

- Applying – применяет указываемый функциональный объект к элементам контейнера.
- Copying – копирует распределенный контейнер (возможно, с обращением порядка элементов).
- Comparing – параллельное попарное сравнение элементов распределенных контейнеров.
- Finding – параллельный поиск элементов в распределенном контейнере по шаблону.
- Filtering – параллельный отбор элементов в распределенном контейнере по шаблону.
- Replacing – параллельная замена элементов распределенного контейнера.

```
import dpj.*;
public class Test {
  final private static int BUF_SIZE = 512;
  public static void main( String[] args )
{
    Subnet n = new Subnet( Subnet.netWorld,
2 );
    DArrayFixedIntArrayClass a = new
    DArrayFixedIntArrayClass( n, BUF_SIZE
);
    DMyIterator init = new DMyIterator( a,
0 );
    init.start();
    A.putValue( 1, a.getValue( 0 ) );
  }
}

class DMyIterator extends
DunaryInputIterator {
  DMyIterator( int root ) { super( root );
}
  public void run() {
    int[] a = (int[])container().get();
    // Инициализация массива
    container().put( a );
  }
}
```

Рис. 4. Параллельная программа с рис.1, записанная на языке *Java* с использованием библиотеки классов *DPJ*

- Sorting – параллельная сортировка элементов распределенного контейнера.

- Transforming – параллельное преобразование распределенного контейнера.

На рис. 4 представлена программа с рис.1, реализованная с помощью библиотеки *DPJ.* Это обычная *Java*-программа, она не содержит новых языковых конструкций и потому привычна для программиста, который ее разрабатывает. Использование распределенных параллельных вычислений осуществляется с помощью точно таких же языковых средств (библиотечных классов), как и ввод-вывод или использование параллельных вычислений над общими данными (трэды языка *Java*).

3.2 Модель *DVM*

Модель *DVM* (*Distributed Virtual Machine*) [6], разработанная в ИПМ им. М.В. Келдыша РАН, тоже относится к классу моделей *SPMD*. В основе модели *DVM* лежат понятия абстрактной параллельной машины (АПМ) и виртуальной параллельной машины (ВПМ). АПМ представляет собой многомерный массив абстрактных параллельных подсистем, каждая из которых является многомерным массивом процессоров, либо подсистем следующего уровня и далее по рекурсии. АПМ может быть задана в программе статически, или построена динамически при выполнении программы. Динамическое построение АПМ необходимо при работе с динамическими массивами, а также для использования библиотек стандартных параллельных программ, хранящихся в виде объектных модулей и не требующих предварительной настройки. При разработке программы для АПМ пользователь исходит из следующей модели ее выполнения. В момент старта программы существует единственная ее ветвь (поток управления), которая выполняется на одном из процессоров АПМ, начиная с первого оператора программы. При входе в параллельную конструкцию, например, параллельный цикл или группу параллельных секций, ветвь разбивается на некоторое количество параллельных ветвей, каждая из которых выполняется на одном процессоре (или подсистеме) АПМ. При выходе из параллельной конструкции все ветви снова сливаются в единственную ветвь, которая выполнялась до входа в эту конструкцию. В этот момент все изменения разделяемых переменных, которые были произведены параллельными ветвями, становятся видны всем процессорам, выполняющим программу.

Формальная зависимость параллельных ветвей по данным устраняется путем объявления некоторых переменных приватными. Тогда для каждой параллельной ветви заводится своя копия (неинициализированная) таких переменных, которая используется независимо от других. Все остальные данные, используемые параллельными ветвями, считаются разделяемыми. Режим доступа к разделяемым переменным в параллельных ветвях может быть специфицирован как только чтение; распределенный доступ (например, разные витки цикла используют различные элементы массива) и произвольный доступ. В случае произвольного доступа, программист должен осуществлять явную синхронизацию параллельных ветвей.

ВПМ – это машина, которая предоставляется задаче пользователя аппаратурой и базовым системным программным обеспечением. Эта машина по своей структуре и количеству процессоров должна быть близка к реальной параллельной ЭВМ. Примером ВПМ может служить *MPI*-машина. ВПМ также может быть представлена в виде иерархии подсистем. При этом должен быть однозначно определен состав каждой подсистемы – номера входящих в нее процессорных узлов ВПМ. Каждая подсистема может содержать только те процессоры, которые входят в состав ее материнской подсистемы. Модель *DVM* не предполагает динамического изменения состава ВПМ, поскольку в распределенных системах подключение нового процессора требует переноса на него контекста программы и связано с большими накладными расходами.

Отображение АПМ на ВПМ машину заключается в задании соответствия между подсистемами АПМ и ВПМ, в результате которого каждая подсистема АПМ будет отображена на подсистему или процессор ВПМ. При этом используются два метода задания такого соответствия:

- Отображение указанной подсистемы АПМ на указанную подсистему ВПМ. Ветвь, отображенная на такую подсистему АПМ, будет выполняться на всех процессорах соответствующей подсистемы ВПМ.
- Регулярное (блочное или циклическое) распределение всех дочерних подсистем указанной подсистемы АПМ между процессорами указанной подсистемы ВПМ. При этом подсистема АПМ (и ее дочерние подсистемы) специфицируется путем задания одного из своих представлений.

При распараллеливании программы на системах с распределенной памятью необходимо задать расположение данных в локальной памяти процессоров АПМ. Это осуществляется посредством выравнивания массивов между собой и отображением их на подсистему АПМ в соответствии с указанным ее представлением в виде многомерного массива подсистем следующего уровня иерархии. При этом допускается сжимающее отображение (все элементы какого-то измерения отображаются на одну подсистему) и размножающее отображение (один элемент массива дублируется на многих подсистемах). Дублирование данных производится в тех случаях, когда их вычисления выгоднее выполнять многократно на разных процессорах вместо того, чтобы заниматься их пересылкой между процессорами. Кроме того, дублируются данные, о расположении которых пользователь указаний не задал. Эти данные дублируются по всем процессорам подсети, на которой выполняется ветвь, содержащая описания этих данных.

Описание объектной модели библиотеки *Java-DVM.* Библиотека классов *Java-DVM* является реализацией модели *DVM* в среде *Java*. Распределение данных в *Java-DVM* осуществляется путем "разрезания" некоторых массивов на части и размещения этих частей на разных *JavaVM*. Такое "разрезание" осуществляется путем разделения какого-либо измерения массива на отрезки. При этом многопроцессорная система из P процессоров, на которой будет выполняться параллельная программа, рассматривается как многомерная решетка (матрица) процессоров, например, как двумерная решетка размера *P1* на *P2* (*P1*P2=P*). В этом случае одно из измерений массива может быть разделено на *P1* отрезков, а какое-либо другое – на *P2* отрезков. В результате массив будет разрезан на *P1*P2* секций, каждая из которых будет отображена на соответствующий процессор двумерной решетки. Такие массивы называются распределенными массивами. Остальные массивы (а также скаляры) называются размноженными и размещаются на каждом процессоре целиком.

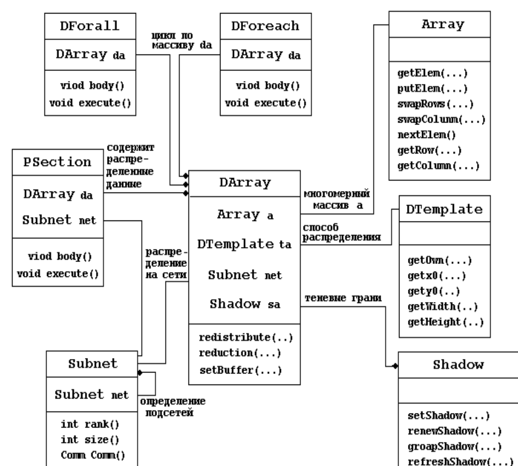
Доступ к удаленным данным осуществляется путем их буферизации в памяти процессоров и обменом буферов сразу для нескольких витков цикла одновременно. Выделение буферов для хранения копий удаленных данных, посылка и прием сообщений с копиями удаленных данных, а также замена обращений к удаленным данным на обращения к их копиям в буферах – все это осуществляется методами классов библиотеки *Java-DVM*.

На рис. 5 представлена объектная модель библиотеки Java-DVM. Описание виртуальной параллельной машины осуществляется методами класса Subnet. Для описания абстрактной параллельной машины предназначены следующие классы: класс Darray, реализующий параллельные массивы; классы Dforall и DForeach, реализующие параллельные циклы, класс PSection реализующий группу параллельных секций. Отображение АПМ на ВПМ осуществляется методами класса Dtemplate, а буферизация и локализация удаленных данных – методами класса Shadow. Класс Array позволяет моделировать многомерные массивы с помощью одномерных.

Распределение данных. В библиотеке *Java-DVM* многомерные массивы моделируются на одномерном массиве. Каждый многомерный массив является объектом класса Array. В рамках этого класса реализованы операции доступа к элементам массива и его подмассивов, а также некоторые стандартные операции над массивом: перестановка строк, столбцов, транспонирование и пр. Способ распределения подмассивов данного массива между процессорами задается в объектах класса Dtemplate. Реализованы следующие три способа распределения данных:

- **Блочный**: элементы *i*-го измерения массива отображаются на процессоры непрерывными блоками.
- **Циклический**: элементы *i*-го измерения массива отображаются на процессоры циклически.
- **Отображаемый**: элементы массива не распределяются между процессорами, а целиком отображаются на каждый процессор.

Распределение вычислений. Параллельная программа, использующая библиотеку классов *Java-DVM***,** выполняется в модели *SPMD*: на все *JavaVM* загружается одна и та же программа, но каждая *JavaVM* в соответствии с *правилом собственных вычислений* выполняет только те операторы присваивания, которые изменяют значения переменных, размещенных на ней. Таким образом, вычисления распределяются в соответствии с размещением данных (параллелизм по данным). В случае размноженной переменной оператор присваивания выполняется на всех процессорах, а в случае распределенного массива – только на процессоре (или процессорах), где размещен соответствующий элемент массива. Ответственность за согласованное распределение вычислений и данных полностью возлагается на пользователя.



.Рис 5. Объектная модель *Java-DVM*

Отображение витков параллельного цикла осуществляется блочно – на каждый процессор отображается сразу несколько соседних витков цикла. Кроме того, если имеется гнездо тесно вложенных циклов, оно может быть специфицировано как единый многомерный параллельный цикл. Для реализации параллельных циклов используются абстрактные классы DForall и Dforeach, имеющие абстрактный метод Body. Создав класс, родителем которого является один из этих классов, можно описать тело параллельного цикла. Витки цикла DForall могут выполняться в произвольном порядке. Витки цикла DForeach выполняются строго последовательно, в рамках одного процессора. Часто в программе встречаются циклы, в которых выполняются редукционные операции – в некоторой переменной суммируются элементы массива или находится их максимальное

или минимальное значение. Эти циклы так же могут выполняться параллельно, методом reduce класса DArray.

Организация доступа к удаленным данным. После спецификации распределения данных и витков циклов программа готова к параллельному выполнению при условии, что все данные, необходимые процессору, размещены на этом процессоре. Для большинства программ это условие не выполняется. Предусмотрены средства, позволяющие использовать удаленные данные. Импортируемыми данными процессора будем называть используемые им данные, расположенные на других процессорах. Основной способ оптимизации доступа к удаленным данным уменьшение количества импортируемых данных. Это достигается совместным распределением нескольких массивов (выравнивание массивов). При выравнивании массивов возможны следующие виды соответствия:

- совмещение двух массивов одинаковой формы (одинаковой размерности и с одинаковыми размерами в каждом измерении);
- совмещение двух массивов одинаковой формы с реверсом (первому элементу одного массива соответствует последний элемент другого);
- совмещение двух массивов с поворотом (*j*-ое измерение одного массива совмещается с *k*-ым измерением другого);
- вложение меньшего массива в больший со сдвигом, не приводящим к выходу за пределы большего массива;
- вложение меньшего массива в больший с раздвижкой (например, каждому элементу первого массива с индексом *i* соответствует элемент второго массива с индексом *2*i*);
- размножение массива меньшей размерности;
- отображение массива меньшей размерности на секцию другого массива, получающуюся путем фиксации индексов в некоторых его измерениях;
- сжатие измерения массива большей размерности;
- выравнивание массивов (осуществляется с помощью методов класса DArray).

Если перед выполнением цикла скопировать грани импортируемых данных в соответствующие теневые грани, то цикл можно выполнять без доступа к удаленным данным. Теневые грани задаются методом setShadow класса Shadow Максимальная ширина теневых граней может задаваться пользователем. По умолчанию максимальная ширина теневых граней по каждому измерению равна 1. Методом renewShadow можно уточнить размеры теневых граней (задать меньше, чем было задано при первоначальной установке). Перезапись в теневые грани (обновление) осуществляется перед параллельным циклом.

Если импортируемые переменные не являются “соседними” и для доступа к ним нельзя использовать теневые грани, то их буферизация осуществляется через отдельный буферный массив. Методом setBuffer класса DArray задается, какая часть массива должна быть буферизована на каждом процессоре. Размер этой части определяет размер буфера.

Совмещение счета и обменов данными между процессорами. Обновление значений теневых граней выполняется каждым процессором в два этапа: сначала запускаются операции обмена данными – операции посылки экспортируемых данных локальной секции массива и операции приема значений теневых граней; затем ожидается завершение выданных операций. На фоне этого ожидания можно выполнять вычисления по внутренним элементам секции массива, если этапы операции обновления теневых граней задавать с помощью отдельных директив (запуск операции и ожидание ее окончания).

Если перед циклом необходимо обновление теневых граней нескольких массивов, то эти операции можно объединить в одну групповую операцию обновления, что также может уменьшить накладные расходы. Организация группового асинхронного обновления теневых граней осуществляется методом groupShadow класса Shadow. Обновление теневых граней (посылка и прием экспортируемых данных) реализуется методом startShadow. Ожидание завершения обновления теневых граней осуществляет метод waitShadow.

Мы не приводим примера параллельной программы с использованием библиотеки *JavaDVM*, так как соответствующая программа представляет собой программу с рис. 2, записанную в объектно-ориентированном виде (это связано с тем, что модель *DVM* имеет много общего с моделью, лежащей в основе системы *HPF 2*).

4. Примеры параллельных программ. Анализ производительности.

Рассмотрим несколько примеров параллельных программ, разработанных для тестирования библиотек *DPJ* и *JavaDVM*. Тексты соответствующих программ, а также тексты исходных модулей библиотек *DPJ* и *JavaDVM* доступны через *Internet*, адрес сайта `www.ispras.ru/~dpj`.

Параллельная версия программы *QuickSort* в модели *DPJ*. В качестве примера использования распределенных контейнеров и их итераторов, определенных в модели *DPJ,* рассмотрим алгоритм параллельной сортировки массива целых чисел, основанный на последовательном алгоритме *QuickSort*. В алгоритме используется распределенное сбалансированное бинарное дерево с полностью заполненными уровнями.

В начальный момент исходный массив размещается на корневом узле дерева. Определяется множественный итератор, обходящий дерево по уровням, имеющий корневой узел в качестве начального значения. На множестве узлов, принадлежащих значению итератора, запускается параллельный алгоритм, который разбивает сортируемый массив на две части так, как это делается в последовательном алгоритме *QuickSort*. Затем левая часть массива отправляется левому поддереву текущего узла, а правая – правому. На следующей итерации новое значение итератора есть множество всех узлов, составляющих следующий уровень дерева. На этой итерации каждая из сыновних вершин нового уровня получает свою часть массива от родительского узла и производит соответствующие действия. Работа алгоритма продолжается до достижения итератором значения, содержащего множество всех листовых узлов дерева. На листовых узлах оставшийся массив сортируется последовательным алгоритмом `QuickSort`. Дальнейшее приращение значения итератора приводит к получению им специального значения `at-end`, и алгоритм завешает свою работу. Результатом работы алгоритма является отсортированный массив, распределенный по листовым узлам дерева. Алгоритм был запрограммирован и выполнен на сети из 8 процессоров *UltraSPARC* 167 МГц и 128 Мбайт оперативной памяти каждый.

Блочно-циклическое распределение. Блочно-циклическое распределение двумерного массива является одним из стандартных способов распределения массивов в системе программирования *HPF*. Оно характеризуется набором из четырёх чисел *P, Q, m* и *n*, где $P \times Q$ – решётка процессов, а $m \times n$ – размер блока исходной матрицы. Блочно-циклическое распределение фиксирует шаг, с которым назначаются строки и столбцы матрицы каждому процессорному узлу вычислительной сети.

Пусть дано *M* массивов (блоков исходной матрицы), проиндексированных целыми значениями *0, 1, ..., M-1*. В блочно-циклическом распределении глобальный индекс *m* представляет собой тройку чисел (*p, b, i*), где *p* - логический номер узла, *b* - номер блока находящегося на узле *p* и *i* индекс внутри блока *b*. Распределение блочной матрицы можно представить в виде совокупности двух отображений: первое это распределение строк матрицы между *P* процессорами, второе – распределение столбцов между *Q* процессорами (если решётка процессорных узлов имеет размеры $P \times Q$).

На рис. 6а представлен пример блочно-циклического распределения данных для двухмерного массива (матрицы), содержащего 11 строк и столбцов. Пронумерованные прямоугольники представляют собой блоки, на которые разделена матрица, а номер указывает на какой процессор помещён соответствующий блок – все блоки с одинаковыми номерами находятся на одном процессоре. Номера сверху и слева от матрицы представляют собой соответствующие индексы строк и столбцов блоков матрицы. На рис. 6b представлен другой способ распределения той же матрицы: на каждый из процессоров помещается 64 блока распределённой матрицы.

Программа решения системы алгебраических уравнений с помощью алгоритма Холесского.

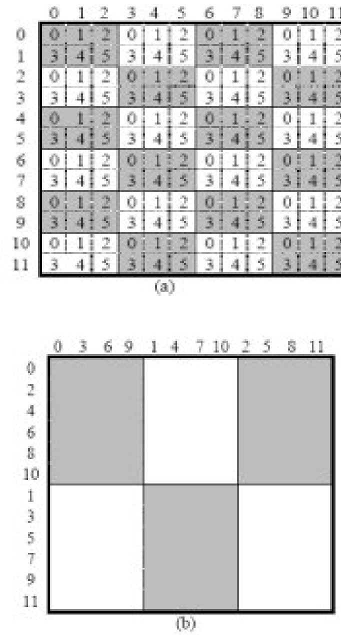


Рис. 6. Примеры блочно-циклического распределения данных

Коммуникационные характеристики алгоритма и баланс загрузки процессов существенно зависят от распределения элементов матрицы между узлами вычислительной сети. Для алгоритма Холесского хорошие результаты по быстродействию получаются при использовании блочно-циклического распределения.

Количество процессоров	Время (миллисекунды)	Ускорение
1	40224	1.00
2	30 488	1.32
4	22 744	1.77
6	18 883	2.13
8	15 995	2.51

Рис. 7. Результаты измерений быстродействия параллельной программы, реализующей разложение Холесского

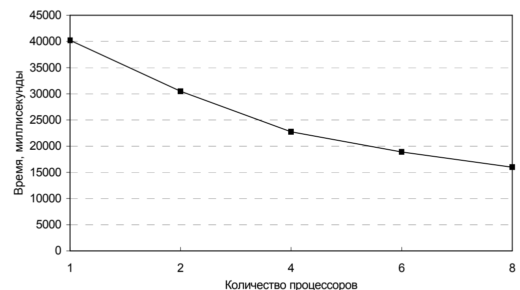


Рис. 8. Зависимость времени выполнения алгоритма Холесского от числа процессоров в сети

На рис. 7, 8 и 9 приводятся результаты измерений производительности параллельной версии алгоритма Холесского, который был реализован с помощью библиотеки DPJ. В программе использовалась матрица размера 240240 элементов, вычислительный комплекс состоял из кластера рабочих станций *UltraSparc-1* с процессором *UltraSPARC* 167 МГц и 128 Мбайт оперативной памяти каждая. Все станции соединены 100 Мбитной сетью *Ethernet.* Оборудование любезно предоставлено администрацией *IRISA/INRIA*. Как показывают результаты, представленные на рис. 7, 8 и 9, использование библиотеки DPJ обеспечивает ускорение выполнения параллельных *Java*-программ по сравнению с их последовательными версиями. Из графика, представленного на рис. 9, видно, что программа показывает удовлетворительные результаты по масштабируемости: при увеличении количества процессоров в сети ускорение программы возрастает пропорционально числу процессоров в сети.

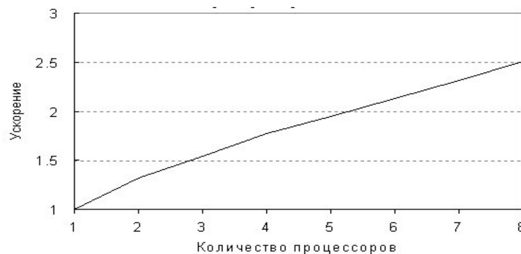


Рис. 9. Зависимость ускорения выполнения алгоритма Холесского для матрицы размером 240×240

Как показывают результаты, представленные на рис. 7, 8 и 9, использование библиотеки DPJ обеспечивает ускорение выполнения параллельных *Java*-программ по сравнению с их последовательными версиями. Из графика, представленного на рис. 9, видно, что программа показывает удовлетворительные результаты по масштабируемости: при увеличении количества процессоров в сети ускорение программы возрастает пропорционально числу процессоров в сети.

Как показывают результаты, представленные на рис. 7, 8 и 9, использование библиотеки DPJ обеспечивает ускорение выполнения параллельных *Java*-программ по сравнению с их последовательными версиями. Из графика, представленного на рис. 9, видно, что программа показывает удовлетворительные результаты по масштабируемости: при увеличении количества процессоров в сети ускорение программы возрастает пропорционально числу процессоров в сети.

5. Заключение

Параллельное программирование в модели *SPMD* не требует специальных языковых средств для описания параллельной программы, так как такие средства дают лишь одну дополнительную возможность – более простые системные способы описания распределения данных между процессорами вычислительной сети и, как следствие этого, автоматическую генерацию операторов обмена данными между процессорами сети. Как оказалось, эта возможность не привлекает прикладных программистов, так как она лишает их средств оптимизации приложений по обменам данными, т.е. не позволяет выразить наиболее важные аспекты многих прикладных параллельных алгоритмов. Прикладные программы, в которых применение системных способов распределения данных оправдано, составляют лишь узкий класс. В других приложениях удобны другие, менее общие, способы распределения данных. Следовательно, язык для описания параллельных вычислений должен допускать много различных способов распределения данных между процессорами. Наиболее просто это можно реализовать в объектно-ориентированном языке.

Конечно, требования высокого быстродействия вынуждают брать в качестве базового объектно-ориентированного языка такой язык как *C++*, позволяющий писать программы с высоким быстродействием, но, к сожалению, *C++* не является строго объектно-ориентированным языком, что затрудняет его использование в качестве базы для изучения особенностей объектно-ориентированного параллельного программирования. Поэтому в данной работе, которая имеет не прикладную, а исследовательскую направленность, в качестве базового языка был взят менее эффективный, но более строгий объектно-ориентированный язык *Java*. Библиотеки для языка *Java,* описанные в работе, легко переписываются на *C++*. Кроме того, близкие результаты для *C++* содержатся в работе [12]. Относительно этой работы интересно отметить, что в начале язык *Charm*, лежащий в ее основе, представлял собой параллельное расширение языка *C*, но популярность приобрел не *Charm*, а его объектно-ориентированная версия *Charm++*, которая, по существу представляет набор системных библиотек на *C++*.

Эксплуатация библиотек DPJ и JavaDVM показала их удобство для разработки прикладных параллельных программ. В настоящее время они используются для разработки прикладного пакета для расчета прочности оболочек из композитных материалов. Дальнейшие наши планы связаны с разработкой и реализацией инструментальных средств, поддерживающих отладку и мониторинг параллельных программ. Эти диалоговые средства позволят прикладному программисту улучшать характеристики своей параллельной программы.

Библиография

1. J. Gosling, “The Java Language Environment”, white paper, Sun Microsystems, Mountain View, Calif., 1995; http://java.sun.com
2. Ted G. Lewis, Foundations of Parallel Programming: A Machine Independent Approach, IEEE Computer Society Press, Los Alamitos, CA, 1993.

3. Nan's Parallel Computing Page http://www.cs.rit.edu/~ncs/parallel.html
4. HPF: High Performance Fortran Language Specification, High Performance Fortran Forum, version 2.0, January 31 1997; http://www.crpc.rice.edu/HPFF/hpf2/index.html
5. MPC++ Version 2:Massively Parallel, Message Passing, Meta-level Processing C++ http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html
6. N.A. Konovalov, V.A. Krukov, S.N. Mihailov and A.A. Pogrebtsov, “Fortran-DVM language for portable parallel programs development”, Proceedings of Software for Multiprocessors and Supercomputers: Theory, Practice, Experience (SMS-TPE 94), Inst. for System Programming RAS, Moscow, Sept. 1994.
7. MPI: Message Passing Interface Standard, Message Passing Interface Forum, June 12 1995; http://www.mcs.anl.gov/mpi/index.html
8. D. Wood, P. Welch OCCAM 2: Kent Retargetable Occam Compiler. http://www.idiom.com/free-compilers/LANG/OCCAM2-1.html
9. G.A. Geist, J.A. Kohl, P.M. Papadopoulos. PVM and MPI. A Comparison of Features. Calculateurs Paralleles Vol. 8 No. 2 (1996), http://www.epm.ornl.gov/pvm/pvm_home.html
10. Nicolau, David Gelernter, T. Gross, and D. Padua, editors. Advances in Languages and Compilers for Parallel Computing. The MIT Press, 1991.
11. Nicholas Carriero and David Gelernter. Linda and Message Passing: What Have We Learned? Technical Report 984, Yale University Department of Computer Science, Sept. 1993. http://www.cs.yale.edu/Linda/tech-reports.html
12. Dmitry Arapov, Alexey Kalinov, Alexey Lastovetsky, Ilya Ledovskih, and Ted Lewis, "A Programming Environment for Heterogeneous Distributed Memory Machines", Proceedings of 6th Heterogeneous Computing Workshop (HCW'97), IEEE Computer Society, Geneva, Switzerland, April 1997, pp.32-45
13. Standard Library for Parallel Programming, http://charm.cs.uiuc.edu/
14. ScaLAPACK Users' Guide, http://www.netlib.org/scalapack/slug/scalapack_slug.html
15. Susan F. Hummel, Ton Ngo, Harini Srinivasan, “SPMD Programming in Java”, tech. report, IBM T.J. Watson Research Center; http://www.npac.syr.edu/projects/javaforcse/cpande/IBM spmdjava_new.ps
16. V. Ivannikov, S. Gaissaryan, M. Domrachev, V. Etch, N. Shtaltovnaya. DPJ: Java class library for development of data-parallel programs. http://www.ispras.ru/~dpj