Оценка времени работы параллельной программы с помощью интерпретатора среды ParJava

В. А. Падарян

Аннотация. В работе рассматривается интерпретация параллельных программ в среде ParJava, позволяющая получать оценки времени их выполнения на инструментальном компьютере. Рассмотрена схема работы и некоторые особенности реализации интерпретатора, описано взаимодействие пользователя со средой при интерпретации программы, показано, как использовать оценки времени выполнения программы для определения границ ее области масштабируемости. Для трех модельных программ проведены численные эксперименты, которые позволили установить, что получаемые оценки времени работы достаточно точны (относительная погрешность не превышает 10%).

1. Введение

В работе рассматривается интерпретация параллельных программ в среде ParJava, позволяющая получать оценки времени их выполнения на инструментальном компьютере. Точность прогнозирования времени выполнения оценивается на примере трех модельных программ: сравнивается предсказанное время выполнения с действительным, полученным на кластерах ИСП РАН (использовались сети Myrinet и Ethernet) и НИВЦ МГУ (сеть SCI). Для построения моделей программ и их интерпретации использовался однопроцессорный инструментальный компьютер, оценка времени работы базовых блоков получалась на соответствующих целевых вычислительных системах в соответствии с методикой, описанной в [1].

Также в работу включены оценки производительности Java-программ полученные на основе сравнения времени работы C и Java версий одной из программ на кластерах ИСП РАН и НИВЦ МГУ. Использовались C-компиляторы gcc 3.2.2. (кластер ИСП РАН), icc 8.0 (компилятор фирмы Intel, кластер НИВЦ МГУ) и SUN JDK 1.4.2_03.

Работа состоит из пяти разделов. Во втором разделе описывается использование методики прогнозирования времени работы параллельной программы в рамках среды ParJava. В третьем разделе описывается реализация интерпретатора параллельных программ и особенности построения модели параллельной программы. В четвертом разделе представлены результаты проводившихся численных экспериментов: описаны использовавшиеся вычислительные кластеры и модельные программы, использовавшиеся для

демонстрации возможностей методики прогнозирования времени работы. Приведены сравнительные результаты предсказанного и действительного времени работы, а также производительности C и Java. Пятый раздел – заключение.

2. Оценка времени выполнения параллельной программы

Разрабатывая прикладную параллельную программу в рамках среды ParJava, пользователь (прикладной программист) взаимодействует с графическим интерфейсом среды. Пользователь создает проект (совокупность файлов исходного кода), добавляя в него либо уже существующие фалы, либо создавая новые с помощью текстового редактора среды. Помимо исходного кода в рамках проекта хранятся: байт-код, статические модели классов, модель параллельной программы, трассы и профили программы. Управление проектом осуществляется через диалоговое окно, в котором отображается список файлов исходного кода и расположен набор кнопок, позволяющих эффективно применять инструментальные средства среды.

После ввода программы пользователь выполняет трансляцию исходного кода во внутреннее представление (строятся статические модели классов): с помощью мыши отмечает транслируемые файлы в диалоговом окне управления проектом и вызывает транслятор. В результате трансляции во внутреннее представление для каждого указанного файла исходного кода (содержащего один или несколько классов) строится абстрактное синтаксическое дерево (АСД), которое затем передается преобразователю, строящему набор статических моделей классов – объекты класса `FileInfo`. Трансляция выполняется независимо: в случае изменения исходного кода отдельного класса Java-программы повторно транслируется только файл, содержащий данный класс. После трансляции исходного кода в окне управления проектом начинает отображаться информация о структуре класса: каждый файл исходного кода отображается как корень дерева, потомок которого – класс, описанный в данном файле, для каждого класса в виде потомков выводятся методы и поля данного класса. Внутреннее представление (объект класса `FileInfo`) сохраняется на диск с помощью сериализации в виде файла с расширением `.ast`.

Статические модели классов используются для решения следующих задач:

- определения времени работы параллельной программы,
- выявления участков критического кода,
- выявления зависимостей в телах циклов,
- оценки интенсивности коммуникаций.

В случае если пользователь запрашивает у среды оценку времени работы программы, среда выполняет следующие действия: строит модель параллельной программы, отражающую как статические, так и динамические свойства программы (рис. 1), и выполняет ее интерпретацию. Модель (объект класса `Model`) появляется в результате компоновки статических моделей

классов, составляющих программу и оценок времени работы базовых блоков программы, записанных в текстовом файле. Оценки времени работы базовых блоков позволяют дополнить модель параллельной программы динамическими свойствами.

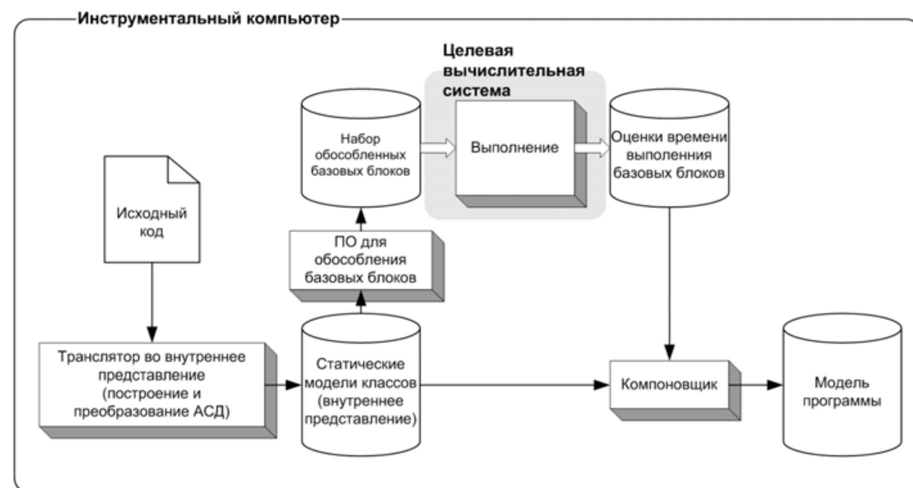


Рис. 1. Порядок построения модели параллельной программы.

По внутреннему представлению программы строится набор независимых служебных программ; результатом работы каждой такой программы является оценка времени выполнения одного из базовых блоков исходной программы. Полученные служебные программы оформляются в виде задания для системы управления кластером. Затем задание передается на целевую вычислительную систему, где оно выполняется, после чего результаты (полученные оценки, записанные в файле в виде текста определенного формата) забираются на инструментальную машину.

Статическая модель каждого класса и оценки времени выполнения его базовых блоков передаются компоновщику, который объединяет их в модель параллельной программы. Построенная таким образом модель передается затем интерпретатору, учитывающему свойства целевой вычислительной системы, для получения оценки времени работы путем интерпретации.

В процессе интерпретации (рис. 2) моделируется работа параллельной программы, использующей *P* процессоров вычислительной системы с распределенной памятью. При этом на каждом процессоре выполняется только один процесс параллельной программы. Каждый такой процесс представляется в интерпретаторе *логическим процессом*.

Логический процесс – это объект класса `LP`, являющегося расширением класса `Thread` языка Java. Каждый объект класса `LP` содержит модельные часы, очередь входящих сообщений, набор значений переменных (в виде стека выполняющихся методов) и объектную ссылку на текущую интерпретируемую

вершину модели. Модель представлена в памяти одним общим экземпляром для всех логических процессов. Пользователь получает информацию о статусе выполнения логических процессов и управляет интерпретацией с помощью соответствующих средств GUI.

Подсистема управления интерпретатора следит за работой логических процессов и передает информацию об их выполнении пользователю. Подсистема состоит из *монитора* и набора *наблюдателей*. Монитор обеспечивает пользователя информацией о состоянии интерпретируемых логических процессов и позволяет контролировать процесс интерпретации в системе (приостанавливать, возобновлять интерпретацию). Каждый из наблюдателей отслеживает интерпретацию логических процессов и при наступлении в параллельной программе определенного состояния информирует об этом через монитор пользователя. Как монитор, так и каждый из наблюдателей представляет собой нить, расширяющую класс `Thread` и реализующую интерфейс `Iobserver`.

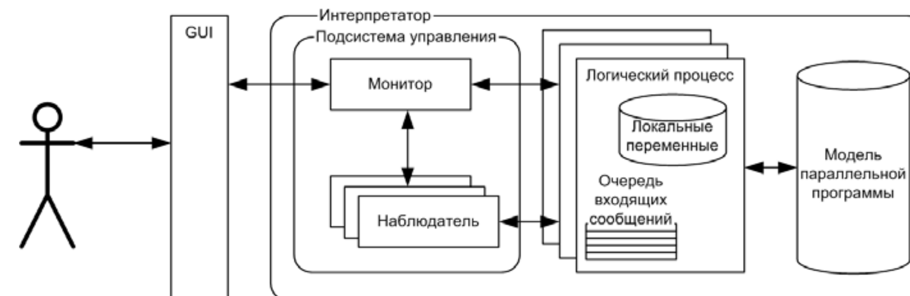


Рис. 2. Устройство интерпретатора.

В среде ParJava определена частичная интерпретация модели. Если интересующий пользователя фрагмент программы обладает рядом свойств [2], среда способна предоставить оценку его времени работы, выполнив интерпретацию только данного фрагмента, а не всей программы. После интерпретации фрагмента, он заменяется на вершину типа «редуцированный блок» (редуцируется). Частичная интерпретация позволяет сократить время затрачиваемое на изучение свойств программы, и за счет использования результатов интерпретации фрагментов ускорить интерпретацию всей программы.

Выполнение частичной интерпретации в среде ParJava протекает следующим образом. Пользователь (прикладной программист) задет *схему интерпретации* через GUI среды ParJava (рис. 3). Схема представляет собой последовательность внутренних вершин и методов модели, которые необходимо автономно интерпретировать и редуцировать. Выбор вершины осуществляется через контекстное меню в окне визуализации. Перечень вершин передается классу `Validator`, который проверяет условия корректности частичной интерпретации. В случае, если для какой-либо вершины условия не выполняются, пользователь получает сообщение об этом

в окне диагностики. Если в результате инспекции схемы все вершины, включенные в нее, удовлетворяют условиям корректности частичной интерпретации, схема и описание целевой вычислительной системы передаются интерпретатору. Интерпретатор, используя описание целевой вычислительной системы, последовательно редуцирует вершины модели, перечисленные в схеме.

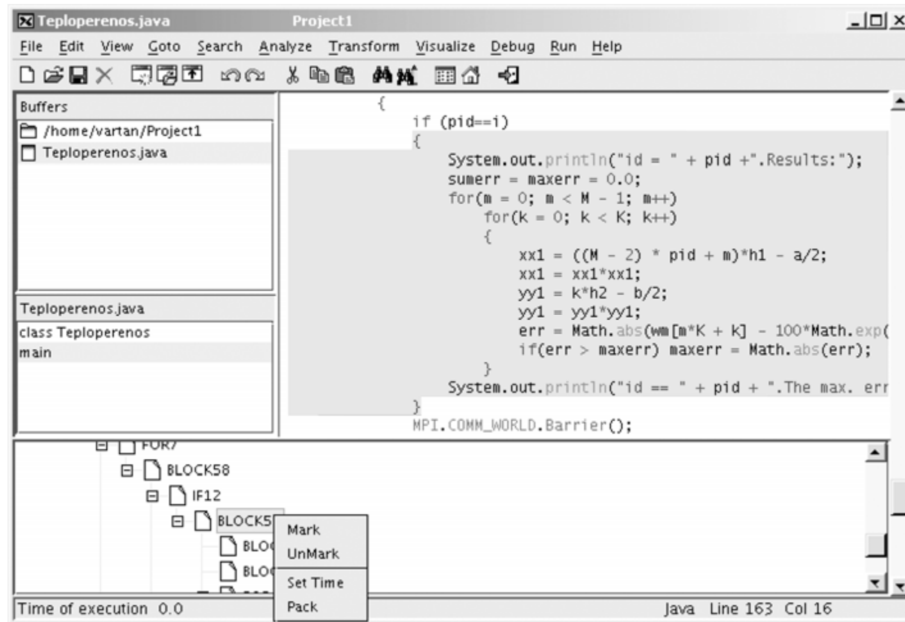


Рис. 3. Выбор вершины при проведении частичной интерпретации.

Если интерпретатор был вызван из контекстного меню вершины модели метода, результаты интерпретации (усредненная оценка времени работы) заносятся в вершину типа `ReducedBlock`, которая заменяет в модели интерпретированную вершину.

Если интерпретатор был вызван из основного меню среды ParJava, то он выполняет полную интерпретацию модели программы; результат интерпретации выводится в окне буфера результатов.

Непосредственно перед выполнением интерпретации вершины пользователь задает ряд параметров в диалоговом окне (рис. 4). На переднюю панель диалогового окна вынесены элементы управления основными параметрами: количество интерпретируемых процессов, пропускные способности сети и оперативной памяти при передаче сообщений, размер системного буфера, используемого в коммуникациях, (т.е. MTU), имя класса, использующегося в качестве коммуникационной модели, и др. На закладке «Output» расположены элементы управления выдачи диагностики: пользователь может потребовать

ведения журналов изменения значений переменных логических процессов, выполнения коммуникаций и др.

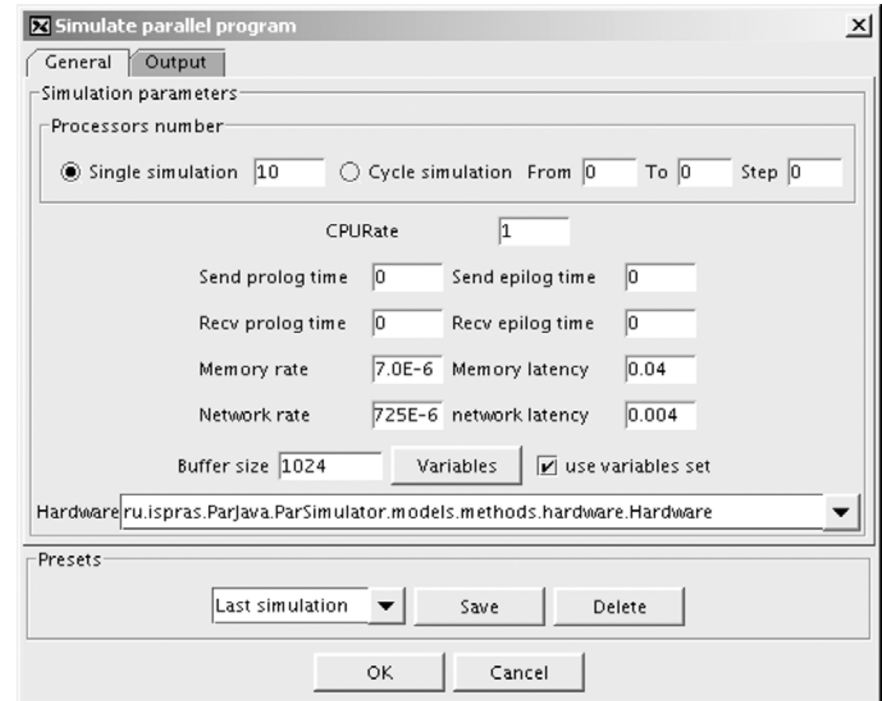


Рис. 4. Диалоговое окно запуска интерпретатора.

По требованию пользователя среда открывает окно монитора, в котором графи-чески отображается состояние интерпретируемых процессов параллельной программы. Помимо этого монитор позволяет пользователю через соответ-ствующие кнопки приостанавливать и возобновлять процесс интерпретации.

Графический интерфейс монитора в обычном состоянии показывает только текущий статус процессов (вычисления, коммуникация, простой), с помощью набора пронумерованных прямоугольников, занимающих основную часть окна (рис. 5). Цвет прямоугольника отражает действия, выполняемые данным процессом. Зеленый цвет соответствует выполнению вычислений, желтый – выполнению обмена данными, красный – простою процесса. Информация о состоянии процессов обновляется с задаваемой пользователем частотой.

Более подробную информацию о процессе можно получить, «нажав» мышью на соответствующий ему прямоугольник. После нажатия появляется новое окно, в котором выдаются данные о количестве переданных/принятых данных, времени, потраченном на вычисления и коммуникации, а также строка исходного кода, соответствующая текущей позиции выполнения процесса.

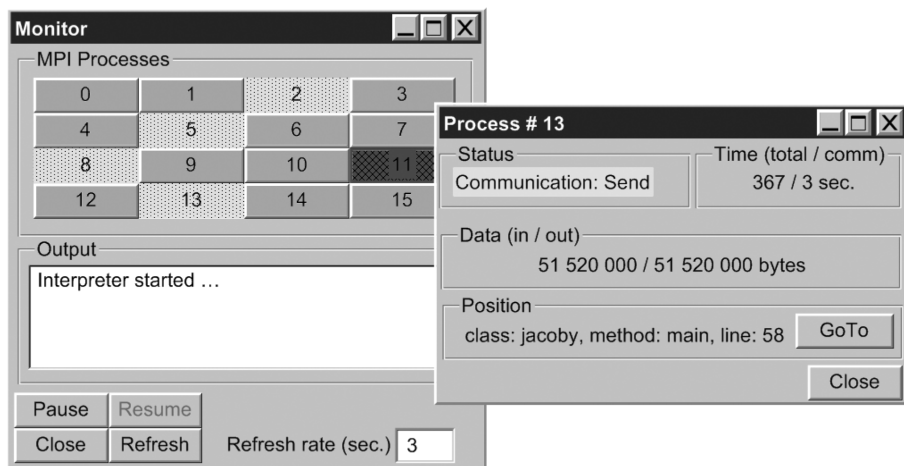


Рис. 5. Окна монитора интерпретатора.

Во время интерпретации по описанию целевой вычислительной системы порождается набор логических процессов, каждый из которых строит оценку времени работы процесса параллельной программы на целевой вычислительной системе, интерпретируя модель программы. Максимальная оценка принимается как оценка времени работы редуцируемой вершины в рамках всей параллельной программы. Если в качестве редуцируемой вершины выбрать корень одного из методов `main`, то будет интерпретирована вся программа. Схематически процесс интерпретации представлен на рис. 6.

Рис. 6. Схема выполнения интерпретации модели параллельной программы.

Интерпретатор при своей работе использует различные параметры целевой системы: мощность вычислительных узлов, латентность и пропускную

способность каналов передачи данных, длительность базовых операций MPI. Таким образом, располагая моделью программы и описанием целевой вычислительной системы, пользователь получает профиль требуемого уровня модели программы, используя только инструментальный компьютер.

3. Интерпретатор параллельной программы

Интерпретация параллельной программы выполняется интерпретатором ParSimulator на наборе синхронизирующихся между собой нитей (объектов класса `Thread`), что позволяет использовать для интерпретации как однопроцессорную, так и SMP-машину.

Интерпретация параллельной программы состоит в выполнении набора логических процессов (контейнер `LP[]`), вызывающих методы `simulate`, определенные в классах, реализующих интерфейс `IBlock`. Методы `simulate` используют службу синхронизации логических процессов, подсистему управления и коммуникационную модель.

Логический процесс (объект класса `LP`) имеет следующие поля:

- `double currentTime` (модельные часы),
- `Vertex currentV` (объектная ссылка на текущую вершину модели тела интерпретируемого метода),
- `StackContext context` (контекст: множество значений переменных интерпретируемой программы в рамках данного логического процесса),
- `CommModel commModel` (модель коммуникационного оборудования: набор системных переменных, описывающих свойства коммуникационного оборудования),
- `CompModel compModel` (модель вычислительного узла: набор системных переменных, описывающих свойства вычислительного узла),
- `int status` (Статус логического процесса. Принимает значения: «выполняется», «заблокирован», «исключительная ситуация», «успешно завершил работу»),
- `MsgContainer queue` (очередь входящих сообщений).

Набор переменных интерпретируемой программы (*контекст*), используемый в логическом процессе, локален для него – т.к. каждый процесс параллельной программы обладает своим адресным пространством. Набор этих переменных представляет собой стек таблиц, каждая из которых содержит все локальные переменные, объявленные в соответствующем методе. Значение каждой локальной переменной хранится в объекте соответствующего типа. Для атомарных типов используются классы-оболочки пакета `java.lang`: `Boolean`, `Byte`, `Short` и др. Доступ к локальным переменным осуществляется с помощью индекса. Организация доступа к локальным переменным, реализованная в интерпретаторе, аналогична работе с памятью в JVM, описанной в стандарте языка Java [3].

При вызове очередного метода в стек помещается новый элемент, и на него начинает ссылаться указатель текущего элемента в объекте `context`. При выходе из метода указатель перемещается на один элемент вниз, делая верхний элемент стека доступным для сборщика мусора.

В интерпретаторе поддерживается три вида контекстов (см. Таблицу 1):

1. ***Программный контекст*** кроме переменных, определенных в программе, содержит следующие системные переменные: номер моделируемого процесса (`//proc_num`) и общее количество процессов (`//size`).
2. ***Контекст коммуникационной модели*** содержит системные переменные, значения которых описывают параметры сети: скорость передачи данных, время задержки, время инициализации коммуникационных функций MPI и др.
3. ***Контекст модели вычислительного узла*** содержит параметры, описывающие характеристики узлов вычислительной системы. В настоящей реализации этот контекст содержит две системные переменные: `//CPURate`, описывающую *отношение производительностей* моделируемого и эталонного процессоров, и `//memcpy`, описывающую время копирования одного байта данных в памяти вычислительного узла.

Если оценки времени выполнения базовых блоков программы были получены заранее на целевой вычислительной системе и требуется только получить оценку времени работы программы, то значение `//CPURate` устанавливается равным единице.

Переменная `//CPURate` используется для определения изменения времени выполнения исследуемой параллельной программы при изменении производительности процессоров распределенной вычислительной системы (кластера).

Использование переменной `//CPURate` для пересчета оценок времени работы базовых блоков, полученных на инструментальном (или каком-либо другом, но не целевом) компьютере, нецелесообразно, т.к. приводит к большим погрешностям в оценках (порядка 15%).

Контексты инициализируются по описанию целевой вычислительной системы, передаваемой в ParSimulator при его запуске (рис. 3). Попадая в контекст, параметр системы представляется системной переменной. В таблице 1 приводится список используемых системных переменных.

Интерпретатор работает следующим образом. Перед началом интерпретации все используемые данные (модель интерпретируемой программы, описание целевой вычислительной системы, схема интерпретации) загружаются с помощью набора классов-фабрик, которые обеспечивают загрузку модели параллельной программы с диска, обработку параметров интерпретации и размещение их в контекстах, размножение контекстов по числу логических процессов и инициализацию системных переменных, значения которых зависят от номера логического процесса.

Монитор процессов порождает необходимое количество логических процессов и запускает их. Завершивший работу логический процесс информирует об этом монитор, передавая ему результат: свое заключительное состояние и значения атрибутов корня дерева, которое он интерпретировал. Когда все логические процессы завершатся, монитор передает полученные результаты пользователю.

Каждый логический процесс интерпретирует указанную ему вершину модели тела метода согласно правилам интерпретации [1]. Вершина, с которой начинается интерпретация, указывается через параметры запуска ParSimulator с помощью пары <имя метода, идентификатор вершины>. Имя метода включает в себя также имя класса, которому метод принадлежит. Для интерпретации всей программы пара должна представлять собой `<имя_класса.main, root>`. Интерпретация вершины происходит с помощью вызова на объекте, представляющем эту вершину метода `simulate`, определенного в интерфейсе `IBlock` (рис. 7).

Имя переменной	Описание
`//proc_num`	Номер процесса.
`//size`	Количество процессов интерпретируемой параллельной программы.
`//init`	Длительность операции `Init` (мс.)
`//free`	Длительность операции `Free` (мс.)
`//MakeThread`	Время, необходимое для порождения нити, в которой будет выполняться подсистема поддержки времени выполнения (мс.).
`//network_transfer_rate`	Время передачи одного байта данных по сети (мс.).
`//network_latency`	Время передачи по сети сообщения нулевой длины (мс.).
`//memory_transfer_rate`	Время передачи одного байта данных через разделяемую память (мс.).
`//memory_latency`	Время передачи сообщения нулевой длины через разделяемую память (мс.).
`//CPUrate`	Отношение производительностей моделируемого и эталонного процессоров.
`//memcpy`	Время копирования одного байта данных в рамках оперативной памяти процесса (мс.).

Таб. 1. Служебные переменные, описывающие вычислительную систему, и используемые средства обмена сообщения между процессами.

```
public interface IBlock {
   public Vector simulate(IProcess proc)
                            throws SimulationException;
   public int getId();
}
```

Рис. 7. Исходный код интерфейса `IBlock`.

Посредством параметра `proc` вершина получает данные о состоянии логического процесса, в частности, о его контексте (рис. 8).

Если интерпретируется вершина типа `Root`, то метод `simulate` заключается в вызове метода `simulate` его потомка и последующем вызове метода `setStatus` на объекте `proc` со значением «успешно завершил работу».

В случае, если интерпретируется внутренняя вершина модели, т.е. она является представителем одного из классов, расширяющих класс `InnerVertex`, метод `simulate` переопределяется таким образом, что интерпретация вершины сводится к интерпретации ее потомков [1].

```
public interface IProcess extends ISubject {
   public ICommModel getCommModel();
   public IContext getContext();
   public int getID();
   public IMachineModel getTargetMachine();
   public double getTimeLine();
   public void setTimeLine(double t);
}
```

Рис. 8. Исходный код интерфейса `IProcess`.

Листовые вершины соответствуют объектам класса `BasicBlock` (вычислительный блок) или классов, его расширяющих (остальные типы базовых блоков). Интерпретация объектов класса `BasicBlock` (вычислительный блок) выполняется следующим образом. Сначала происходит выполнение последовательности байт кода P , в нем содержащейся. Для этого вызывается статический метод вспомогательного класса `ProgramBlocks`, имя которого формируется по следующему правилу:

`bb<имя файла исходного кода>$<идентификатор базового блока>`, –

тело указанного метода представляет собой байт код интерпретируемого базового блока (последовательность P). Переменные из множества I являются формальными параметрами метода, и при его вызове значения соответствующих переменных из контекста используются в качестве фактических параметров. Возвращаемое значение метода – объект типа `Vector`, содержащий значения переменных из множества O данного базового

блока. Затем оценка времени работы базового блока (значение поля `elapsedTime`) добавляется к текущим показаниям модельных часов логического процесса.

В классах, расширяющих базовый класс `BasicBlock`, метод `simulate` переопределяется.

При интерпретации базового блока, представляющего вызов пользовательского метода (класс `UserMethodBlock`), по списку описаний локальных переменных и формальных параметров создается таблица (`frame`) для хранения значений переменных вызываемого метода и помещается в контекст логического процесса. При создании таблицы `frame`, в ее поля, соответствующие формальным параметрам, записываются значения соответствующих фактических параметров. Если фактический параметр содержит выражение, то в модели ему соответствует вычислительный базовый блок, который интерпретируется до вызова метода. Затем на корневой вершине тела вызываемого метода вызывается метод `simulate`. После окончания интерпретации корня, если вызванный метод имеет возвращаемое значение, это значение, взятое из таблицы `frame`, записывается в соответствующую переменную из множества O интерпретируемого базового блока. После этого контекст освобождается от таблицы `frame`.

Интерпретация базового блока, представляющего вызов служебной функции пакета MPI (класс `MPIServiceBlock`), сводится к записи в переменную из множества O данного базового блока значения системной переменной, которая специфицирована в одном из классов, расширяющих класс `MPIServiceBlock` (например, `MPIRankBlock` специфицирует переменную `//proc_num`), экземпляром расширяющего класса и является данный базовый блок.

Интерпретация базового блока, представляющего вызов коммуникационной функции, выполняется согласно разработанной модели в рамках которой коммуникационные функции рассматриваются как композиция базовых операций обмена. Определены следующие базовые операции.

`Init` – выделение ресурсов под служебные структуры данных, используемые в коммуникации, и их инициализация.

`Free` – освобождение ресурсов, захваченных во время инициализации.

`Pack(n, type)` – преобразования, выполняемые MPI над массивом данных типа *type*, общей размерности *n* байт. Имеется в виду преобразование представления данных, упомянутое в стандарте, – перевод чисел с плавающей точкой в машинно-независимый формат или перекодировка данных типа `MPI_CHAR` (*маршалинг данных*).

`Unpack(n, type)` – действие, обратное *`Pack`*, выполняется при приеме данных.

Post(n) – отправка буфера памяти длиной *n* байт другому процессу: буфер передается в абстрактный канал, после чего MPI перестает контролировать процесс отправки.

Get(n) – действие обратное Post. Выполняется принимающим узлом.

PostSM(serv_msg) – отправка другому процессу служебного сообщения. Данное действие может быть сведено к предыдущему: PostSM = Post(k), где *k* – размер служебного сообщения.

GetSM(serv_msg) – прием служебного сообщения.

Process(serv_msg) – обработка служебного сообщения.

Copy(n) – копирование буфера размерности *n* байт в памяти вычислительного узла.

MakeThread – создание потока выполнения в программе с помощью средств пользователя или операционной системы.

Wait(событие) – ожидание события. Под событием понимается прибытие на вычислительный узел сообщения.

Для обозначения длительности операции будем в дальнейшем использовать *Time(**op**)*, где ***op*** – рассматриваемая операция.

Длительность операций Init, Free, Pack, Unpack, Process, Copy, MakeThread определяется для пакета MPI установленного на кластере и по характеристикам вычислительного узла. Длительность операций Post, Get, PostSM, GetSM определяется согласно модели LogGP:

$$Time(\text{Post(n)}) = b*n + d_P(n),\ d_P(0) = l$$
$$Time(\text{Get(n)}) = b*n + d_G(n),\ d_G(0) = l,$$

где $d_P(n)$ и $d_G(n)$ – функции накладных расходов отправки и приема сообщения соответственно, b – пропускная способность сети, l – латентность сети.

Длительность операции Wait определяется по показаниям модельных часов логических процессов участвующих в коммуникации.

Правила интерпретации коммуникационных функций реализованы в методах simulate соответствующих классов. Эти методы используют одну из коммуникационных моделей, реализованных в системе. Каждая коммуникационная модель определяет правила вычисления функции *Time(*Post(n)*)*, используя длину *системного буфера*, правила распределения процессов по вычислительным узлам и характеристики коммуникационного оборудования. В системе реализованы две коммуникационные модели: в первой модели вычислительные узлы предполагаются однопроцессорными (класс Hardware), а во второй рассматриваются SMP-узлы (класс SMP). В обоих случаях вычисление *Time(*Post(n)*)* базируется на модели LogGP.

В обеих моделях учитывается фрагментирование пользовательского сообщения при его отправке в случае конечного размера *системного буфера* (MTU). Размер MTU определен для каждого конкретного коммуникационного

оборудования (например, для сети Ethernet размер MTU варьируется в приделах 64 – 1536 байт, для сети Myrinet MTU по умолчанию равен 9 Кб) и может быть получен из конфигурационных файлов операционной системы. В рамках одной операции Post передается не более чем MTU байт данных. По умолчанию значение MTU считается неограниченным. В случае конечного размера системного буфера и размера сообщения *n*, превышающего длину системного буфера, общее время *Time(*Post(n)*)* вычисляется по формуле

$$Time(\text{Post(n)}) = \lfloor n / MTU \rfloor Time(\text{Post(MTU)}) + Time(\text{Post}(n \bmod MTU)).$$

В программной реализации коммуникационная модель должна быть представлена классом, реализующим интерфейс IHardwareDescriptor. В состав интерфейса входит метод getTransmitTime, возвращающий время, требуемое для передачи данных из одного процесса в другой, оценивая тем самым *Time(*Post(n)*)*. Описанные выше модели включены в пакет ParSimulator.models.methods.hardware. Как было отмечено выше, класс SMP реализует модель, в рамках которой каждый вычислительный узел обладает *p* процессорами (*p* задается в описании вычислительной системы). При обращении к методу getTravellingTime объекта этого класса определяется расположение процессов на узлах вычислительной сети, и если они расположены на одном узле, то при вычислении *Time(*Post(n)*)* используются параметры, определенные для коммуникаций через разделяемую память. Пользователь может использовать собственную реализацию интерфейса IHardwareDescriptor, указав ее перед запуском интерпретатора. Интерпретатор динамически (используя API пакета java.ref) загружает данный класс, порождает его представителя и передает объектную ссылку на него в логические процессы в качестве используемой коммуникационной модели.

Завершается интерпретация коммуникационной функции добавлением времени выполнения коммуникации к текущему показанию модельных часов логического процесса.

Служба синхронизации используется при выполнении коммуникаций для обмена показаниями модельных часов (через временные метки сообщений), блокировки выполнения логических процессов и непосредственно передачи пользовательских данных, составляющих сообщение, между логическими процессами. Служебные данные, помещаются в описатель сообщения, далее именуемый *заголовком*. Заголовок содержит следующие параметры: идентификаторы процессов отправителя и приемника, тэг, расчетное время прибытия.

Каждый логический процесс обладает собственной очередью входящих сообщений. Каждая базовая коммуникационная операция Post (PostSM) добавляет в очередь принимающего процесса сообщение, заголовок которого сформирован должным образом. Добавление сообщения выставляет флаг Unread наличия необработанного сообщения. При приеме сообщения от явно указанного процесса сначала просматривается очередь входящих сообщений, и

при наличии соответствующего сообщения (от требуемого процесса) выполняется интерпретация. В противном случае процесс блокируется до тех пор, пока не придет какое-либо новое сообщение, что вызывает очередной просмотр очереди. Так происходит, пока логический процесс не примет соответствующее сообщение. После каждого просмотра очереди сообщений флаг `Unread` сбрасывается. Располагая входящим сообщением, локальный процесс выбирает соответствующий сценарий интерпретации функции приема. Последующие обмены сообщениями (уже не требующие выбора сценария, но влияющие на оценку длительности операции `Wait`) выполняются аналогично. Принятые сообщения удаляются из очереди.

Выполнение приема сообщения от процесса `MPI_ANY_SRC` требует блокировки выполнения логического процесса до тех пор, пока очередь входных сообщений не будет гарантированно содержать требуемое сообщение. Для этого необходимо, чтобы $\min_{m \in M} TM(m) < \min_{i=0,\dots,r-1,r+1,\dots,P} T_i$, где M – множество входящих сообщений принимающего процесса r, *(P + 1)* – количество моделируемых процессов, T_i – текущие показания модельных часов логического процесса *i*.

При возникновении исключительной ситуации логический процесс выставляет свое состояние в «исключительная ситуация» и информирует об этом монитор, передавая ему значение своего состояния и имеющуюся диагностическую информацию. После чего интерпретация в логическом процессе приостанавливается до принятия монитором решения, следует ли завершить работу интерпретатора или возобновить работу логического процесса. Помимо монитора, за логическими процессами могут следить и другие объекты – *наблюдатели*, анализирующие состояния логических процессов и вызывающие в них исключительные ситуации. Совокупность объектов-наблюдателей и монитор формируют систему управления интерпретатора. Наблюдатель выявляет ошибки работы параллельной программы в целом, тогда как монитор отслеживает только локальные для логического процесса ошибки. Пример наблюдателя – детектор коммуникационных конфликтов, который следит за блокировками в коммуникациях. Детектор конфликтов способен выявлять ошибки следующих типов: ожидание сообщения от «мертвого» узла, взаимное ожидание сообщений (для произвольного числа ждущих процессов). Для разре-шения этой задачи используется методика, близкая к описываемой в работе [4].

Интерпретация программы с целью прогнозирования времени ее работы использует оценки времени работы базовых блоков. Методика профилирования свойств базовых блоков за счет добавления в программу инструментальных операторов не позволяет собрать временной профиль, т.к. время работы базовых блоков много меньше точности системных часов. Для решения задачи оценки времени работы базового блока была предложена следующая методика. Класс `BBSeparator`, используя статическую модель класса, генерирует исходный код на языке Java: набор *инструментальных программ*.

Каждая созданная инструментальная программа представляет собой класс с именем `BB<id>`, помещенный в пакет `auxbb`. `id` является целочисленным идентификатором, инструментальные программы получают сквозную нумерацию, начиная с нуля, соответствие номеров инструментальных программ исходной программе сохраняется в служебном файле `map` в виде строк:

<имя класса>,<id базового блока>:<id инструментальной программы>

Каждая инструментальная программа содержит только метод `main`, в теле которого, согласно описанной методике, расположены пустой и вспомогательный циклы. Помимо инструментальных программ в пакет `auxbb` помещен класс `DistrLauch`, также содержащий только метод `main`. `DistrLauch` представляет собой параллельную Java-программу, ее выполнение заключается в вызове метода `main` классов `BBid`, где *id* удовлетворяет условию $Q \cdot r / S \le id < Q \cdot (r+1) / S$, где Q – количество созданных инструментальных программ, r – порядковый номер процесса, S – количество процессов. Т.е. класс `DistrLauch` обеспечивает распределенное выполнение инструментальных программ на параллельной вычислительной системе. Помимо этого класс `BBSeparator` обеспечивает распределенное выполнение инструментальных программ в виде PBS- задания. PBS [5] – система пакетной обработки заданий, установленная и успешно работающая на ≈40% кластеров 21-ой редакции списка Top500 [6]. Класс `BBSeparator` создает шаблон PBS- задания, в который пользователю достаточно внести только количество запрашиваемых вычислительных ресурсов кластера (вычислительные узлы, процессоры) и имя сервера очередей PBS.

Результаты работы инструментальных программ сохраняются в виде набора текстовых файлов `BBestim.r`, где *r* – номер процесса программы `DistrLauch`. Каждый файл содержит строки вида

< id инструментальной программы >:<оценка времени работы>

Количество итераций вспомогательного и пустого циклов, выполняемых в инструментальной программе, передается через параметры ее метода `main`.

Необходимость выполнять базовый блок отдельно от исходной программы требует воссоздания контекста, в рамках которого исходный код базового блока способен компилироваться и выполняться без возникновения исключительных ситуаций. Воссоздание контекста заключается в объявлении всех переменных, фигурирующих в базовом блоке, и инициализации переменных, значения которых используются.

Для объявления переменных используется их описание (тип и имя), присутствующее в статической модели класса.

Инициализация переменных примитивных типов и строк происходит путем присваивания им значений при объявлении. Присвоение значений происходит таким образом, чтобы избежать деления на ноль при выполнении арифметических операций. Для анализа выражений используются ссылки на соответствующие вершины исходного АСД, содержащиеся в объекте типа

`BasicBlock`. В данной реализации отслеживается только одна ситуация – когда делитель является переменной из I, переменная гарантированно не будет инициализирована нулем.

4. Результаты численных экспериментов

Три тестовые программы (бенчмарки) выполнялись на кластерах ИСП РАН и НИВЦ МГУ.

Высокопроизводительный кластер ИСП РАН, состоит из восьми узлов, связанных сетями Myrinet и Fast Ethernet. Каждый вычислительный узел работает под управлением операционной системы Linux с ядром версии 2.4.20-8, содержит два процессора Athlon с тактовой частотой 1533 MHz и один гигабайт оперативной памяти. Для передачи данных во время работы параллельной программы используется сеть Myrinet, обеспечивающая полнодуплексный канал с пропускной способностью 2 Gbit/sec; топология сети Myrinet представляет собой сеть Клоса (Clos network). Сеть Fast Ethernet используется только для управления работой кластера: монтирование файловых систем, запуск процессов параллельных программ, сбор данных системного мониторинга вычислительных узлов.

Высокопроизводительный кластер Leo НИВЦ МГУ, состоит из шестнадцати узлов, связанных сетями SCI и Fast Ethernet. Каждый вычислительный узел работает под управлением операционной системы RedHat Linux 7.3, SSP 3.1, содержит два процессора Xeon с тактовой частотой 2.6 ГГц и два гигабайта оперативной памяти. Для передачи данных используется сеть SCI (сетевые адаптеры D335 на вычислительных узлах) с пиковой пропускной способностью 3.2 Gbit/sec, топология сети – двумерный тор. Управляющая сеть – Fast Ethernet (onboard-адаптеры, коммутатор 3Com, гигабитный канал к файл-серверу), сервисная сеть "СКИФ-ServNet" (вкл/выкл/reset узлов, сериальная консоль).

Также в работе упоминаются кластеры НИВЦ МГУ SCI и SKY.

Высокопроизводительный кластер SCI НИВЦ МГУ, состоит из шестнадцати узлов, связанных сетями SCI и Fast Ethernet. Каждый вычислительный узел работает под управлением операционной системы RedHat Linux 7.3, SSP 2.1, содержит два процессора Pentium III с тактовой частотой 500 МГц и один гигабайт оперативной памяти. Для передачи данных используется сеть SCI, управляющая сеть – Fast Ethernet.

Кластер (вычислительная ферма) SKY состоял из двадцати вычислительных узлов, связанных сетью Fast Ethernet. Каждый вычислительный узел работал под управлением операционной системы RedHat Linux 7.2, и был оснащен двумя процессорами Pentium III с тактовой частотой 850 МГц и одним гигабайтом оперативной памяти. На данный момент кластер SKY является частью кластера AQUA.

Среда ParJava (построение и интерпретация моделей параллельных программ) работала на инструментальном компьютере. Оценки времени работы базовых блоков получались на соответствующих целевых вычислительных системах. В

качестве инструментального компьютера использовался персональный компьютер с одним процессором Intel Pentium IV 2,66 GHz и 512 Mb оперативной памяти.

4.1. Предсказание времени работы программы

При моделировании коммуникаций кластера ИСП РАН использовались следующие параметры: Time(`Prolog`) = Time(`Epilog`) = Time(`Process`) = Time(`Marshal`) = Time(`Unmarshal`) = Time(`MakeThread`) = `0`. (в используемой реализации MPI операции `Marshal` и `Unmarshal` не выполнялись), $k = 4$, β=0.5, b = 200 Mb/sec и l = 30 usec для сети Myrinet, b = 10 Mb/sec и l = 160 usec для сети Ethernet.

Для описания SCI-кластера использовались аналогичные значения параметров, за исключением: b = 300 Mb/sec, l = 4 usec (взяты пиковые значения характеристик сети заявленные производителем).

Время выполнения базовых блоков оценивалось непосредственным выполнением вспомогательных программ (см. раздел 5) на кластере для которого строился прогноз.

Все особенности и возможные изменения значений параметров вычислительных систем описываются непосредственно при рассмотрении конкретного примера.

Программа «Теплоперенос». Программа «Теплоперенос» представляет собой реализацию параллельного алгоритма решения задачи о распространении тепла в однородном стержне. В задаче рассматривается линейное уравнение теплопроводности:

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right)+F(x,t)=c\rho\frac{\partial u}{\partial t},$$

где u=u(x,t) и x – точка n-мерного пространства. В двумерной квадратной области `[0, 1]×[0, 1]` уравнение принимает вид

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial u}{\partial y}\right)+F(x,y,t)=c\rho\frac{\partial u}{\partial t},$$

$$u(x,y,t),\quad 0<x<1, 0<y<1, 0<t.$$

Начальные и граничные условия задаются соответственно следующим образом:

$$u\big|_{t=0}=e^{-(x_1+y_1)},\quad 0\le x\le 1, 0\le y\le 1,$$

$$u\big|_{x=0}=u\big|_{x=1}=100e^{-(x_1+y_1)},\ 0\le y\le 1, 0\le t,$$

$$u\big|_{y=0}=u\big|_{y=1}=100e^{-(x_1+y_1)},\ 0\le x\le 1, 0\le t,$$

$$x_1=(x-\tfrac{1}{2})^2,\ y_1=(y-\tfrac{1}{2})^2.$$

Распараллеливание происходит за счет распределения узлов сетки между разными процессами: матрица «нарезается» на прямоугольники вдоль оси y. В рамках каждой итерации происходят обмены теневыми гранями с соседними процессами. При увеличении количества процессов параллельной программы, увеличивается только одна размерность общей матрицы. Программа имеет особенность: i-ый процесс в рамках j-ой итерации принимает теневые грани от (i+1)-го процесса в котором выполняется (j-1)-ая итерация. Таким образом, при увеличении количества процессов на единицу, увеличивается на единицу количество итераций алгоритма.

Программа выполнялась на узлах кластера, занимая оба процессора каждого вычислительного узла. Каждый процесс обрабатывал локальную часть матрицы размером 3000x3000 элементов. Программа выполняла 1000 итераций внешнего последовательного цикла.

Для определения фактической пропускной способности сети на данной задаче на кластере ИСП РАН был выполнен бенчмарк Ping-Pong, меряющий время передачи данных на уровне Java MPI (Round-Trip Time). Фактическая пропускная способность составила 77Mb/sec на сети Myrinet и 20Mb/sec на сети Ethernet. Такие значения фактической пропускной способности сети объясняются тем, что процессы программы при коммуникациях образуют «линейку», обмен данными с одним соседом происходит через разделяемую память, с другим через сеть. Такой механизм обменов в случае сети Myrinet дает среднюю пропускную способность в рамках всей программы хуже пиковой пропускной способности сети, а в случае сети Ethernet наоборот, пропускную способность в рамках программы улучшает. Время затрачиваемое на коммуникации, составило ~ 1 сек на сети Myrinet и ~ 4 сек на сети Ethernet.

Время работы программы, как замеренное (черная линия), так и предсказанные (серые линии), представлены на рис. 9 – 11.

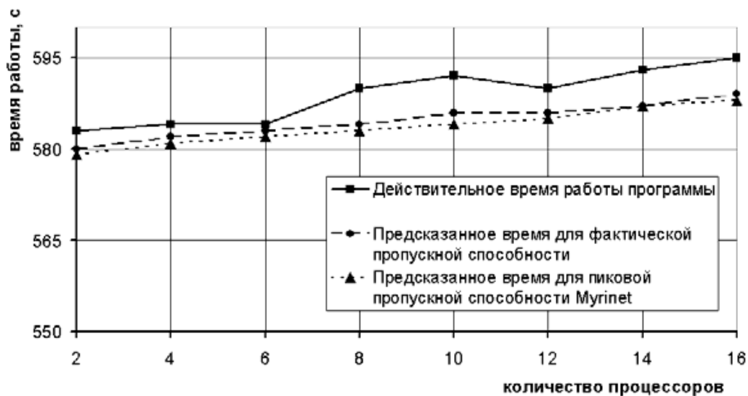


Рис. 9. Сравнение предсказанного и действительного времени работы программы «Теплоперенос» с пересылкой данных через сеть Myrinet. MPI: mpich-gm-1.2.5-10.

Относительная погрешность предсказания не превышает 8% на кластере НИВЦ МГУ, 1,5% и 3% на кластере ИСП РАН для сетей Myrinet и Ethernet соответственно. Менее заметное возрастание времени работы программы в случае выполнения коммуникаций через сеть Myrinet можно объяснить особенностью топологии сети. Этап коммуникаций в рамках каждого витка последовательного цикла состоит в попарном обмене теневыми гранями между соседними процессами.

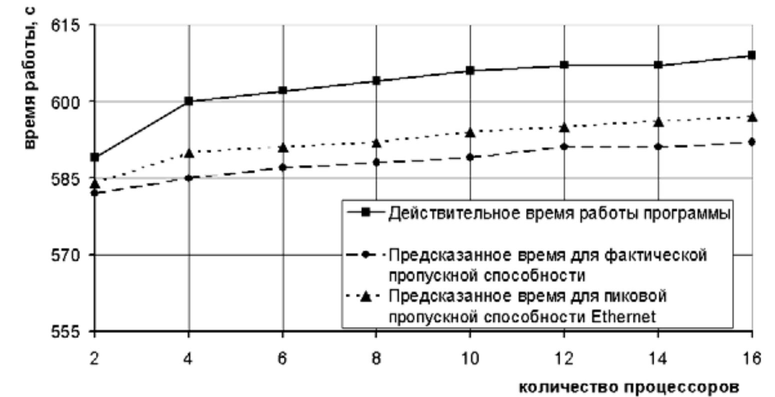


Рис. 10. Сравнение предсказанного и действительного времени работы программы «Теплоперенос» с пересылкой данных через сеть Ethernet. MPI: lam-7.1.1.

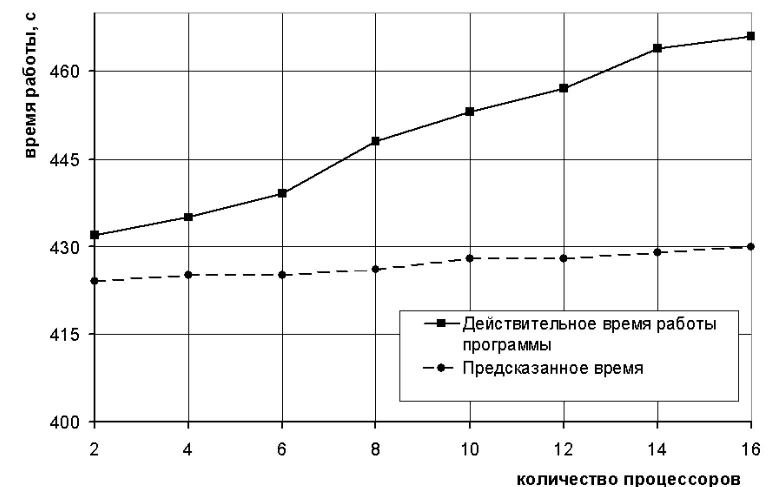


Рис. 11. Сравнение предсказанного и действительного времени работы программы «Теплоперенос» с пересылкой данных через сеть SCI. MPI: ScaMPI.

Топология сети Myrinet обеспечивает отсутствие блокировок процессов во время коммуникации за счет наличия физического канала при любом разбиении процессов на пары. В рамках данной программы это приводит к тому, что на сети Myrinet время коммуникаций не увеличивается при увеличении количества процессов параллельной программы.

Программа «Якоби». Второй пример представляет собой программу, решающую уравнение Лапласа итеративным методом Якоби. Программа состоит из основного последовательного цикла, внутрь которого вложены параллельные циклы, вычисляющие значения элементов локальной подматрицы в рамках очередной итерации. После выполнения параллельных циклов выполняются коммуникации: процессы обмениваются теневыми гранями. Программа выполняла 1000 итераций, обрабатывая матрицу 7300x7300 (80% загрузка памяти на кластере ИСП РАН).

Прогнозирование времени работы программы дало следующие результаты (рис. 12 – 14).

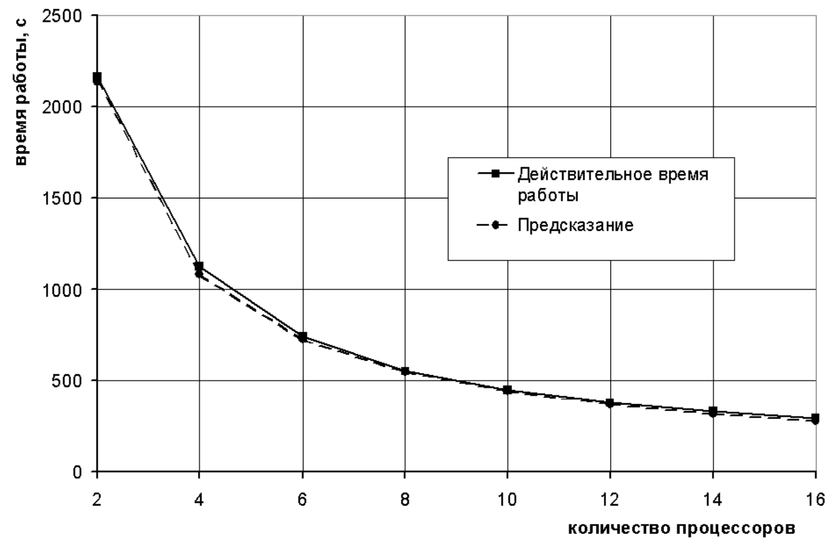


Рис. 12. Сравнение предсказанного и действительного времени работы программы «Якоби» с пересылкой данных через сеть Myrinet. MPI: mpich-gm-1.2.5-10.

При моделировании использовались те же значения характеристик коммуникационный сетей, что и в предыдущих примерах, за исключением того, что прогнозирование времени работы в случае сети Ethernet делалось для $b = 10$ и 5 Мб/сек.

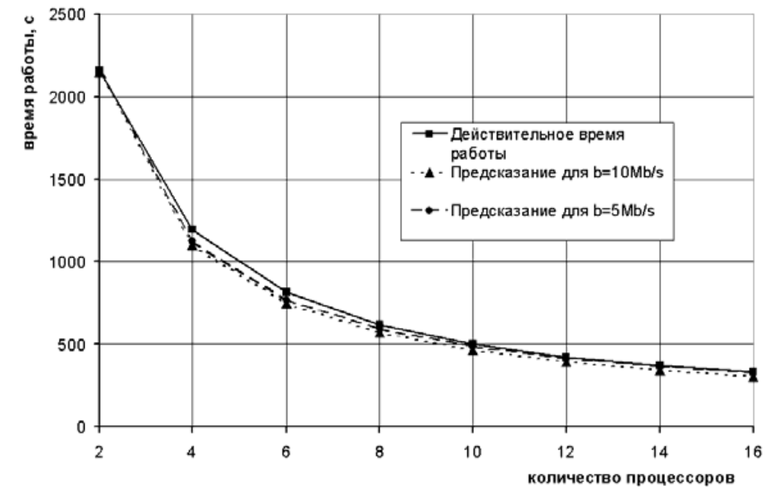


Рис. 13. Сравнение предсказанного и действительного времени работы программы «Якоби» с пересылкой данных через сеть Ethernet. MPI: lam-7.1.1.

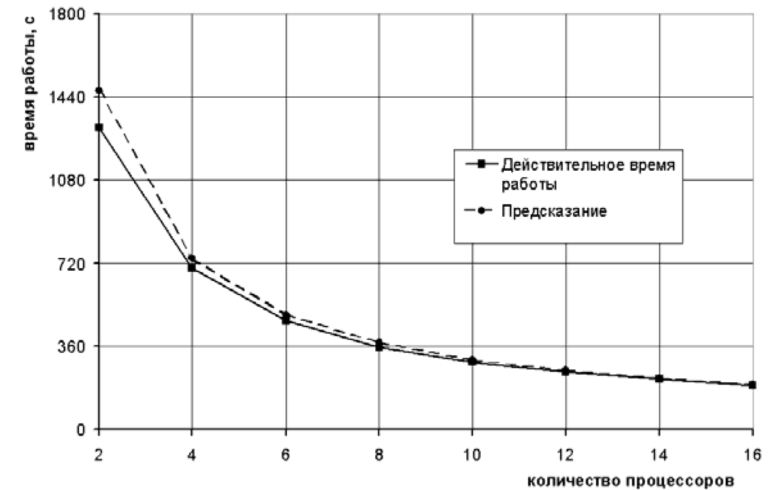


Рис. 14. Сравнение предсказанного и действительного времени работы программы «Якоби» с пересылкой данных через сеть SCI. MPI: ScaMPI.

На кластере НИВЦ МГУ прогнозирование дало ошибку ~11% для двух процессов, в остальных случаях погрешность не превысила 6%. На кластере

ИСП РАН погрешность предсказания составила не более 4% в случае сети Myrinet. При использовании пиковой пропускной способности Ethernet (10Мб/сек) погрешность прогноза составила ~8-9% для последних трех точек графика. При использовании половинной пропускной способности (5 Мб/сек) прогнозирование дало погрешность не более 2% на том же количестве процессоров.

Программа «Net». Параллельная программа «Net», разработанная в лаборатории параллельных информационных технологий НИВЦ МГУ [7], представляет собой модель итеративного численного метода обрабатывающего матрицу, элементами которой являются числа с плавающей точкой двойной точности. Обрабатываемая в программе матрица распределена по процессам параллельной программы. Часть матрицы, находящуюся в памяти процесса будем называть в дальнейшем *локальной подматрицей*. Во время счета используются следующие структуры данных: *основная матрица*, которая содержит значения элементов локальной подматрицы, вычисленные на предыдущей итерации, *вспомогательная матрица*, размерами совпадает с основной матрицей, содержит значения элементов локальной подматрицы, вычисленные в рамках текущей итерации, *теневые грани*, одномерные массивы, хранящие отправляемые и принимаемые данные.

Распределение происходит «нарезкой» матрицы по двум направлениям, образуя тем самым двумерную решетку процессов (рис. 15). Программа вычисляет значение элемента матрицы используя шаблон «крест». Вычисленное значение сохраняется во вспомогательной матрице. После вычисления значений всех элементов локальной подматрицы, основная и вспомогательная матрица меняются местами и в рамках следующей итерации исходной матрицей является вспомогательная. Таким образом используемая в каждом процессе программы память приблизительно оценивается как *2*size_of_double*M*N/P*, где *M* и *N* – размеры всей матрицы, *P* – количество процессов программы, *size_of_double* – размер числа типа `double` в байтах.

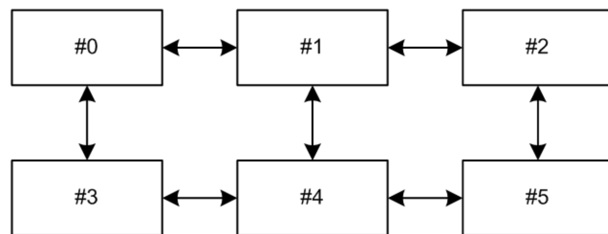


Рис. 15. Взаимодействие процессов программы Net в случае разделения матрицы на три столбца и две строки.

Программа позволяет варьировать размеры матрицы и шаблона. За счет увеличения размера шаблона увеличивается количество строк (столбцов) пересылаемых между процессами программы. Пересылаемые данные в

дальнейшем будут именоваться как теневые грани. Ширина теневой грани – количество строк (столбцов) локальной подматрицы передаваемых в другой процесс.

Программа выполняет следующие действия. Вначале вычисляются значения служебных переменных и происходит выделение памяти для локальной части основной и вспомогательной матриц, а также буферов представляющих теневые грани. Локальная часть основной матрицы инициализируется произвольными числами, после чего выполняется основной цикл модельного численного метода. Количество итераций цикла определяется заранее. Итерация начинается с обмена теневыми гранями. Каждая теневая грань собирается с помощью функции `CopyMatrixToBuffer`, записывающей соответствующие элементы основной матрицы в буфер теневой грани, после чего вызывается пара функций MPI, инициирующих не блокирующие посылку в стандартном режиме и не блокирующий прием данных. Посылаемые и принимаемые данные располагаются в различных буферах, общее число которых может доходить до 8 (в случае если имеется 4 других процесса, с которыми происходит обмен данными). После инициирования не блокирующих коммуникаций вызывается функция `MPI_Waitall`, ожидающая завершения всех начатых в рамках текущей итерации неблокирующих коммуникаций. По завершении коммуникаций содержимое принимающих буферов копируется в основную матрицу с помощью функции `CopyBufferToMatrix`.

Далее происходит обработка матрицы. Граничные элементы основной матрицы переписываются во вспомогательную без каких-либо изменений, остальные элементы пересчитываются применением шаблона «крест».

При завершении работы программа (процесс с нулевым номером) выдает общее время счета, время затраченное на копирование теневых граней и их передачу, и остальное время.

При переносе программы на язык Java был создан один класс net, функции исходной программы были объявлены как статические методы класса. Вместо функции `MPI_Wtime` был использован метод `System.currentTimeMillis()`.

В программе Net можно выделить шесть основных фрагментов, являющихся базовыми блоками:

- B1. Вычисление служебных переменных программы перед телом последовательного цикла.
- B2. Тело цикла, в котором элементы основной матрицы инициализируются произвольными числами.
- B3. Тело цикла, в котором элементы основной матрицы копируются в пересылаемый буфер (теневую грань).
- B4. Тело цикла, в котором элементы теневой грани копируются в основную матрицу.
- B5. Тело цикла, в котором элементы основной матрицы копируются во вспомогательную матрицу.

B6. Тело цикла, в котором пересчитывается значение элемента матрицы.

За исключением первого фрагмента все остальные являются телами однородных циклов.

При построении модели программы оценивалось время работы только перечисленных фрагментов программы. Оценка строилась для каждого значения ширины теневой грани. Оценки подвергался сразу весь однородный цикл: извлекался не только базовый блок, являющийся телом цикла, но и оператор цикла. Переменные управляющие выполнением цикла инициализировались теми же значениями, что при действительном выполнении программы. Измерялось время работы всего цикла, после чего вычислялось среднее время работы одной итерации, включающее в себя время работы базового блока и накладные расходы на организацию цикла.

Предсказание времени работы программы состояло из двух действий: получение оценок времени работы базовых блоков программы, непосредственно интерпретации модели программы.

Среди входных данных были зафиксированы: общий размер распределенной матрицы (2600x2600), количество итераций внешнего последовательного цикла (1000). Выполнялось шесть процессов, распределенных по одному на вычис-лительный узел. Таким образом, изменялась только ширина теневой грани.

Ширина теневой грани	B1	B2	B3	B4	B5	B6
2	3,40E+02	2,21E-04	1,80E-05	1,40E-05	1,30E-05	4,90E-05
3	3,11E+02	2,22E-04	1,65E-05	1,35E-05	1,20E-05	4,23E-05
6	2,57E+02	2,23E-04	1,70E-05	1,37E-05	1,23E-05	4,15E-05
8	3,37E+02	2,21E-04	1,70E-05	1,35E-05	1,23E-05	4,11E-05
10	3,19E+02	2,21E-04	1,68E-05	1,36E-05	1,22E-05	4,44E-05
12	3,43E+02	2,23E-04	1,70E-05	1,35E-05	1,28E-05	5,28E-05
14	3,43E+02	2,21E-04	1,67E-05	1,34E-05	1,23E-05	4,25E-05
16	3,22E+02	2,21E-04	1,71E-05	1,34E-05	1,23E-05	4,15E-05
18	3,54E+02	2,23E-04	1,94E-05	1,47E-05	1,22E-05	4,31E-05
20	3,26E+02	2,21E-04	2,49E-05	2,06E-05	1,71E-05	5,79E-05
Среднее значение	**3,25E+02**	**2,22E-04**	**1,80E-05**	**1,44E-05**	**1,28E-05**	**4,56E-05**
С. К. О.	**2,73E+01**	**7,40E-07**	**2,55E-06**	**2,22E-06**	**1,53E-06**	**5,73E-06**

Таб. 2. Оценки времени работы базовых блоков (мсек.), полученные на кластере ИСП РАН.

Ширина теневой грани	Блок №1	Блок №2	Блок №3	Блок №4	Блок №5	Блок №6
2	1,15E+03	3,32E-04	1,23E-05	1,00E-05	9,00E-06	3,80E-05
3	1,34E+03	3,20E-04	1,18E-05	9,67E-06	9,00E-06	3,17E-05
6	1,65E+03	3,32E-04	1,16E-05	9,67E-06	9,11E-06	2,75E-05
8	1,26E+03	3,32E-04	1,15E-05	9,25E-06	8,92E-06	2,61E-05
10	1,82E+03	3,19E-04	1,15E-05	9,20E-06	8,80E-06	2,50E-05
12	1,10E+03	3,32E-04	1,16E-05	9,28E-06	8,83E-06	2,34E-05
14	1,09E+03	3,32E-04	1,24E-05	9,67E-06	8,86E-06	2,33E-05
16	1,58E+03	3,20E-04	1,14E-05	9,21E-06	8,79E-06	2,43E-05
18	1,86E+03	3,32E-04	1,16E-05	9,37E-06	9,11E-06	2,41E-05
20	1,03E+03	3,33E-04	1,15E-05	9,23E-06	8,87E-06	2,67E-05
Среднее значение	**1,39E+03**	**3,29E-04**	**1,17E-05**	**9,45E-06**	**8,93E-06**	**2,70E-05**
С. К. О.	**9,95E+01**	**1,93E-06**	**1,17E-07**	**8,74E-08**	**3,80E-08**	**1,46E-06**

Таб. 3. Оценки времени работы базовых блоков (мсек.), полученные на кластере Leo НИВЦ МГУ.

Для каждого значения ширины теневой грани получался отдельный набор оценок, за счет чего были вычислены средние значения и среднее квадратичное отклонение. В среднем отклонения не превышают 10% от среднего значения оценки. Полученные средние значения были внесены в модель, после чего была проведена интерпретация для всех значений ширины теневой грани. Полученные значения оценок показаны в Таблицах 2 (кластер ИСП РАН) и 3 (кластер НИВЦ МГУ).

При прогнозировании времени работы программы на кластере ИСП РАН для сети Ethernet использовалась пропускная способность 5 Мб/сек. (интерпретатор моделирует сеть с полнодуплексными каналами, 10 Мб/сек. – пропускная способность Fast Ethernet). Результаты прогнозирования представлены на рис. 16 и 17.

При прогнозировании времени работы программы на кластере Leo НИВЦ МГУ пропускная способность сети SCI была взята 150 Мб/сек. (½ пропускной способности от заявленной производителем для C MPI, накладные расходы на вызов функции из Java не учитывались). Результаты прогнозирования представлены на рис. 18.

Известны результаты выполнения программы “Net”, реализованной на языке C, на кластерах НИВЦ МГУ (рис. 19) [7]. Задача выполнялась на кластерах SCI и SKY, использовалось 8 процессоров. Размер обрабатываемой матрицы составлял 3000x3000 элементов, что дает распределение ~ 1125000 элементов матрицы на один процесс параллельной программы. На рис. 19 видно, что при ширине теневой грани равной 2, 4, 6, 8 время выполнения программы меньшее

на кластере SKY, при ширине теневой грани 10 и более программа быстрее выполняется на кластере SCI.

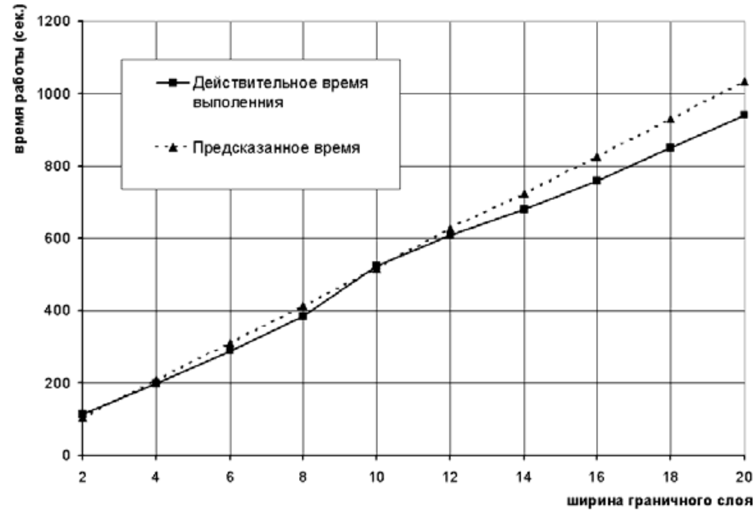


Рис. 16. Сравнение предсказанного и действительного времени работы программы Net. Кластер ИСП РАН, сеть Myrinet.. MPI: tmpich-gm-1.2.5-10.

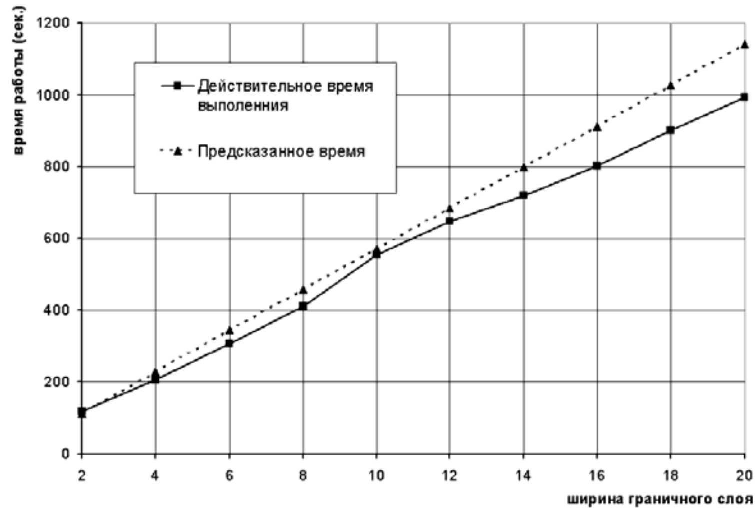


Рис. 17. Сравнение предсказанного и действительного времени работы программы Net. Кластер ИСП РАН, сеть Ethernet.. MPI: tmpich-1.2.5.10-ch_p4-gcc.

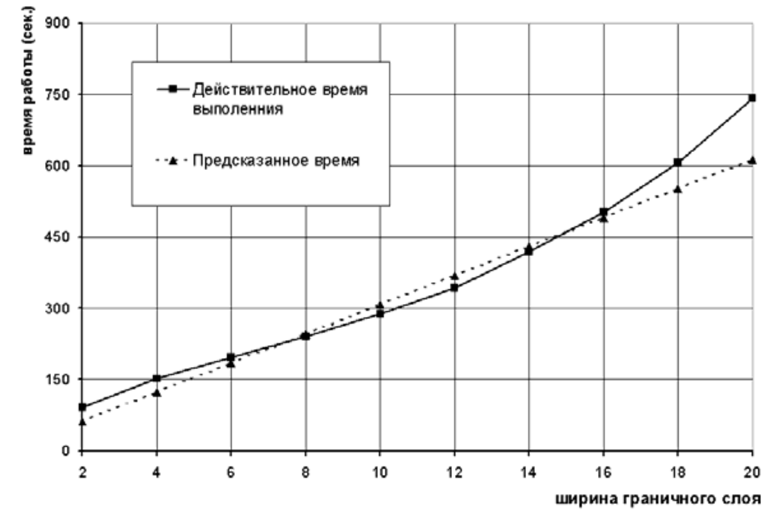


Рис. 18. Сравнение предсказанного и действительного времени работы программы Net. Кластер НИВЦ МГУ, сеть SCI.. MPI: ScaMPI.

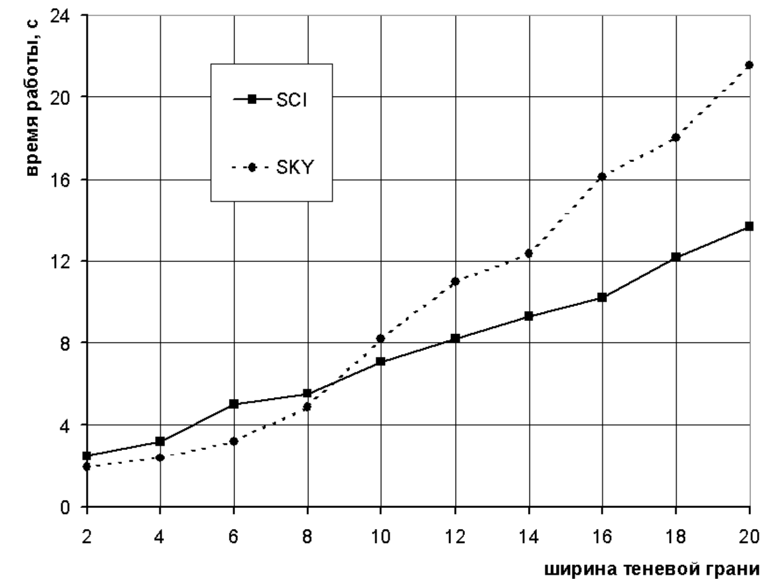


Рис. 19. Время выполнения программы Net на кластерах SCI и SKY.

Была проведена интерпретация модели Java-программы Net в целях получения аналогичных результатов. Оценки времени работы базовых блоков были получены на кластере SCI, при прогнозировании времени для кластера SKY данные оценки были скорректированы с помощью служебной переменной интерпретатора `//CPUrate`, получившей значение 0.59 (отношение тактовых частот процессоров). Характеристики коммуникационных сетей были b = 300 Mb/s, l = 4 us кластера SCI и b = 0.25 Mb/s, l = 160 us соответственно. Результаты прогнозирования показаны на рис. 20. Прогнозирование показало, что кластер с процессорами 500 MHz начинает показывать лучшее время работы программы начиная с ширины теневой грани равной 6. Погрешность предсказания поведения программы может быть объяснена тем, что кластер SKY не был доступен для непосредственного анализа его характеристик.

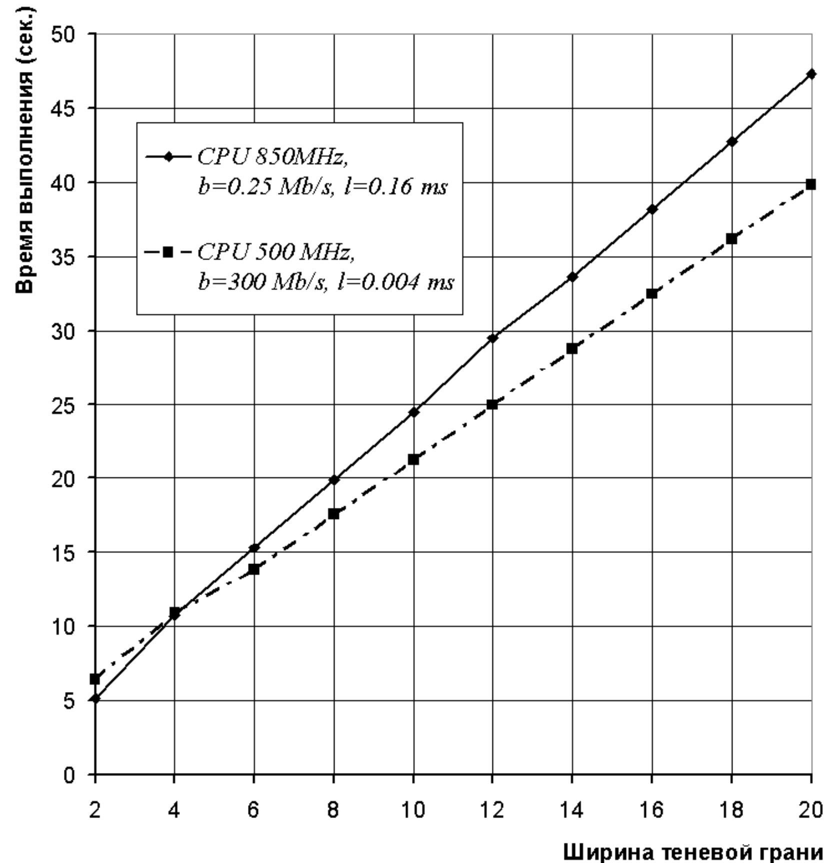


Рис. 20. Предсказанное время работы программы Net для различных характеристик вычислительного кластера.

4.2. Сравнение производительности C и Java.

Во время исследования свойств программы, использовалось 6 процессоров, количество обрабатываемых элементов матрицы в рамках одного процесса было сохранено (см. выше, рис. 19) за счет уменьшения размера обшей матрицы до 2600x2600, программа выполняла 1000 итераций внешнего цикла. Процессы параллельной программы были распределены по одному на вычислительный узел. Результаты выполнения программы (как C, так и Java версии) представлены на рис. 21-22. Java-программа выполнялась в двух режимах, с клиентской и серверной версиями VM, использовался SUN JDK 1.4.2_03. Для компиляции C программы использовался компилятор gcc 3.2.2; ключи компиляции: «-O3». Данные передавались через сети Myrinet (рис. 21) и Ethernet (рис. 22).

В программе подсчитывалось время затрачиваемое на подготовку теневых граней к пересылке (запись в них значений соответствующих элементов матрицы), непосредственно коммуникации, и копирование полученных теневых граней в матрицу. Это операции являются накладными расходами на организацию параллельных вычислений. На рис. 23-25 представлены графики, где время вычислений отделено от времени коммуникаций и подготовки передаваемых буферов. Несмотря на то, что работало всего 6 процессов, время коммуникаций с использованием сети Ethernet составило значительную часть от общего времени работы программы (порядка 25%). Аналогичный показатель для сети Myrinet составил ~5%.

Следует отметить что реализации MPI mpich-1.2.5.10-ch_p4-gcc и lam-7.1.1 показали на данной программе различную эффективность выполнения неблокирующих коммуникаций (рис. 24, 26).

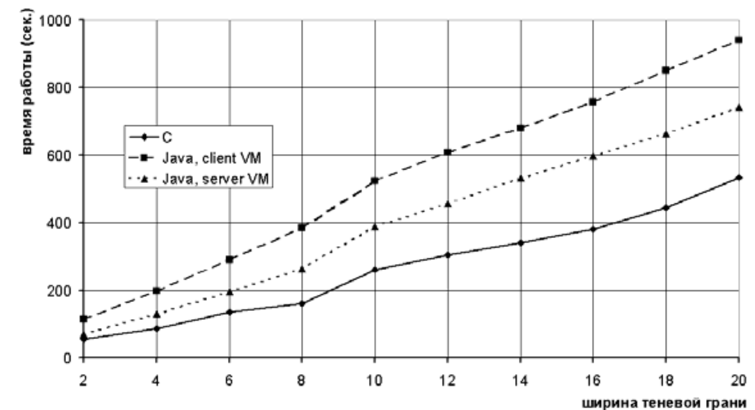


Рис. 21. Сравнение времени работы Java и C версий программы Net. Кластер ИСП РАН. Коммуникационная сеть – Myrinet, MPI: mpich-gm-1.2.5-10.

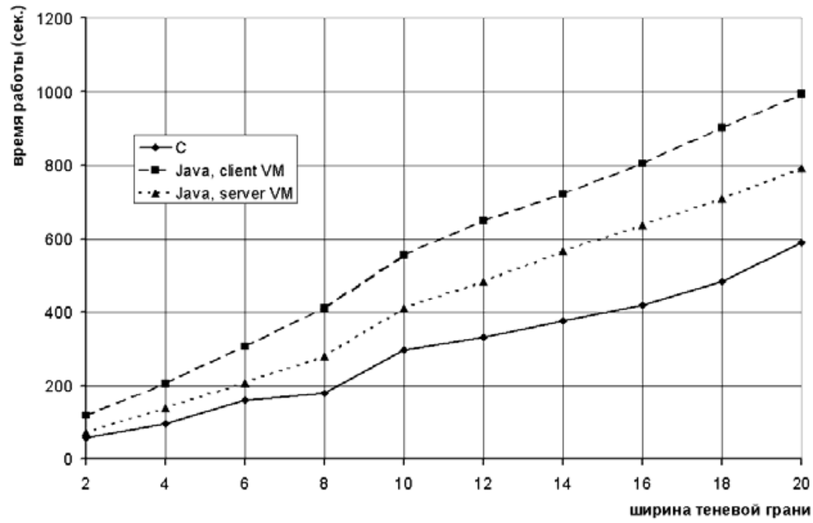


Рис. 22. Сравнение времени работы Java и C версий программы Net. Кластер ИСП РАН. Коммуникационная сеть – Ethernet, MPI: lam-7.1.1.

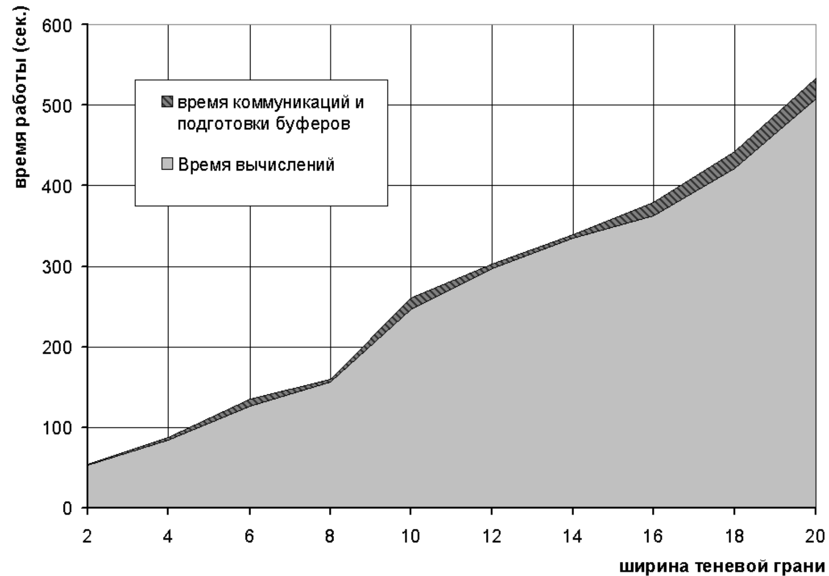


Рис. 23. Разделение времени работы программы Net (C версия). Кластер ИСП РАН. Коммуникационная сеть – Myrinet, MPI: mpich-gm-1.2.5-10.

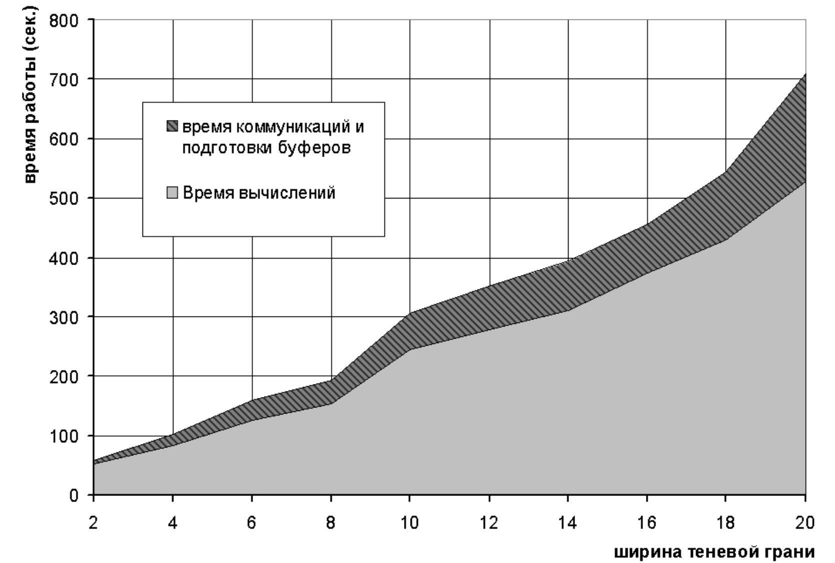


Рис. 24. Разделение времени работы программы Net (C версия). Кластер ИСП РАН. Коммуникационная сеть – Ethernet, MPI: mpich-1.2.5.10-ch_p4-gcc.

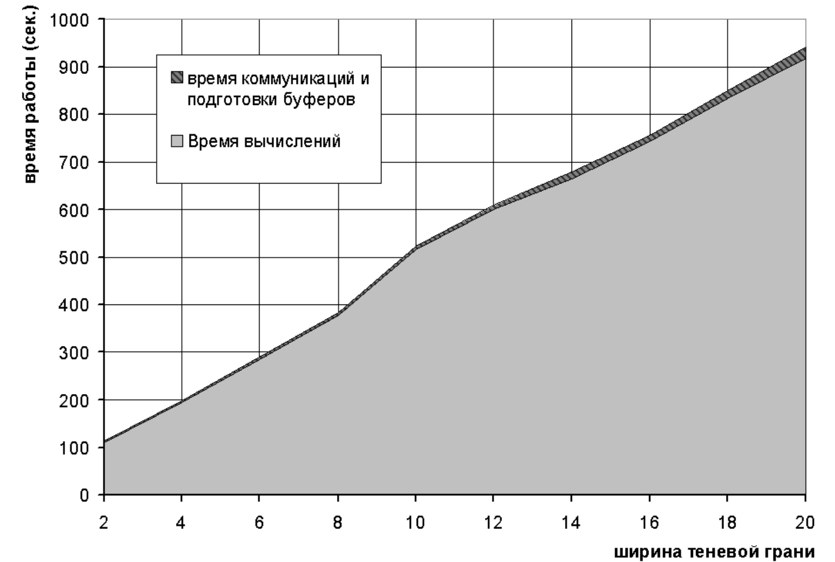


Рис. 25. Разделение времени работы программы Net (Java версия, JDK 1.4.2_03 client VM). Кластер ИСП РАН. Коммуникационная сеть – Myrinet, MPI: mpich-gm-1.2.5-10.

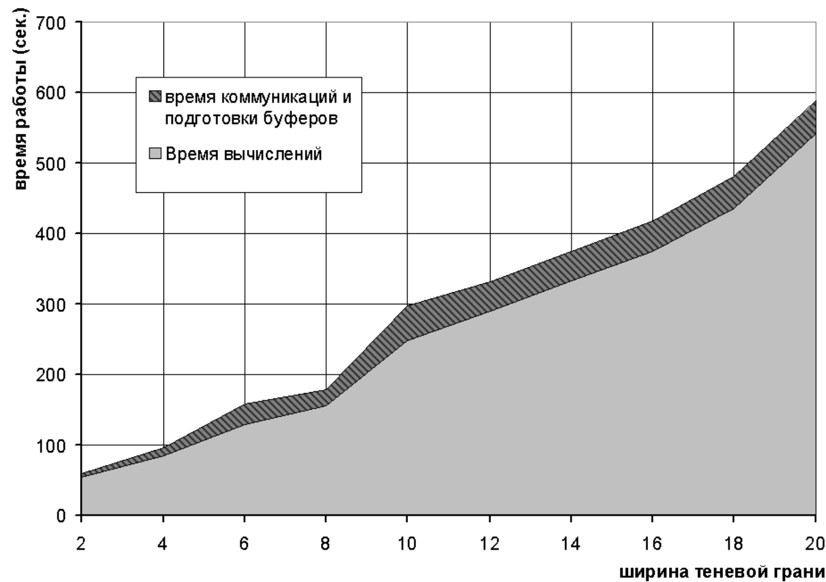
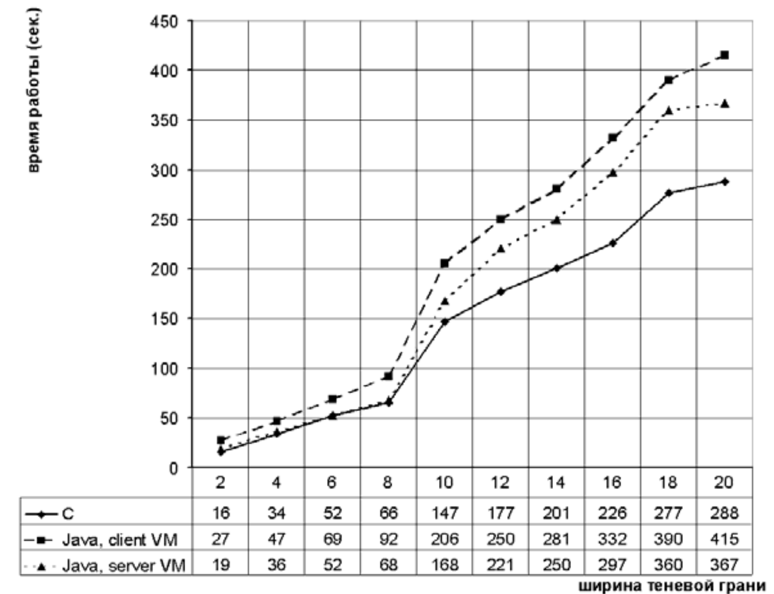


Рис. 26. Разделение времени работы программы Net (C версия). Кластер ИСП РАН. Коммуникационная сеть – Ethernet, MPI: lam-7.1.1.

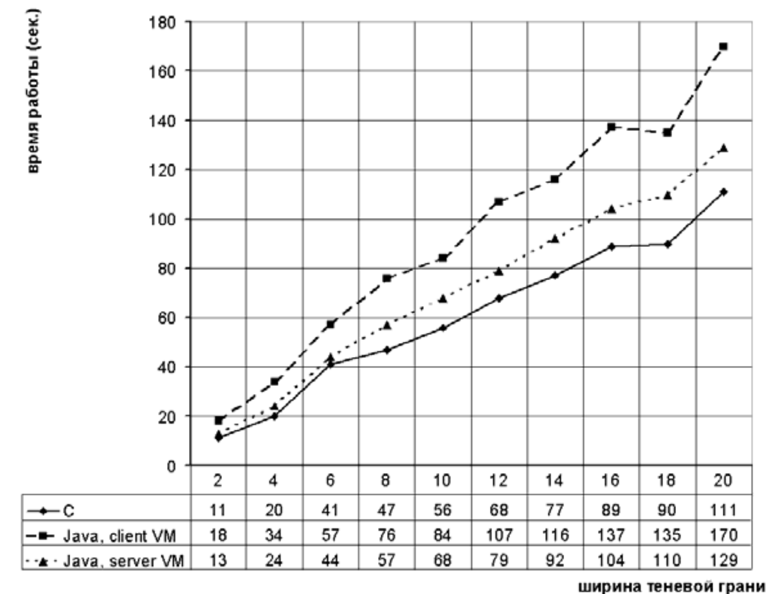
Также программа выполнялась при большем размере матрицы. На кластере ИСП РАН бралась матрица 17500x17500, что соответствует 80% загрузке оперативной памяти вычислительного узла. На кластере Leo бралась матрица 15000x15000, что соответствует ~34% загрузке оперативной памяти. Работало 12 (кластер ИСП РАН) и 10 (кластер Leo) процессов, распределенных по два на один вычислительных узел, программа выполняла 10 итераций внешнего последовательного цикла. В рамках одного процесса выполнялось не более чем 80 операций посылки/приема данных за все время работы программы. Сравнение времени работы версий программы, написанных на языках Java и C показано на рис. 27 и 28. Производительность серверной версии VM на кластере ИСП РАН при больших значениях теневой грани уступает производительности оптимизированной C-программы 27%, при малых значениях теневой грани (8 и меньше), производительности C и Java достаточно близки. На кластере НИВЦ МГУ серверная версия VM уступает аналогичной программе на C от 7 до 18% производительности.

Основную часть времени работы программы заняли вычисления, поскольку при увеличении ширины теневой грани увеличивается объем вычислений, требуемый для элемента матрицы. Доля копирования данных в коммуникационные буферы и их последующая пересылка занимали порядка 2% от общего времени работы программы на кластере Leo и менее 1% на кластере ИСП РАН.



	2	4	6	8	10	12	14	16	18	20
C	16	34	52	66	147	177	201	226	277	288
Java, client VM	27	47	69	92	206	250	281	332	390	415
Java, server VM	19	36	52	68	168	221	250	297	360	367

Рис. 27. Сравнение времени работы Java и C версий программы Net. Кластер ИСП РАН.



	2	4	6	8	10	12	14	16	18	20
C	11	20	41	47	56	68	77	89	90	111
Java, client VM	18	34	57	76	84	107	116	137	135	170
Java, server VM	13	24	44	57	68	79	92	104	110	129

Рис. 28. Сравнение времени работы Java и C версий программы Net. Кластер Leo НИВЦ МГУ.

5. Заключение

В работе рассмотрен интерпретатор среды ParJava, позволяющий получать оценки времени работы на инструментальном компьютере. Было показано как пользователь (прикладной программист) взаимодействует со средой в процессе получения оценки времени выполнения разрабатываемой программы.

Были рассмотрены три модельные программы, на примере которых были продемонстрированы возможности интерпретатора.

В качестве инструментальной машины использовался персональный компьютер под управлением операционной системы Solaris 9 с одним процессором Intel Pentium IV 2,66 GHz и 512 Mb оперативной памяти.

Результаты численных экспериментов показали, что относительная погрешность предсказания не превосходит 11%, и в среднем находится в интервале 2-8%. Несмотря на использование обычного персонального компьютера продолжительность интерпретации не превосходила двадцати секунд (при прогнозировании времени работы шести процессов интерпретация длится около 10 секунд).

Полученные результаты показывают, что прикладной программист способен вести разработку, достаточно точно оценивая масштабируемость текущей версии параллельной программы, не покидая инструментальный компьютер.

Среда ParJava используется в учебном процессе кафедры системного программирования факультета ВМиК МГУ и кафедры системного программирования ФУПМ МФТИ.

Среда ParJava доступна по http://www.ispras.ru/groups/ctt/parjava.html.

Литература

1. Виктор Иванников, Сергей Гайсарян, Арутюн Аветисян, Вартан Падарян. Применение среды ParJava для разработки параллельных программ. // Труды Института Системного Программирования. 2004. с. 41-62.
2. Victor Ivannikov, Serguei Gaissaryan, Arutyun Avetisyan, Vartan Padaryan. Improving properties of a parallel program in ParJava Environment // The 10th EuroPVM/MPI conference, Venice, Sept. 2003.
3. J2SE Documentation. http://java.sun.com/docs/index.html
4. S. R. Sarukkai and J. C. Yan. Event-Based Study of the Effect of Execution Environments on Parallel Program Performance. //Proceedings of the Forth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOT '96), San Jose CA, February 1996. pages 257-261.
5. Portable Batch System. http://www.openpbs.org/
6. TOP 500 Supercomputer Sites. http://www.top500.org/
7. Информационно-аналитический центр по параллельным вычислениям. http://www.parallel.ru/