Московский государственный университет
им. М.В. Ломоносова

факультет Вычислительной Математики и Кибернетики

На правах рукописи
УДК 519.682.3+519.687.1

Сальников Алексей Николаевич

СИСТЕМА РАЗРАБОТКИ И ПОДДЕРЖКИ ИСПОЛНЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация
на соискание учёной степени
кандидата физико-математических наук

Научные руководители:

член-корреспондент РАН,
профессор Л.Н. Королёв

к.ф.-м.н., доцент Н.Н. Попова

Москва – 2006

Оглавление

1. Введение

Целью диссертационной работы является создание инструментальных средств, облегчающих разработку параллельных программ для сред, гетерогенных по процессорным мощностям и коммуникациям, не требующих от пользователя знания архитектуры многопроцессорной системы. Алгоритм решаемой задачи представляется в виде ориентированного ациклического графа, в вершинах которого сосредоточены вычислительные операции (действия над данными), а рёбра задают зависимость по данным. Вершины графа на каждом уровне независимы между собой и могут быть исполнены параллельно. Таким образом, определённый выше граф задаёт параллельную программу. Описание графа содержится в текстовых файлах, которые можно подать на вход набору утилит, осуществляющих преобразование граф-программы в исходный код на C++ с вызовами MPI функций. Реализовано несколько способов балансировки загрузки процессоров многопроцессорной системы с учётом накладных расходов на передачу данных для гетерогенных коммуникационных сред.

1.1. Необходимость разработки высокоуровневых средств для создания параллельных программ

Существует некоторое количество научно-практических задач, решение которых требует большого количества ресурсов вычислительной системы. Такие задачи могут предъявлять огромные требования как к ресурсу процессорного времени, так и к ресурсу памяти. В качестве примера можно привести несколько таких задач.

При решении задачи моделирования климата на планете и составления прогноза погоды приходится искать компромисс между точностью и своевременностью получаемого решения. В случае реализации алгоритма на вычислительной системе это компромисс между временем работы алгоритма и числом точек сетки, на которой задаются характеристики. Использование многопроцессорной вычислительной системы позволяет задавать большое число точек, тем самым своевременно получать более точное решение. В отчёте [67] рассказывается об использовании кластерной вычислительной системы ASCI Blue, некоторое время назад возглавлявшей Top500 – список наиболее производительных вычислительных систем мира. Подобной задачей является и задача моделирования экосистемы [66].

Моделирование молекулярной динамики является одним из математических методов, активно используемых при решении проблем биохимии. Здесь в трёхмерную замкнутую область помещается некоторое количество молекул. Считается, что связь между атомами, составляющими молекулу, не может быть разорвана, но допустима такая деформация молекулы, которая не приводит к разрыву связи между атомами. В начальный момент набор молекул находится в стабильном состоянии, однако атомам молекул могут быть заданы векторы скорости, которые выведут систему из равновесного состояния. Далее в соответствии с химическими законами взаимодействия атомов между собой ищется такое состояние молекул, при котором система вновь придёт в равновесное состояние. В процессе моделирования скорости и координаты атомов модифицируются таким образом, чтобы минимизировать энергию системы [68]. Данная задача важна в биохимии для моделирования взаимодействий белков с мембранами и белков с цепочками ДНК и РНК [69]. Теоретически таким методом по первичной или вторичной форме белка возможно получить третичную форму, однако для молекулярной динамики в этом

случае требуется очень большое количество ресурсов и успевают накопиться погрешности метода.

Очень высока потребность в параллельных вычислениях при создании реалистических изображений, а также при создании компьютерных фильмов и анимации. Алгоритм здесь можно распараллеливать по кадрам и по пикселям, в статье [70] рассказывается об одном из методов построения изображений по пикселям с использованием многопроцессорной техники.

Список таких задач можно продолжать достаточно долго. Практика показывает, что использование однопроцессорной системы даже очень высокой производительности как правило не позволяет эффективно решать данные задачи за приемлемое время. Единственный способ решения таких задач – распараллеливание вычислений. В случае, когда задача оперирует огромным количеством данных, как например при обработке экспериментальных данных, полученных с ускорителя элементарных частиц, бывает очень трудно использовать одну какую-то многопроцессорную систему, поскольку данные на неё не будут помещаться. Далее в тексте рассмотрены ещё некоторые примеры задач, решаемых при помощи многопроцессорной техники. Для распараллеливания этих задач применяется подход, предлагаемый в данной работе.

Современные многопроцессорные вычислительные системы весьма разнообразны в своих архитектурных особенностях. В данной работе рассмотрено несколько архитектур многопроцессорных систем. В связи с этим возникает необходимость в программах, которые умеют автоматически подстраиваться под эти архитектурные особенности. Поскольку большинство современных вычислительных комплексов построено на основе многопроцессорных систем, для эффективного их использования необходимо создавать параллельные программы и заниматься проблемами, связанными с распараллеливаемостью алгоритмов. К сожалению, для каждой архитектуры многопроцессорной системы существует своя специфика написания эффективных параллельных программ. Программы, эффективные для одной архитектуры, могут быть неэффективны для другой. Таким образом, в большинстве случаев приходится выбирать между скоростью работы созданной параллельной программы и переносимостью данной параллельной программы с одной архитектуры на другую с точки зрения эффективности. Предложенный в работе подход стремится снизить потери в эффективности при переносе программы с одной архитектуры на другую.

В настоящий момент наиболее популярный способ написания параллельных программ – это создание параллельного программного кода с использованием таких библиотек, как MPI для кластеров и систем с распределённой памятью, а также OpenMP для SMP систем. Эти библиотеки позволяют создавать параллельные программы с переносимым исходным кодом между различными архитектурами многопроцессорных систем. Однако тот факт, что программу можно откомпилировать на произвольной многопроцессорной системе, совсем не означает, что на этой многопроцессорной системе мы не получим значительного проигрыша в производительности для той же программы. Добиться переносимости параллельной программы с точки зрения эффективности значительно сложнее и требует некоторых навыков в “искусстве программирования” [12], а также предварительного анализа алгоритма для выявления возможностей к распараллеливанию. Некоторую помощь в анализе программного кода могут оказать V-ray[71,72,73] и анализатор зависимостей, рассматриваемый в данной работе.

Всё вышесказанное и сложность учёта всех особенностей многопроцессорной системы оправдывает создание высокоуровневых средств параллельного программирования, базирующихся на MPI, OpenMP, pthread. Данные средства должны облегчать создание параллельных программ и стремиться учитывать

особенности многопроцессорных систем. К таким средствам можно отнести DVM[1], Cilk[2], PETSc[3,4], mpC[5] и предлагаемый в данной работе «PARUS».

1.2. Цель работы

Целью диссертационной работы является создание среды программирования для разработки и исполнения параллельных программ в гетерогенной среде. Создаваемая среда программирования должна отличаться от других высокоуровневых сред программирования ориентированностью на представление программы как графа зависимостей по данным и не требовать от пользователя знания архитектуры вычислительной системы. Для достижения основной цели диссертационной работы сформулируем несколько подцелей:

1. Разработать алгоритмы тестирования коммуникационной среды многопроцессорной системы, определяющие задержки при передаче сообщений и имитирующие при помощи фоновых шумовых сообщений наличие задач других пользователей.
2. Разработать алгоритмы тестирования, учитывающие гетерогенность коммуникаций, включая физическую структуру коммуникационной среды, но на этапе планирования вычислений позволяющие учитывать только результаты тестирования, с тем, чтобы достичь независимости этапа планирования от архитектурных особенностей многопроцессорной системы.
3. Создать анализатор зависимостей по данным в исходном коде на языке программирования C, адаптированный к нуждам системы, и разработать способ описания алгоритма решения задачи как набора действий, взаимодействующих между собой через передачу сообщений.
4. Разработать алгоритмы планирования вычислений для гетерогенных многопроцессорных систем, учитывающие нелинейность задержек от размера сообщения при передаче данных в многопроцессорной системе. Эти алгоритмы должны работать в статическом и динамическом режимах, а также в режиме с учётом подсказок пользователя.

Для создания среды программирования нужно решить ряд перечисленных ниже подзадач.

1.3. Проблема оптимизации последовательной части параллельной программы

В любой параллельной программе всегда остаётся доля последовательных вычислений, распараллеливание которых может оказаться весьма трудоёмкой задачей. Получается, что важно знать, как быстро выполняется тот или иной фрагмент последовательной программы и как оптимизировать последовательный код таким образом, чтобы он наиболее быстро выполнился на определённой платформе. Существует множество средств, предназначенных для поиска «узких» мест в программе, например, средства профилирования программ (такие, как gprof[11], Vtune[10], Xprofiler[6]). Существуют также более специализированные средства, предназначенные для выявления конкретной проблемы в коде программы – например, Valgrind[7,8,9], Intel Thread Checker.

1.4. Проблема тестирования производительности процессоров многопроцессорной системы

С точки зрения планирования вычислений чрезвычайно важно знать производительность определённой платформы многопроцессорной вычислительной системы, на которой в дальнейшем предполагается производить вычисления. Под платформой будем понимать аппаратную составляющую и операционную систему без систем ввода-вывода и внешней памяти. Иными словами, это операционная система, кэш-память и процессор. Для планирования вычислений необходимо получить сведения о производительности процессоров и задержках при передаче данных по коммуникационной среде многопроцессорной системы. Физически коммуникационная среда многопроцессорной системы может быть гетерогенна. Система тестирования должна быть устроена таким образом, чтобы информация о несбалансированности пропускной способности коммуникаций попала на этап планирования в форме, унифицированной для произвольной многопроцессорной системы.

Производительность процессоров обычно определяется на основе набора эталонных тестов, например теста LAPACK, на основе которого строится список 500 наиболее производительных вычислительных систем мира – Top500.

1.5. Тестирование производительности внутренней коммуникационной среды многопроцессорной системы

Одной из наиболее важных проблем при создании параллельных программ является проблема учёта производительности обменов данными между процессорами. Процесс выяснения обстоятельств определённого поведения коммуникационной среды весьма сложен, и на текущий момент нет общего механизма, который позволил бы дать ответ на вопрос: какой промежуток времени будет задействован при передачи порции данных определённой длины от одного процессора к другому. При ответе на него главной трудностью является выяснение всех деталей поведения каждого элемента коммуникационной среды, что почти всегда очень трудно сделать. Поэтому существует потребность в создании средств, способных выдавать какие-либо оценки относительно поведения сети, не опираясь на детальную информацию о физической составляющей коммуникационной среды. Предложенная в данной работе система тестирования коммуникационной среды многопроцессорной системы позволяет отчасти разрешить данную проблему.

1.6. Виды зависимостей по данным

Одной из проблем, которую приходится решать при создании параллельной программы, является проблема собственно распараллеливания вычислений. Как правило, на этапе распараллеливания приходится анализировать зависимости по данным между отдельными шагами и действиями алгоритма. Зависимость по данным – одно из наиболее важных понятий, которое в дальнейшем будет активно использоваться в работе. В главе, посвящённой реализации системы «PARUS», присутствует описание разработанного в диссертации анализатора зависимостей в С-программе. По С-программе строится граф зависимостей по данным, который в дальнейшем может быть использован для поиска фрагментов кода, которые могут быть распараллелены. Для анализатора зависимостей, предлагаемого в данной работе, зависимость по данным определяется как зависимость между отдельными операторами языка программирования С. Понятие зависимости может быть расширено до зависимостей по данным между отдельными шагами алгоритма и в таком виде использоваться для описания параллельного алгоритма. Зависимости по

данным активно используются при оптимизации кода компиляторами, о чем написано в книгах [74,75]. В России вопросами, связанными с зависимостями по данным, занимается Воеводин В.В. Теория зависимостей по данным им обсуждается в книге [76].

Пусть у нас есть оператор языка программирования S_i. Он оперирует со множеством переменных $Variables = Variables_{\text{in}} \cup Variables_{\text{out}}$. Здесь $Variables_{\text{in}}$ – множество переменных, которые оператор использует для своей работы, $Variables_{\text{out}}$ – множество переменных, которые меняют своё значение после работы оператора. В принципе, пересечение этих множеств может быть не пустым. Оператор может сперва прочитать значение переменной, а затем изменить его на новое значение в соответствии со старым. В дальнейшем для оператора S_i будем обозначать множество входных переменных как $\text{In}(S_i) = Variables_{\text{in}}$ и множество выходных переменных как $\text{Out}(S_i) = Variables_{\text{out}}$. Рассмотрим в качестве примера 2 оператора языка программирования С: «i++» и «a=b». В этом случае: $\text{In}('\text{i++}') = \{i\}$, $\text{Out}('\text{i++}') = \{i\}$ и $\text{In}('\text{a=b}') = \{b\}$, $\text{Out}('\text{a=b}') = \{a\}$.

Пусть множество операторов $S = \{S_1, S_2, \ldots, S_n\}$ упорядоченно. Упорядоченно в том смысле, что если $i < j$, то S_i идёт раньше по тексту программы, чем S_j. Между двумя операторами зависимость имеет место, если от изменения порядка этих операторов меняется результат работы программы (алгоритма).

Учитывая наличие зависимостей, сопоставим программе ориентированный граф. Операторам будут соответствовать вершины графа. Рёбрами будут соединены те вершины, между которыми имеет место зависимость.

Между двумя операторами по одной и той же переменной возможны 4 типа взаимоотношений, которые могут приводить к соответствующим зависимостям:

- **Зависимость по входам “in-in”**
- **Прямая зависимость “out-in”**
- **Обратная зависимость “in-out”**
- **Зависимость по выходам “out-out”**

Зависимость по входам описывается следующим образом: $dependence_{\text{in, in}}(S_j, S_i) = \{\text{In}(S_j) \cap \text{In}(S_i)\}$. По сути это означает, что 2 различных оператора читают одну и ту же переменную. Этот факт не накладывает никаких ограничений на порядок следования операторов и тем самым реально не определяет никакой зависимости, поэтому этот случай в дальнейшем можно не рассматривать.

Прямая зависимость описывается так: $dependence_{\text{out, in}}(S_j, S_i) = \{\text{In}(S_j) \cap \text{Out}(S_i)\}, i < j$. Здесь оператор S_j зависит от оператора S_i, при условии, что оператор S_j следует после оператора S_i. Данный тип зависимости наиболее естественен, поскольку описывает ситуацию, когда один из операторов напрямую использует данные, вырабатываемые другим оператором. В качестве примера рассмотрим следующий фрагмент кода:

```
1. a=1;
2. b=a+1;
```

Здесь оператор **1** имеет на выходе переменную **a** и $\text{Out}(1) = \{a\}$. Оператор **2** требует переменную **a** как входную $\text{In}(2) = \{a\}$. Данному фрагменту кода будет соответствовать граф, приведённый на рисунке 1.

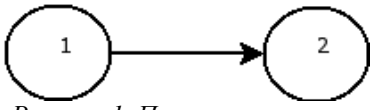

Рисунок 1. Прямая зависимость по данным.

Обратная зависимость, или **антизависимость**, задаётся следующим образом: $dependence_{\text{in},\text{out}}(S_j, S_i) = \{\text{Out}(S_j) \cap \text{In}(S_i)\}, i < j$. Здесь так же, как и для прямой зависимости, оператор S_j зависит от оператора S_i, при условии, что оператор S_j следует после оператора S_i. Этот тип зависимости характеризуется тем, что оператор должен прочитать старое значение переменной до того, как оно изменится. Другими словами, чтение и запись нельзя менять местами. В случае нарушения данной зависимости, то есть перестановки операторов местами, результат изменения данных оператором, у которого они являются выходом, приведёт к порче данных у оператора, для которого они указаны как вход. Рассмотрим следующий фрагмент кода в качестве иллюстрации этой проблемы:

```
1. b=1;
2. a=b+1;
3. c=b+2;
4.  b=0;
```

Заметим, что **4**-й оператор не требует результатов работы предыдущих, но если он будет выполнен между **1**-м и **2**-м или **2**-м и **3**-м, то результат будет отличаться от результата выполнения этого фрагмента кода при начальном порядке следования операторов. Проблемы возникнут, если **4**-й оператор выполнить до **1**-го; в этом случае значение b так и останется 1, а не 0, как было бы, если порядок следования операторов не менять. (Отметим, что порядок выполнения операторов **2** и **3** не фиксирован, они могут выполняться в произвольном порядке, в том числе параллельно). На рисунке 2 приведён граф, который получается для данного фрагмента.

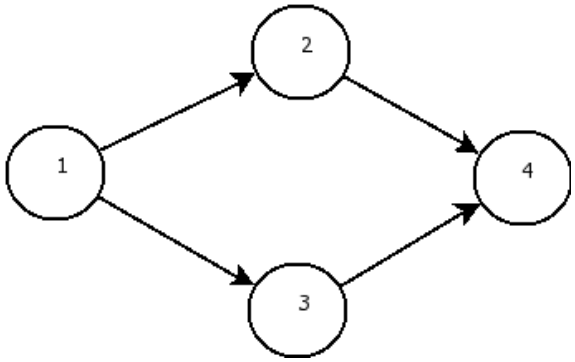

Рисунок 2. Антизависимость по данным.

Зависимость по выходам задаётся так: $dependence_{\text{out},\text{out}}(S_j, S_i) = \{\text{Out}(S_j) \cap \text{Out}(S_i)\}, i < j$. Здесь оператор S_j зависит от оператора S_i, при условии, что оператор S_j следует после оператора S_i. Этот тип зависимостей говорит нам, что порядок записей может быть важен. В случае перестановки операций записи или выполнения операций записи в переменную параллельно может возникнуть ситуация, когда операторы, следующие за этими, получат неправильные данные на вход.

Далее приведён пример фрагмента программного кода, иллюстрирующий

зависимость по выходам.

```
1. b=0;
   …
2. b=1;
3. c=b+2;
4. a=b+1;
```

Для корректной работы программы оператор **2** должен выполняться после **1**-го. Операторы **3** и **4** в любом случае могут быть исполнены независимо, но строго после оператора **2**. На рисунке 3 приведён соответствующий граф.

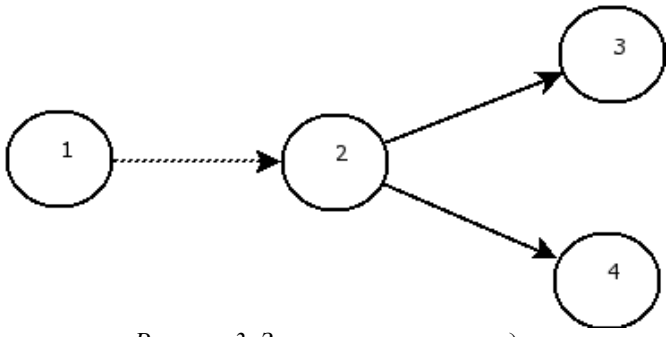

Рисунок 3. Зависимость по выходам.

Если же предположить такую последовательность выполнения операторов, как на рисунке 4, то неизвестно, какое значение **b** окажется в результате перед выполнением **3**-го оператора.

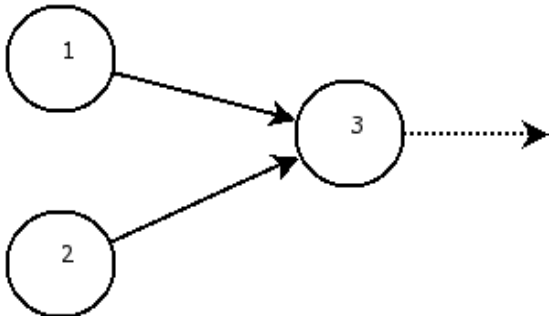

Рисунок 4. Некорректная последовательность операторов для зависимости по выходам.

2. Обзор существующих подходов к созданию параллельных программ

2.1. DVM система

DVM-система, созданная в Институте прикладной математики им. М.В.Келдыша РАН [1], позволяет разрабатывать на языках **C-DVM** и **Fortran-DVM** параллельные программы для многопроцессорных вычислительных систем различной архитектуры**.** Аббревиатура DVM соответствует двум понятиям: Distributed Virtual Memory и Distributed Virtual Machine. Первое отражает наличие единого адресного пространства; второе – использование виртуальных машин для двухступенчатой схемы отображения данных и вычислений на реальную

многопроцессорную систему.

При использовании языков C-DVM и Fortran-DVM исходный код, создаваемый программистом, годится и для последовательного, и для параллельного выполнения программы. Исходный код, помимо описания алгоритма обычными средствами языков Си или Фортран 77, содержит правила параллельного выполнения этого алгоритма. Эти правила оформляются синтаксически таким образом, что они являются “невидимыми” для стандартных компиляторов с последовательных языков Си и Фортран и не препятствуют выполнению и отладке DVM-программы на рабочих станциях как обычной последовательной программы.

Программисту предоставляются следующие возможности спецификации параллельного выполнения программы:

- распределение элементов массива между процессорами;
- распределение витков цикла между процессорами;
- спецификация параллельно выполняющихся секций программы (параллельных задач) и отображение их на процессоры;
- пожелание организации эффективного доступа к удаленным (расположенным на других процессорах) данным;
- пожелание организации эффективного выполнения редукционных операций - глобальных операций с расположенными на различных процессорах данными (например, их суммирование или нахождение максимального, либо минимального значения).

Компилятор переводит программу на языке C-DVM (Fortran-DVM) в программу на стандартном языке Си (Фортран), расширенную функциями системы поддержки выполнения DVM-программ, функциями Lib-DVM, которая предназначена для организации межпроцессорного взаимодействия и использует стандартные коммуникационные библиотеки (MPI, PVM, OpenMP – для машин на общей памяти).

Библиотека Lib-DVM является системой поддержки выполнения DVM-программ. Параллельная программа на языке Fortran-DVM (C-DVM) превращается в программу на языке Си (Fortran 77), содержащую вызовы функций Lib-DVM и выполняющуюся в соответствии с моделью SIMD на каждом выделенном задаче процессоре. Функции Lib-DVM, вызываемые при выполнении параллельной программы, служат для выполнения следующих основных операций:

1. инициализация системы поддержки и завершение работы с ней
2. построение абстрактной машины
3. отображение абстрактной машины
4. создание и уничтожение распределенного массива
5. отображение распределенного массива
6. отображение параллельного цикла
7. отображение параллельных задач
8. выполнение редукционных операций
9. обновление теневых граней распределенных массивов
10. создание и загрузка буферов для доступа к удаленным данным
11. выполнение операций ввода-вывода

Помимо выполнения этих операций, система поддержки должна

обеспечивать:

1. Обращения к подсистеме сбора характеристик выполнения программы на каждом процессоре.
2. Обращения к DVM-отладчику.
3. Накопление системной трассировки – трассировки обращений к функциям Lib-DVM и функциям коммуникационных библиотек. Эта трассировка используется для отладки DVM-программ, самой системы поддержки, а также служит входной информацией для предиктора.
4. Ввод параметров, управляющих работой системы поддержки, DVM-отладчика и подсистемы сбора характеристик выполнения.

Далее более подробно рассмотрим операции Lib-DVM. Ниже приведён список действий системы поддержки времени запуска.

Инициализация системы поддержки и завершение работы с ней. В работе системы поддержки можно выделить две стадии: инициализацию и завершение. При инициализации происходит следующее:

- Один из процессоров (процессор ввода-вывода) читает управляющие параметры из файлов и рассылает их остальным процессорам.
- В соответствии с введенными параметрами осуществляется настройка системы поддержки, DVM-отладчика и подсистемы сбора характеристик выполнения программы.
- Заводятся буфера в памяти всех процессоров для накопления системной трассировки и характеристик выполнения программы.
- Производится синхронизация времени на всех процессорах.
- По указанию пользователя выдаются сообщения, содержащие основные режимы работы системы и характеристики среды её выполнения.

При завершении происходит следующее:

- Производится запись в файлы накопленной системной трассировки и характеристик выполнения программы.
- Осуществляется оповещение пользователя (по его желанию) о завершении выполнения программы.

Построение и отображение абстрактной машины. Является подготовительной операцией для отображения на процессоры данных и вычислений, которое осуществляется через их отображение на абстрактную машину.

Создание и уничтожение распределенного массива. При создании массива строится его дескриптор, но память под него не выделяется. При уничтожении массива его дескриптор уничтожается и освобождается память, занимаемая массивом.

Отображение распределенного массива на абстрактную машину. При отображении массива на каждом процессоре выделяется память для тех его элементов, которые располагаются на данном процессоре, а также для теневых граней.

Отображение параллельного цикла на абстрактную машину. Для каждого процессора определяются витки цикла, которые будут выполняться на данном

процессоре, и порядок их выполнения.

Отображение параллельных задач на абстрактную машину. Для каждого процессора определяются задачи, которые будут выполняться на данном процессоре. Начало и завершение выполнения параллельных задач производится специальными функциями Lib-DVM.

Выполнение редукционных операций. Редукционные операции производятся посредством сбора на специально выделенном процессоре (центральном процессоре задачи) частичных результатов редукции, полученных на тех процессорах, которые участвовали в выполнении параллельного цикла. После вычисления на центральном процессоре окончательного результата он рассылается всем остальным процессорам. Совмещение выполнения редукции с вычислениями достигается имитацией параллельного процесса на центральном процессоре посредством регулярного опроса прихода сообщений.

Обновление теневых граней распределенных массивов. Обновление теневых граней реализовано как асинхронная операция. Совмещение обменов с вычислениями достигается посредством использования неблокирующих операций обмена.

Создание и загрузка буферов для доступа к удаленным данным. Размер буферов определяется посредством анализа на каждом процессоре ссылок на удаленные элементы и диапазона витков цикла, выполняющихся на данном процессоре. Загрузка буферов, как и обновление теневых граней, может совмещаться с вычислениями.

Выполнение операций ввода-вывода. Для программ на языке Си разработаны специальные версии стандартных программ ввода-вывода. Ввод информации производится на процессоре ввода-вывода посредством обычных функций ввода, а затем она рассылается остальным процессорам в соответствии с распределением данных. Для вывода данных, отсутствующих на процессоре ввода-вывода, они сначала пересылаются на него, а затем выводятся с него посредством обычных функций вывода.

Ввод-вывод для программ на языке Фортран организуется компилятором аналогично, при этом используются специальные функции Lib-DVM для перемещения данных между буфером процессора ввода-вывода и теми процессорами, на которых эти данные располагаются.

Накопление трассировки обращений к функциям Lib-DVM и функциям коммуникационных библиотек. Эта трассировка используется для отладки DVM-программ, самой системы поддержки, а также служит входной информацией для предсказателя времени работы DVM-программы.

Трассировка содержит информацию о вызываемых функциях и их параметрах, а также сведения о функционировании внутренних механизмов системы поддержки. Она может либо накапливаться в буфере, либо выводиться непосредственно в файл. Накопленная в буфере трассировка выгружается в файл при завершении выполнения программы. Состав трассируемых событий и режимы трассировки задаются в файле параметров.

Ниже приведён пример DVM программы. В примере матрица разбивается на двумерную решетку таким образом, что процессоры получают равные порции элементов. Если указать специальные опции системы поддержки времени запуска, то можно распределять данные неравными порциями. Первый цикл - инициализация, второй - итерационный. В итерационный цикл вложен параллельный цикл с зависимостью по данным, редукцией и локальной переменной, своей на каждом процессоре.

```
    PROGRAM  SOR
    PARAMETER  ( N = 10 )
    REAL   A( N, N ),  EPS,  MAXEPS, W
    INTEGER   ITMAX
*DVM$   DISTRIBUTE     A  ( BLOCK,  BLOCK )
        PRINT *,  '**********  TEST_SOR   **********'
    ITMAX=20
    MAXEPS = 0.5E - 5
    W = 0.5
*DVM$   PARALLEL   ( J, I )  ON  A( I, J )
    DO  1  J = 1, N
    DO  1  I = 1, N
        IF ( I .EQ.J)   THEN
             A( I, J ) = N + 2
        ELSE
             A( I, J ) = -1.0
        ENDIF
1   CONTINUE
    DO  2   IT = 1, ITMAX
        EPS = 0.
*DVM$   PARALLEL  ( J, I)   ON  A( I,  J),  NEW (S),
*DVM$*  REDUCTION ( MAX( EPS )), ACROSS  (A(1:1,1:1))

    DO  21   J = 2, N-1
    DO  21   I = 2, N-1
        S = A( I, J )
        A( I, J ) = (W / 4) * (A( I-1, J ) + A( I+1, J ) + A( I, J-1 )
     *      A( I, J+1 )) + ( 1-W ) * A( I, J)
    EPS = MAX ( EPS,  ABS( S - A( I, J )))
21  CONTINUE
    PRINT 200,  IT, EPS
200     FORMAT(' IT = ',I4, '   EPS = ', E14.7)
    IF    (EPS  .LT.  MAXEPS )     GO TO   4
2   CONTINUE
4       OPEN (3, FILE='SOR.DAT', FORM='FORMATTED',STATUS='UNKNOWN')
        WRITE (3,*)   A
        CLOSE (3)
    END
```

2.2. T-система

Одной из ключевых компонент программы Российско-Белорусского союза «СКИФ» «Разработка и освоение в серийном производстве семейства высокопроизводительных вычислительных систем с параллельной архитектурой (суперкомпьютеров) и создание прикладных программно-аппаратных комплексов на их основе» является система автоматического распараллеливания вычислений – «T-система»[44].

T-система основывается на парадигме функционального программирования для обеспечения динамического распараллеливания программ. Язык программирования T++ является синтаксически и семантически гладким языковым расширением стандартного языка программирования C++. Под синтаксической и семантической гладкостью здесь понимается прежде всего наличие естественного вложения конструкций языка C++ в расширенный (по отношению к нему) синтаксис и семантику языка T++.

Далее перечисляются добавленные в язык C++ конструкции; контексты, в которых возможно их употребление, и то, как они влияют на выполнение итоговой программы. Всего к стандартному набору C++ добавляется пять ключевых слов:

tfun, **tval**, **tptr**, **tout**, **tct** и две специальные функции: **tdrop**, **twait.** Эти функции **twait** и **tdrop** необходимы для выполнения специфических операций, у которых нет прямых аналогов в языке C++.

- **tfun** – атрибут, который можно указать непосредственно перед описанием типа функции. Функция не может являться методом; это должна быть обычная функция. Описанная с помощью этого ключевого слова функция называется T-функцией.
- **tval** – атрибут, который можно указать непосредственно перед описанием типа переменной. Описанная с помощью этого ключевого слова переменная называется T-переменной; в качестве значения T-переменная содержит неготовую величину (T-величину).
- **tptr** – T++-аналог для определения указателей (ссылок). Используется для описания глобальных указателей в структурах данных.
- **tout** – атрибут, который можно указать непосредственно перед описанием типа выходного аргумента T-функции. T++-аналог для определения аргументов, передаваемых по ссылке ('&') для их дальнейшей модификации в теле функции.
- **tct** – явное определение T-контекста. Служит для определения дополнительных свойств T-сущностей (специфических сущностей, поддерживаемой T-системой).
- **tdrop** – стандартная функция от одного аргумента. Может быть вызвана от любой T-величины.
- **twait** – стандартная функция от двух аргументов: T-сущности и паттерна событий. Возвращает статус произошедших с T-сущностью соответствующих указанному паттерну событий.

Ниже приведён пример простейшей программы на T++ – вычисление числа Фибоначчи с заданным номером в последовательности Фибоначчи. Этот алгоритм является широко распространённым тестом на производительность систем динамического распараллеливания. В тесте порождается огромное количество параллельным образом обрабатываемых последовательных фрагментов кода, которые равномерно распределяются между доступными вычислительными ресурсами.

```
tfun int fib(int n)
{
        if (n<2) return 1;
        return (fib(n-1)+fib(n-2));
}
tfun int main (int argc, char *argv[])
{
        int n = atoi(argv[1]);
        printf("Fibonacci %d is %d\n",n,(int)fib(n));
        return 0;
}
```

Используется единственная T-функция, fib, рекурсивно вызывающая сама

себя. Поскольку результатом T-функции является T-величина, необходимо явное преобразование типов (int)fib(n) при вызове функции printf. Такое преобразование вызывает ожидание потока функции main до тех пор, пока не будет вычислен результат функции fib(n). Сам вызов fib(n) порождает граф (дерево) вызовов fib с меньшими значениями входного аргумента, причем ветви дерева могут вычисляться параллельно.

Архитектура T-системы (OpenTS) построена по «микроядерному» принципу. Микроядро системы структурно организовано как три программных уровня: S, M и T.

S-уровень. Назначение S-уровня:

- Создать высокопроизводительную надстройку над стандартными средствами управления памятью (кэширующий аллокатор, механизмы подсчета ссылок) и потоками исполнения (кэширование тредов).
- Ввести абстракцию данных и обеспечить высокоэффективный разноприоритетный транспорт для их доставки в виде активных сообщений.
- Поддержать «суперпамять», или распределенную, программно-управляемую память, доступную для всех существующих в распределенной супервычислительной среде потоков.

S-уровень может быть по-разному реализован для разных архитектур.

M-уровень. Назначение M-уровня:

- Поддержать мобильные потоки исполнения, объекты и ссылки
- Поддержать «копирование по необходимости» (Copy-On-Write) для данных суперпамяти.
- Поддержать операцию блокирования и освобождения мобильных объектов.
- Поддержать иерархию досок объявлений с информацией о нераспределенных задачах и незадействованных процессорных и прочих ресурсов.

T-уровень. Назначение T-уровня:

- Реализовать понятия неготового значения и ссылки на неготовое значение.
- Реализовать понятие T-функции – мобильной функции, являющейся гранулой параллелизма.

В качестве базового уровня поддержки сущностей T-системы была создана и реализована концепция «суперпамяти» – специализированной общей памяти для организации обменов данными между T-функциями. Для T-величин, хранящихся с ячейках «суперпамяти», реализован подсчёт ссылок и распределённая сборка мусора. С целью устойчивой работы сборщика мусора в условиях сетевых коммуникаций реализован подсчёт ссылок на объекты.

2.3. mpC

Язык программирования **mpC** является высокоуровневым языком параллельного программирования (расширение ANSI C), спроектированным специально для создания переносимых программ, умеющих адаптироваться к гетерогенной среде, создаваемой сетью компьютеров [5,45]. Основная идея **mpC** заключается в том, что mpC-программа явно задаёт абстрактную модель сети и передаёт данные внутри этой модели. Вычисления, по аналогии с передачей данных, производятся в построенной модели абстрактной сети. Система программирования **mpC** в дальнейшем использует информацию о абстрактной сети для отображения её на реальную вычислительную сеть. Реальная сеть может иметь произвольную конфигурацию. Отображение производится таким образом, чтобы обеспечить

эффективность работы программы на данной вычислительной сети. Отображение модели производится во время запуска программы динамически, для этой цели используется информация о производительности процессоров реальной вычислительной сети и информация о скоростях передачи данных между процессорами.

Система программирования **mpC** содержит компилятор, подсистему времени запуска, библиотеку функций и пользовательский интерфейс в командной строке. Компилятор переводит исходный код mpC-программы в исходный код на ANSI C, куда добавляет код вызовов функций подсистемы времени запуска. Подсистема времени запуска управляет процессами, составляющими параллельную программу, и обеспечивает передачу данных между процессами. Данная подсистема скрывает детали реализации конкретных способов передачи данных в многопроцессорной вычислительной системе. В текущий момент подсистема времени запуска реализована через стандарт передачи сообщений **MPI**.

В **mpC** язык программирования C расширен некоторыми дополнительными конструкциями. Для управления распределением функций по процессорам определяется группа синтаксических конструкций. В эту группу входят 2 конструкции, которые выставляются перед описанием функций. Конструкция **[*]** означает, что функция должна быть выполнена всеми виртуальными процессорами mpC-программы. Конструкция **[host]** означает, что функция должна быть выполнена одним процессором, причём тем, с которого ведётся ввод/вывод в терминал.

Присутствует возможность описания реплицированных по виртуальным процессорам mpC-программы переменных. Для этой цели служит ключевое слово **repl**, которое указывается перед объявлением переменной, в результате на каждом виртуальном процессоре оказывается копия данной переменной. Конструкция **[+]**, которая указывается после имени переменной, означает, что все копии на каждом виртуальном процессоре нужно сложить и синхронизировать результат таким образом, чтобы везде оказалось одинаковое значение суммы всех копий переменной.

В **mpC** есть богатые возможности для описания виртуальной сети процессоров. Сеть – это множество виртуальных процессоров. Определение сети вызывает создание группы процессов, представляющей эту сеть, так что каждый виртуальный процессор представляется отдельным процессом параллельной программы. Описание вычислений на виртуальных процессорах сети вызывает выполнение соответствующих вычислений процессами параллельной программы. Реальные процессы MPI-программы представляют виртуальные процессоры **mpC**-сети. Важное отличие реальных процессов от виртуальных процессоров заключается в том, что в разные моменты выполнения параллельной программы один и тот же реальный процесс может представлять различные виртуальные процессоры **mpC**-сети. В момент, когда программа доходит до определения **mpC**-сети, происходит отображение виртуальных процессоров сети на реальные процессы параллельной программы, и это отображение сохраняется на всё время жизни **mpC**-сети.

Сеть объявляется при помощи ключевого слова **net**, за ключевым словом идёт описание типа сети и размерность сети, количество виртуальных процессоров, объединённых сетью, затем следует имя сети. После объявления сети по аналогии с конструкцией **[host]** можно указывать **[имя сети]** перед функциями и переменными. В случае функции это означает, что функция должна быть исполнена на всех виртуальных процессорах созданной сети, а в случае переменной — переменная должна быть определена на всех виртуальных процессорах сети. Время жизни заданной сети ограничивается блоком языка C, в котором она создана. Однако если создать сеть с модификатором **static**, то она будет актуальна всё время жизни программы, а не только на время существования блока. Основным средством языка

mpC для описания сложных обменов данными являются ***подсети***. Подсеть - это любое подмножество виртуальных процессоров некоторой сети. К примеру, любая строка или столбец решётки виртуальных процессоров сети типа **Mesh(m,n)** представляет собой подсеть этой сети. Подсеть задаётся ключевым словом **subnet** и ограничениями на родительскую сеть.

Язык **mpC** предоставляет программисту средства для спецификации относительных объёмов вычислений, выполняемых различными виртуальными процессорами той или иной сети. Система программирования использует эту предоставленную программистом информацию для того, чтобы по возможности отобразить виртуальные процессоры сети на процессы таким образом, чтобы вычисления на различных виртуальных процессорах выполнялись со скоростями, пропорциональными объёмам этих вычислений. Для этого нужно следующим образом определить тип сети:

```
nettype HeteroNet(int n, double v[n]) {
  coord I=n;
  node {
    I>=0: v[I];
  };
  parent [0];
};
```

Объявление **node { I>=0: v[I] }** читается следующим образом: для любого **I>=0** относительный объём вычислений, выполняемых виртуальным процессором с координатой **I**, задаётся значением **v[I]**. Далее вычисления будут распределяться в соответствии с производительностью реальных процессоров.

По умолчанию система программирования языка **mpC** использует одну и ту же оценку производительности участвующих в выполнении программы реальных процессоров, однажды полученную в результате выполнения специальной тестовой параллельной программы при инициализации системы в конкретной параллельной вычислительной среде. Если на задаче пользователя производительность процессоров отличается от той, которая получилась в результате выполнения тестовой программы, пользователь **mpC** может указать свои собственные значения. Язык **mpC** содержит средства, позволяющие программисту изменять оценку производительности реальных процессоров, используемую при отображении на них виртуальных процессоров, настраивая её на те вычисления, которые будут выполняться этими виртуальными процессорами.

2.4. Отличительные черты «PARUS» подхода

У предлагаемого в этой работе подхода к созданию параллельных программ есть некоторое количество особенностей, которые делают использование «PARUS» в определённых ситуациях более предпочтительным в сравнении с рассмотренными выше подходами.

В DVM исходный код программы дополняется специальными конструкциями, которые оформлены невидимым для компилятора с «чистого» языка программирования образом; следовательно, предполагается, что для решения соответствующей задачи должна быть создана последовательная программа. В «PARUS» не требуется реализация последовательной программы. Это даёт преимущество в том случае, когда последовательная реализация будет сложной, и расстановкой DVM-команд сложно добиться нужного эффекта в распараллеливании. В качестве примера такой задачи можно привести задачу построения множественного выравнивания нуклеотидных последовательностей, рассмотренную

в главе, посвящённой использованию «PARUS» для решения практических задач. Это замечание относится также и к T-системе, и к mpC.

Одной из важных особенностей «PARUS» является его ориентированность на гетерогенные многопроцессорные системы. Рассмотренные выше системы учитывают гетерогенность с точки зрения производительности процессоров, однако производительность коммуникаций учитывается в недостаточной степени. В разделе, посвящённом тестированию коммуникационной среды, показано, насколько сложно могут себя вести коммуникации в зависимости от варьирования некоторых параметров при передаче данных. Рассмотренные выше системы не обладают таким богатым, как в «PARUS», набором средств по тестированию коммуникационной среды многопроцессорной системы. На основе предложенных средств тестирования средствами «PARUS» осуществляется балансировка нагрузки на отдельные компоненты многопроцессорной системы с учётом нелинейности скорости передачи данных по коммуникациям от объёма передаваемых данных.

В «PARUS» присутствует анализатор зависимостей по данным. Этот анализатор после существенной доработки можно считать средством для автоматического распараллеливания C-программы. В текущий момент анализатор строит граф зависимостей по данным, но с очень большими ограничениями на исходный код C-программы и с чрезвычайно мелкозернистым параллелизмом. Стоит отметить, что в неявном виде анализ зависимостей производится в T-системе в момент определения зависимостей в параметрах функций. В T-системе допускается наличие только так называемых «чистых функций», то есть функций без побочных эффектов. В отличии от T-системы, где граф строится неявно, в «PARUS» граф строится явно, что является одновременно и достоинством и недостатком. Достоинство заключается в том, что разработчик параллельной программы может определять степень параллелизма в программе; определять, насколько вычислительно сложной делать отдельную вершину. Недостаток состоит в том, что зачастую это довольно трудоёмкая работа.

В mpC для задания гетерогенности нужно явно указывать долю вычислений, которая будет помещена на процессор. В «PARUS» производительность процессоров определяется на этапе тестирования многопроцессорной системы, и доля вычислений определяется автоматически в процессе назначения заданий процессорам.

3. Обзор алгоритмов планирования вычислений для многопроцессорных систем

3.1. Постановка задачи планирования вычислений

Для эффективного использования многопроцессорной системы, особенно для гетерогенных многопроцессорных систем, необходимо заниматься планированием вычислений. Планирование вычислений подразумевает поиск такого способа назначения работ по процессорам, при котором достигается некоторая цель в процессе функционирования многопроцессорной системы. Целей планирования последовательности назначений работ по процессорам может быть несколько. Обычно стремятся минимизировать общее время исполнения всего множества работ на многопроцессорной системе. Однако возможны вариации, когда стратегия построена таким образом, что минимизация времени исполнения происходит косвенно и не гарантируется стратегией. Далее будут рассмотрены некоторые стратегии планирования.

Планирование может проводится в нескольких режимах: заранее статически; в процессе исполнения параллельной программы динамически: специальной программой – планировщиком, которая управляет потоком пользовательских программ, попадающих на процессор; с учётом подсказок пользователя. Для алгоритма планирования всегда выделяется время планирования. Время планирования − это время между получением задания на планирование, например, в случае освобождения процессора или поступления новой работы, и выдачи реакции, например, непосредственное назначение работы на исполнение. Важной задачей является минимизировать это время. К сожалению, здесь приходится искать компромисс. Обычно близкое к оптимальному расписание назначений требует большого времени планирования и наоборот, легко сделать какое-то расписание, которое будет далеко от оптимального. Время планирования обычно зависит от количества работ. Важной характеристикой алгоритма планирования является характер данной зависимости. Алгоритм может строить хорошее расписание, но быть плохо масштабируемым к числу планируемых работ.

В целом теория планирования вычислений и построения расписаний подробно обсуждается в книге [46]. В книге [58] показано, что в общем случае данная задача относится к классу NP-полных задач, и это говорит о том, что к данной задаче не применимы переборные алгоритмы, однако можно использовать эвристические. Например, в работах [77,78,79] предлагаются алгоритмы для составления расписаний в многопроцессорных системах реального времени. Данные алгоритмы обладают полиномиальной и псевдополиномиальной сложностью в зависимости от строгости накладываемых ограничений. Эти алгоритмы предполагают прерывания исполняющихся работ, поэтому не подходят для системы «PARUS», рассматриваемой в данной работе, несмотря на хорошие оценки временной сложности алгоритмов.

Рассмотрим задачу планирования вычислений при следующих условиях. Предполагается наличие многопроцессорной системы с n процессорами, на которой необходимо исполнить m работ. Процессор может выполнять в каждый момент времени только одну работу. Работы обладают длительностью $\tau_{i,j}$ в случае назначения i работы на j процессор. Для каждой работы определён директивный интервал с $T_{start,i}$ и $T_{finish,i}$, которые определяют момент времени, после которого работа должна быть запущена, и момент времени, к которому работа должна быть завершена. Каждая работа обладает приоритетом p_i, который показывает её важность относительно других работ.

3.2. Списочные алгоритмы

Существует ряд довольно простых алгоритмов планирования, которые позволяют выбрать работу для назначения на освободившийся процессор из списка готовых работ. Критериев выбора может быть несколько. Например, назначается первая по списку работа (First Come First Served - FCFS) или работа с максимальным приоритетом (Higest Priority First - HPF). Возможно назначение в первую очередь работы, которая выполнится за кратчайшее время $\text{job\_number}_j = argmin(\tau_{k,j})$, где k – номер одной из работ, директивный интервал которых допускает запуск к текущему моменту времени. Один из вариантов – это случай, когда на процессор назначается работа, у которой верхняя граница директивного интервала наиболее близка к текущему моменту времени $t_{current}$, то есть $\text{job\_number} = argmin(t_{current} - T_{finish,k})$ (Earliest deadline first – EDF). Довольно популярен алгоритм, где процессоры образуют виртуальное кольцо с маркером и

следующая готовая к исполнению работа назначается на процессор, обладающий маркером, а затем маркер передаётся следующему процессору (Round Robin - RR).

Такого рода алгоритмы используются в системах реального времени. Примеры использования данных алгоритмов можно найти в [47,48]. Планирование процессов в операционной системе Linux в ядре, начиная с версии 2.6.8.1 осуществляется на основе модификации RR алгоритма, где, по мнению создателей алгоритма, время планирования не зависит от числа процессов за счёт особой системы подсчёта приоритетов [49].

3.3. Алгоритм, основанный на множестве очередей

Алгоритм FB (feedback) очередей с обратной связью использует n очередей, каждая из которых обслуживается в порядке поступления. Новая работа поступает в очередь с номером 0, затем после получения кванта времени она переходит в очередь со следующим номером и так далее, после очередного кванта времени. Время процессора планируется таким образом, что он обслуживает непустую очередь с наименьшим номером. В методе FB каждая вновь поступающая на обслуживание работа получает высокий приоритет и выполняется подряд в течение такого количества квантов времени, пока не появится новая работа. Если приход новой работы задерживается, то текущая работа не может проработать большее количество квантов времени, чем предыдущая работа.

Данный алгоритм наиболее эффективно работает с большим числом коротких по времени работ. Основное преимущество очередей FB и RR по сравнению с алгоритмом EDF и алгоритмом «кратчайшая по времени работа вперёд» в том, что FB и RR не требуют предварительной информации о времени выполнения работ. Пример использования алгоритмов такого типа можно найти в работе [50].

3.4. Алгоритм имитации отжига

Алгоритм имитации отжига основывается на имитации физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твёрдое. Предполагается, что атомы уже выстроились в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Процесс протекает при постепенно понижающейся температуре. Переход атома из одной ячейки в другую происходит с некоторой вероятностью, причём вероятность уменьшается с понижением температуры. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте. Атом может с некоторой вероятностью перейти и в состояние с большим уровнем энергии, однако глобально процесс протекает с понижением энергии. Впервые метод был предложен в работе [52]. В общем виде алгоритм описан в книге [51].

Для задачи планирования вычислений вводится понятие расписания. Расписание – последовательность исполнения работ для каждого процессора многопроцессорной системы. С каждой работой сопоставляется пара $(proc_j, \text{order\_number})$, задающая процессор, на котором она должна быть выполнена, и порядковый номер на процессоре. Работы на процессоре исполняются по очереди в соответствии с возрастанием порядковых номеров. Вводятся две операции: операция $\text{change\_proc}(job_i, proc_j, proc_k, \text{order\_number})$ и операция $\text{change\_order}(job_i, proc_j, \text{order\_number})$. Здесь первая операция переносит работу с одного процессора на другой и задаёт порядковый номер, а вторая оставляет работу на том же процессоре, но меняет её порядковый номер.

Вводится последовательность расписаний: $S_0, S_1, \ldots, S_n$. По достижении

последовательности S_n алгоритм останавливается. Задаётся каким-либо образом начальное расписание S_0. Далее к начальному расписанию случайным образом применяется несколько раз одна из описанных выше операций. В результате получается новое временное расписание S'. Временное расписание может стать постоянным с некоторой вероятностью $P(S', S_1)$. Таким образом, задаётся переход от одной итерации алгоритма к другой. Для расписания S_k определён переход к расписанию S_{k+1} через случайное применение операций с вероятностью $P(S', S_{k+1})$, где вероятность задаётся формулой:

$$P(S', S_{k+1}) = \begin{cases} 1, & time(S') - time(S_k) \le 0 \\ \exp\left(-\dfrac{time(S') - time(S_k)}{Q_k}\right), & time(S') - time(S_k) > 0 \end{cases}$$

Здесь $time(S)$ – функция, определяющая, как долго многопроцессорная система будет работать по соответствующему расписанию; задаёт аналог энергии для атомов в кристаллической решётки. Q_k – убывающая последовательность значений температуры. Закон, по которому происходит убывание последовательности, и скорость убывания могут задаваться произвольным образом по желанию создателя алгоритма. Также можно варьировать число применений операций по изменению расписания на одну итерацию алгоритма.

Об использовании алгоритма имитации отжига для планирования вычислений в системах реального времени можно прочитать в статье [53]. К сожалению, алгоритм имитации отжига обладает некоторым недостатком. Поскольку он похож на градиентный спуск, для него естественно попадать в локальные минимумы и не выбираться оттуда. Эта проблема решается в следующем классе алгоритмов – генетических алгоритмах.

3.5. Генетический алгоритм

Одним из популярных способов составления расписаний для многопроцессорных систем является использование генетического алгоритма. Генетический алгоритм – это алгоритм поиска максимума или минимума некоторой функции с помощью направленного перебора области определения функции. Перебор проводится методами, схожими с законами естественного отбора. Генетические алгоритмы являются подмножеством эвристических алгоритмов поиска. Генетические алгоритмы описаны в книге Джона Холланда [54].

На некотором n-мерном множестве задаётся числовая функция $f(x_1, x_2, \ldots, x_n)$. Предполагается, что мы ищем максимум данной функции (в дальнейшем она будет называться функцией качества). На множестве выбирается некоторое количество точек $X_1 = (x_{1,1}, \ldots, x_{1,n}), \ldots, X_m = (x_{m,1}, \ldots, x_{m,n})$, которые образуют начальную популяцию P_0. Каждое $X_1, \ldots, X_m$ называются хромосомами, или особями, а $x_1, \ldots, x_n$ – генами в хромосоме. Для алгоритма определено некоторое число операций, при помощи которых получается следующая популяция: операция скрещивания и операция мутации. При помощи генетических операций по текущей популяции P_k создаётся новая временная популяция P'. Временная популяция обычно больше по числу особей, чем та, из которой она получалась. В полученной временной популяции производится отбор. Для отбора особи сортируются в порядке убывания функции качества. Обычно отбор проводится таким образом, что из популяции удаляются особи с наименьшим значением

функции качества, но с учётом случайным образом наложенного штрафа. В результате наложения штрафа в следующей популяции останутся как особи с плохим значением функции качества, так и особи с хорошим значением; такая стратегия может дать лучшую особь в следующем после отобранного поколении.

Операция скрещивания осуществляет перемешивание генетического материала между особями. Полученный таким образом перемешанный потомок, может собрать в себя положительные качества обоих родителей, в результате получится особь со значением функции качества большим, чем у родителей. В процессе применения операции скрещивания 2 хромосомы, X_i и X_j, обмениваются своими фрагментами, при этом родительские особи остаются в новой временной популяции. Далее на рисунке 5 приведён пример с операцией скрещивания, где случайным образом выбирается точка k внутри хромосомы, а всё, что находится правее точки k, меняется местами в обоих хромосомах. Голова первой хромосомы соединяется с хвостом второй хромосомы, и наоборот. В результате во временную популяцию добавляется 2 новых особи X'_i и X'_j. Математически это можно записать следующим образом: $X'_i = \begin{pmatrix} x_{i,r}, & r<k \\ x_{j,r}, & r \geq k \end{pmatrix}$ и $X'_j = \begin{pmatrix} x_{j,r}, & r<k \\ x_{i,r}, & r \geq k \end{pmatrix}$, где r задаёт смещение по хромосоме для осуществления обмена частями. То есть $r \in \{1,\ldots,n\}$.

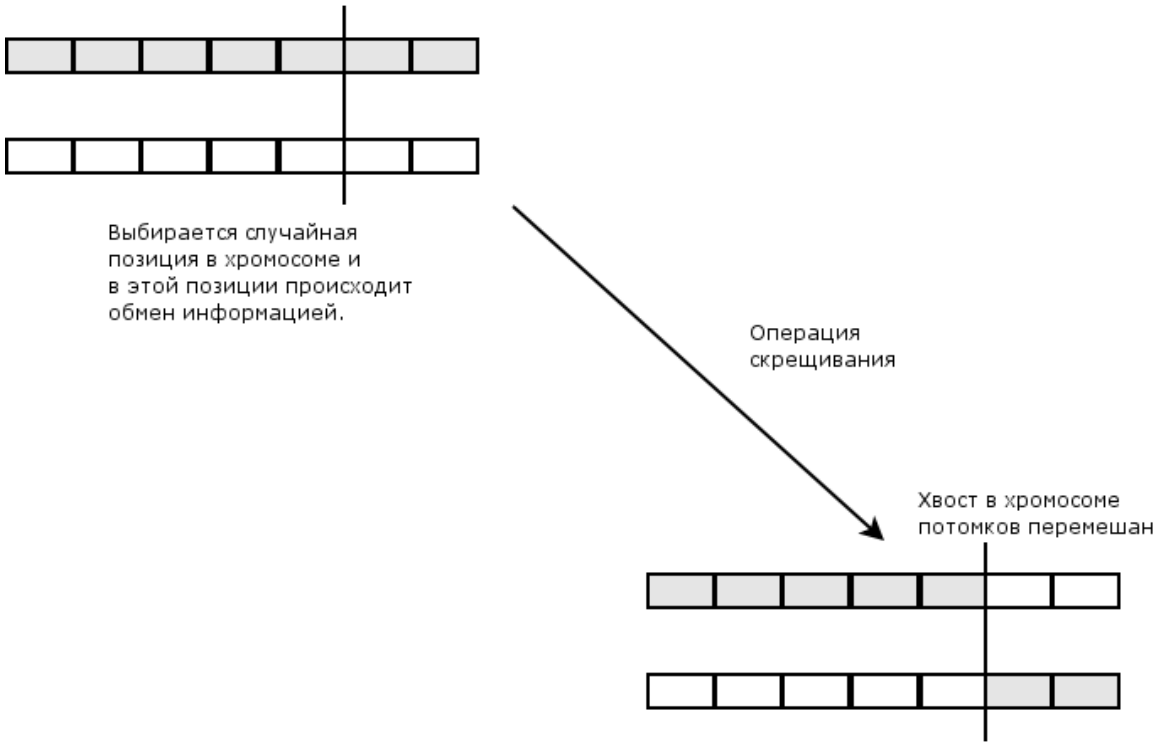

Рисунок 5. Пример операции скрещивания в генетическом алгоритме.

Операция мутации вводится для того, чтобы алгоритм не скатывался в локальный экстремум. Если применять только операцию скрещивания, то в популяции не будет появляться принципиально новых особей, и в конце концов будут исчерпаны все возможные вариации обменов фрагментами хромосом между особями. То есть операция скрещивания либо не будет менять особи, либо будет приводить к появлению особей с более худшим значением функции качества. С целью предотвращения такой ситуации в новую популяцию могут добавлять

некоторое количество случайных хромосом. Однако иногда можно незначительно изменить один или несколько генов в хромосоме, чтобы существенно улучшить значение функции качества для особи. Операция мутации – как раз такое незначительное изменение одного или нескольких генов в хромосоме. Например, в X_i в позиции k случайным образом меняем значение на $x'_{i,k}$. В результате получим новую хромосому $X'_i=(x_{i,1},\ldots,x'_{i,k},\ldots,x_{i,n})$.

Хотя Холландом в книге [54] было доказано, что генетический алгоритм для двоичных хромосом фиксированной длины всегда сходится к максимальному значению функции качества, реально алгоритм может работать довольно долго, поэтому обычно алгоритм останавливают раньше. Обычно бывает достаточно просто улучшить уже имеющееся приближение. В этом случае алгоритм может останавливаться в нескольких ситуациях:

1. по получении некоторого количества популяций,
2. в случае незначительного изменения функции качества для нескольких поколений популяций,
3. в случае получения в популяции особи с приемлемым значением функции качества.

Для создания расписания исполнения работ на многопроцессорной системе генетический алгоритм составляется несколько отличным от классического способом. Здесь в качестве функции качества обычно используют функцию, вычисляющую время работы многопроцессорной системы по рассматриваемому расписанию $time(S)$. Однако в отличии от классического генетического алгоритма здесь ищется не максимум функции качества, а минимум. В качестве хромосомы в данном алгоритме выступает сама запись расписания, где ген – это пара $(proc_j, \text{order\_number})$, привязанная к каждой работе, входящей в расписание. Для операции скрещивания существенных отличий нет, а операция мутации может перемещать работу с одного процессора на другой и менять порядковый номер работы на процессоре, либо оба варианта одновременно.

Впервые о использовании генетического алгоритма для построения расписаний исполнения работ было написано в статье [55]. О использовании генетического алгоритма для построения расписания исполнения работ с учётом задержек на передачу данных на кластере написано в [56]. Генетические алгоритмы также используются для создания расписаний в системах реального времени [57].

3.6. Алгоритм поиска критического пути

Предположим, что часть работ находятся в зависимости от других работ, то есть пока не закончена одна работа, нельзя начинать некоторое количество следующих. Построим граф, где вершины будут соответствовать работам, а ребра – зависимостям между работами. Предполагается, что в графе есть вершина-сток, к которой сходятся остальные вершины. Предполагается, что если мы достигли вершины-стока, то никаких работ больше выполнять не надо, поскольку достигнута цель деятельности. В результате будет получен граф, в чём-то аналогичный графу зависимостей по данным, рассмотренному ранее. Для данного графа определяется понятие критического пути. Критический путь – это такой путь от вершины истока к стоку, который имеет максимальную длительность по времени исполнения входящих в него вершин.

Основная идея заключается в том, что критический путь ограничивает время исполнения параллельной программы, поскольку остальные пути от истока к стоку в графе короче по времени исполнения вершин. Так можно уменьшить время работы параллельной программы если каким-то образом уменьшить время, затрачиваемое

процессорами на обработку вершин критического пути. Для достижения нужного эффекта в первую очередь на процессоры многопроцессорной системы будут назначаться вершины, входящие в критический путь. Сама задача поиска критического пути в ориентированном графе сводится к задаче поиска кратчайшего пути и может быть решена, например, при помощи алгоритма Дейкстры, который обладает сложностью $O(v^2)$, где v – число вершин в графе [63].

Пример применения такой стратегии к гетерогенным вычислительным системам можно найти в статьях [62,64].

3.7. Алгоритм обратного заполнения

Алгоритм обратного заполнения (Backfill) предназначен в первую очередь для «справедливого» распределения ресурсов многопроцессорной системы между работами. Предполагается, что каждая работа требует для своего исполнения не одного процессора, как это было в начальной постановке задачи планирования вычислений, а сразу нескольких процессоров. Каждая работа требует резервирования определённого количества процессорного времени. По мере поступления в многопроцессорную систему для исполнения работы выстраиваются в очередь с учётом приоритета работы.

Для многопроцессорной системы определяется конструкция, отображённая на рисунке 6. По вертикали задаётся шкала процессоров. По горизонтали задаётся шкала времени, где текущий момент времени отображается слева в начале шкалы. Шкала времени ставится в соответствии с очередью работ, требующих своего выполнения. Начало очереди совпадает с началом шкалы времени. Работы здесь представляются прямоугольниками, где по одной шкале откладывается число процессоров, возможно с именами, а по другой – зарезервированное работой процессорное время. Если работа находится в самом начале шкалы времени, то это означает, что она назначена на соответствующее подмножество множества процессоров многопроцессорной системы. На рисунке 6 работы 1 и 2 назначены на процессоры. В очереди могут быть работы, для которых, в текущий момент времени, нет необходимого количества свободных ресурсов. Такие работы следуют за назначенными на исполнение, например, работа 3 на рисунке 6. Вследствие невозможности исполнения работы на многопроцессорной системе будут образовываться окна. Окно – множество свободных процессоров и интервал времени, в который невозможно назначить никакую работу на свободный процессор. На рисунке 6 представлено два окна. Момент запуска работы 3 ограничен работой 1. В результате образуется окно 2, интервал которого во времени совпадает со временем исполнения работы 1 на процессоре. Поскольку работа 3 не блокирует собой всё множество ресурсов, образуется окно 1.

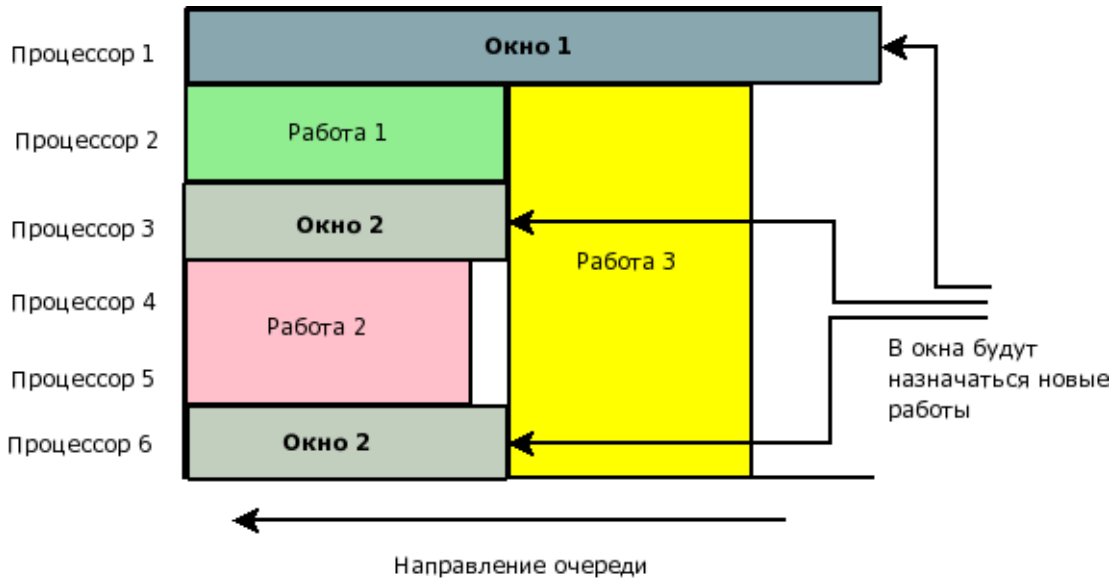

Рисунок 6. Пример работы алгоритма обратного заполнения.

Цель алгоритма – наиболее плотно заполнить образующиеся окна. Для этого среди имеющихся окон выбирается наиболее широкое окно, то есть с максимальным количеством процессоров, и следующие работы, которые попадают в очередь, будут назначаться на процессоры этого окна. Если новая работа не помещается ни в одно из доступных окон, то она помещается в конец очереди. Таким образом работы распространяются в обратную сторону относительно шкалы времени.

Алгоритм обратного распространения довольно часто используется в системах очередей, осуществляющих «справедливый» доступ пользовательским задачам к ресурсам многопроцессорных систем. К таким системам можно отнести, например, Maui. В его состав в качестве одного из алгоритмов управления задачами пользователя используется Backfill [59].

3.8. Алгоритм управления группами работ с прерываниями

Как и в случае с алгоритмом обратного заполнения, алгоритм планирования группами работ (Gang) осуществляет распределение ресурсов многопроцессорной системы между множеством работ. Основное назначение данного алгоритма – распределять ресурсы между группами работ. Предполагается, что работы объединены в группы по степени важности, то есть приоритет работы p_i приписан сразу же для всей группы работ. Работы одной группы разделяют множество процессоров таким же образом, как и в случае алгоритма Backfill, однако допускается прерывание работ, если на многопроцессорную систему поступает группа работ с большим приоритетом. В этом случае страницы оперативной памяти на многопроцессорной системе, занимаемые работой с меньшим приоритетом, сохраняются во внешнюю память, где они образуют очередь отложенных работ. Из полученной очереди работы восстанавливаются согласно приоритетам групп и порядку сохранения во внешнюю память. Сперва будут восстанавливаться работы из группы с наибольшим приоритетом в том порядке, в котором они были отложены, а затем произойдёт переход к группам с меньшим приоритетом.

О такого рода способе распределения ресурсов многопроцессорной системы написано в статьях: [13,60]. Данный алгоритм используется на SMP системах, в том числе на IBM pSeries 690. Алгоритм Gang включён в систему управления очередями IBM LoadLeveler [61].

3.9. Особенности алгоритмов планирования вычислений в «PARUS»

Анализ приведённых выше стратегий планирования показал, что в «PARUS» целесообразно использовать списочные алгоритмы и генетический алгоритм. В текущей реализации в «PARUS» представлено 3 алгоритма планирования вычислений. Все три алгоритма реализованы таким образом, чтобы учитывать гетерогенность многопроцессорной системы с точки зрения производительности процессоров и с точки зрения коммуникаций.

Первый алгоритм статически, то есть заранее, до момента запуска параллельной программы, строит расписание исполнения внутренних работ программы на многопроцессорной системе. Для создания статического расписания из рассмотренных выше подходят: алгоритм имитации отжига, генетический алгоритм и алгоритм поиска критического пути. Как было сказано выше, алгоритм имитации отжига может сваливаться в локальный минимум. Алгоритм поиска критического пути находит слишком грубое решение.

По этим причинам статический алгоритм построения расписания в «PARUS» реализован на основе описанного выше генетического алгоритма. Однако основное отличие реализованного генетического алгоритма от других реализаций генетических алгоритмов, например, от алгоритма, приведённого в [56], заключается в сложности функции качества. При вычислении функции качества тщательно определяется, сколько времени атомарная работа параллельной программы будет исполняться процессором с учётом времени, затрачиваемого на передачу сообщений между процессорами. Такой тщательный учёт времени приводит к существенному усложнению функции качества, увеличению времени работы алгоритма, но зато способствует построению более сбалансированного расписания. В предложенном алгоритме реализованы более сложные схемы мутации, отбора и скрещивания по отношению к генетическим алгоритмам, описанным в [55,56,57], что в совокупности с возможностью настройки параметров может давать лучшую сходимость.

Второй алгоритм динамически, уже в процессе исполнения параллельной программы, назначает работы по MPI-процессам. Данный алгоритм можно отнести к классу списочных алгоритмов. Он является совмещением RR алгоритма с алгоритмом назначения кратчайшей работы вперёд. Основное преимущество такого алгоритма заключается в скорости его срабатывания и низкой вычислительной сложности, что очень важно для назначения работ на процессоры в динамике. Данный алгоритм обходит по кругу множество процессоров многопроцессорной системы и назначает на очередной свободный процессор ту работу из списка готовых к исполнению работ, которая выполнится на данном процессоре за кратчайшее время. Идея этого алгоритма близка к идее алгоритма Backfill. Он, так же, как и Backfill, стремится полностью заполнить работами многопроцессорную систему. К сожалению, идея использовать более продвинутый алгоритм Gang не применима, поскольку в «PARUS» нет прерывания работ.

Третий алгоритм, так же, как и второй, работает в динамическом режиме. Однако в отличие от предыдущего алгоритма, данный алгоритм способен учитывать подсказки пользователя по назначению работ на процессоры. Об использовании такого рода подхода при планировании процессов в операционной системе написано в [65], однако упоминания в литературе об использовании такого подхода при планировании вычислений на гетерогенных многопроцессорных системах найти не удалось.

4. Система «PARUS»

4.1. Краткое описание

На вход системе «PARUS» поступает либо ориентированный ациклический граф, представление которого будет рассмотрено ниже, либо С-программа, которая затем всё равно преобразуется в ориентированный граф.

Алгоритм решаемой задачи представляется в виде ориентированного графа, где в вершинах сосредоточены вычислительные операции (действия над данными), а рёбра задают зависимость по данным. При этом дуга направлена от вершины источника данных к вершине, принимающей данные. Предполагается, что вершина, принимающая данные, будет их как-то использовать, на основе этих данных вырабатывать данные, которые будут использоваться другими вершинами графа. Таким образом, программа описывается как сеть с вершинами – истоками, где чаще всего происходит чтение входных данных из файлов внутренних вершин, в которых происходит обработка данных, а также вершин – стоков, где обычно происходит запись результатов в файлы. С точки зрения эффективности выполнения построенной таким образом граф должен обладать некоторыми особенностями. Вершины графа должны являться полновесными, тяжёлыми в вычислительном смысле, фрагментами кода, вплоть даже до целых отдельных независимых программ. Иначе накладные расходы на передачу данных могут свести на нет весь выигрыш от обширного параллелизма, который, возможно, появится в графе после создания большого количества независимых между собой легковесных вершин. Граф выстраивается по уровням. Номер уровня – максимальная длина пути от одной из вершин истоков до текущей вершины. Вершины определённого уровня могут получать данные только от вершин, отнесённых к уровню с меньшим номером. Вершины графа на каждом уровне независимы между собой и могут быть исполнены параллельно. Таким образом, определённый выше граф задаёт граф-программу. Общие принципы системы «PARUS» обсуждены в [81].

Описание графа содержится в текстовых файлах, которые можно подать на вход набору утилит, осуществляющих преобразование граф-программы в исходный код на C++ с вызовами MPI-функций. Форматы файлов были опубликованы в статье [14] и будут рассмотрены ниже.

После преобразования в код на C++ с MPI-вызовами полученный исходный файл компилируется совместно с компонентом времени запуска, который осуществляет непосредственное управление назначениями вершин графа по процессорам, а также осуществляет контроль передаваемых по рёбрам данных. В текущий момент реализовано 3 режима планирования исполнения вершин графа на процессорах. Динамический режим – выбор вершины для исполнения осуществляется в процессе исполнения параллельной программы на основе имеющихся сведений о текущем состоянии многопроцессорной системы. Статический режим – решение о порядке назначений вершин графа по процессорам принимается ещё до запуска программы. Комбинированный – решение принимается из соображений динамики, но в случае затруднения в выборе планировщик обращается к файлу с подсказками пользователя. Описание динамического режима было опубликовано в статье [18]. Для построения статического расписания назначений вершин графа по процессорам была создана программная реализация генетического алгоритма. О данной реализации говорится в публикации [17].

Каждой вершине и ребру графа приписан вес. Вес вершины характеризует вычислительную сложность вершины, выраженную как отношение к эталонным операциям. Под эталонной операцией можно понимать, например, время решения эталонной системы линейных алгебраических уравнений. Вес ребра характеризует

объём данных, которые нужно передать по данному ребру от одной вершины графа к другой. Эта информация используется при назначении вершины на процессор.

Была разработана система тестирования многопроцессорной системы. О системе тестирования многопроцессорной системы рассказано в публикации [15]. Создан тест, позволяющий определить производительность процессоров многопроцессорной системы. В диссертационной работе создана система тестов, помогающая прогнозировать поведение коммуникационной среды многопроцессорной системы относительно размера передаваемого сообщения, номера процессора и уровня «шума». Подробнее тесты будут рассмотрены ниже.

С целью облегчения навигации в данных, получающихся после проведения тестирования многопроцессорной системы, был создан графический интерфейс на Java. Описание графического интерфейса для системы тестирования приведено в статье [15]. Разработанный интерфейс позволяет отображать 4-х мерную модель сети.

Также создано графическое средство для визуального представления граф-программы и её редактирования. В данное программное средство встроена возможность редактирования расписания исполнения программы на многопроцессорной системе. Об этом написано в работе [16].

С целью предоставления возможности исследования С-программы на возможность распараллеливания отдельных участков кода было создано специальное приложение, которое анализирует зависимости по данным в С-программе и строит по ним граф зависимости по данным в формате, понимаемом системой «PARUS». Данное программное средство можно считать прототипом автораспараллеливающего компилятора. Используемый алгоритм опубликован в статье [17].

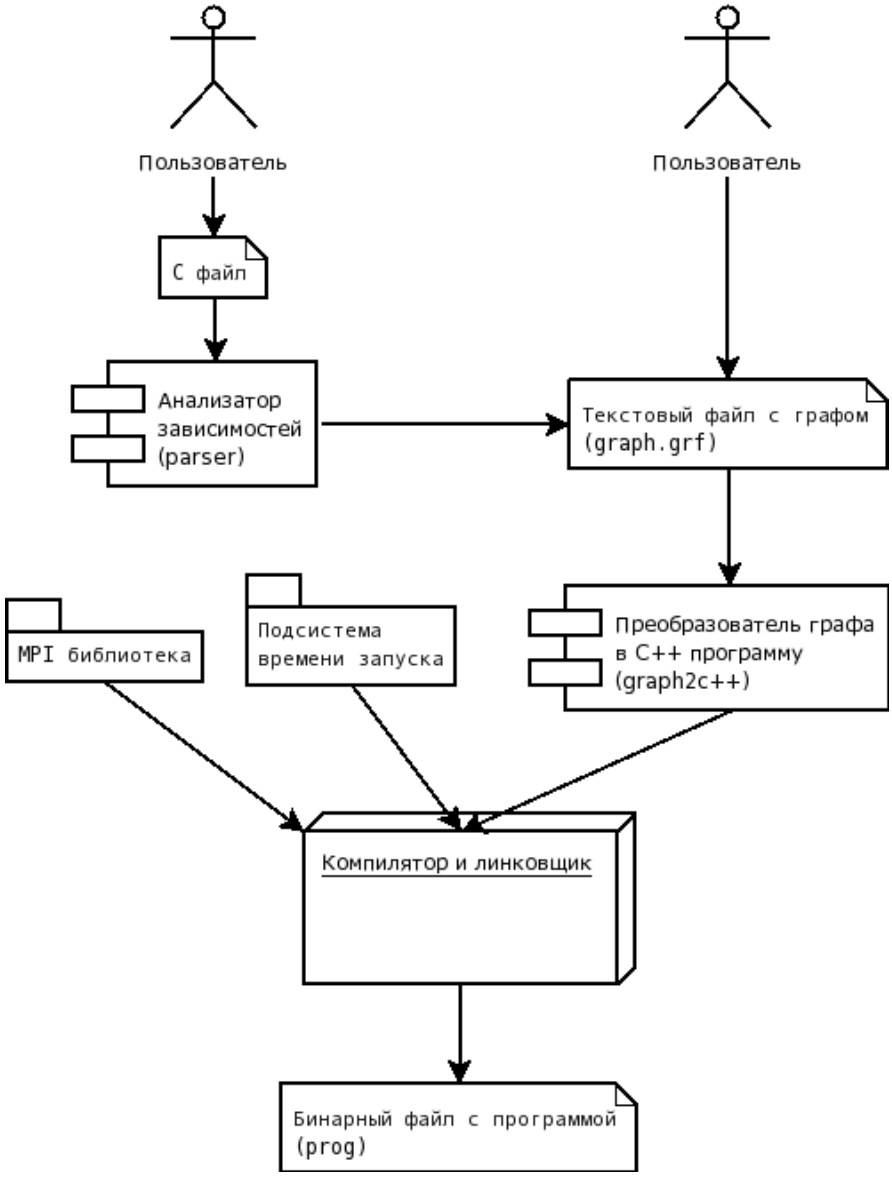

Рисунок 7. Компоненты и связи (преобразование в C++ программу).

На рис. 7 приведён набор инструментов системы «PARUS» и связи между ними. Пользователю предоставляется возможность создать параллельную программу двумя различными способами. Первый способ – предлагается самостоятельное создание текстового файла с граф-программой. Соответствующий файл с графом можно также получить, написав специализированный генератор кода. Второй способ – запустить анализатор зависимостей по данным и получить по программе на языке С граф зависимости между отдельными переменными в программе. В любом случае, вне зависимости от того, каким образом был получен текстовый файл с граф-программой, он отправляется на вход преобразователю графа в C++ код с MPI-вызовами. На рисунке 7 представлены следующие программные компоненты:

parser – анализатор зависимостей в С-программе;

graph2c++ – преобразователь графа в C++ код с MPI-вызовами.

Далее рассматриваются связи между компонентами системы «PARUS», актуальные для процесса исполнения граф-программы на многопроцессорной системе (см. рис. 8). Для работы внутреннего планировщика, встроенного в созданную C++ программу, необходимо получить сведения о производительности

процессоров и задержках при передаче данных по коммуникационной среде многопроцессорной системы. Сведения собираются один раз в момент установки системы «PARUS» или в случае изменения физической конфигурации многопроцессорной системы. Сбор сведений осуществляется программными компонентами ***proc_test*** и ***network_test***. Система тестирования многопроцессорной системы устроена таким образом, чтобы учитывать гетерогенность коммуникаций, включая физическую структуру коммуникационной среды, но на этапе планирования вычислений позволять учитывать только результаты тестирования, с тем, чтобы достичь независимости этапа планирования от архитектурных особенностей многопроцессорной системы. Результаты тестирования можно передать как непосредственно внутреннему планировщику подсистемы времени запуска граф-программы, так и внешнему планировщику, построителю расписаний, ***graph2sch***. По причине большой длительности процедуры построения расписания предусмотрен режим полностью динамического назначения вершин графа по процессорам — связь «построитель расписаний – параллельная программа» не является обязательной. Факт необязательности использования расписания на рисунке 8 отображён пунктирной стрелкой.

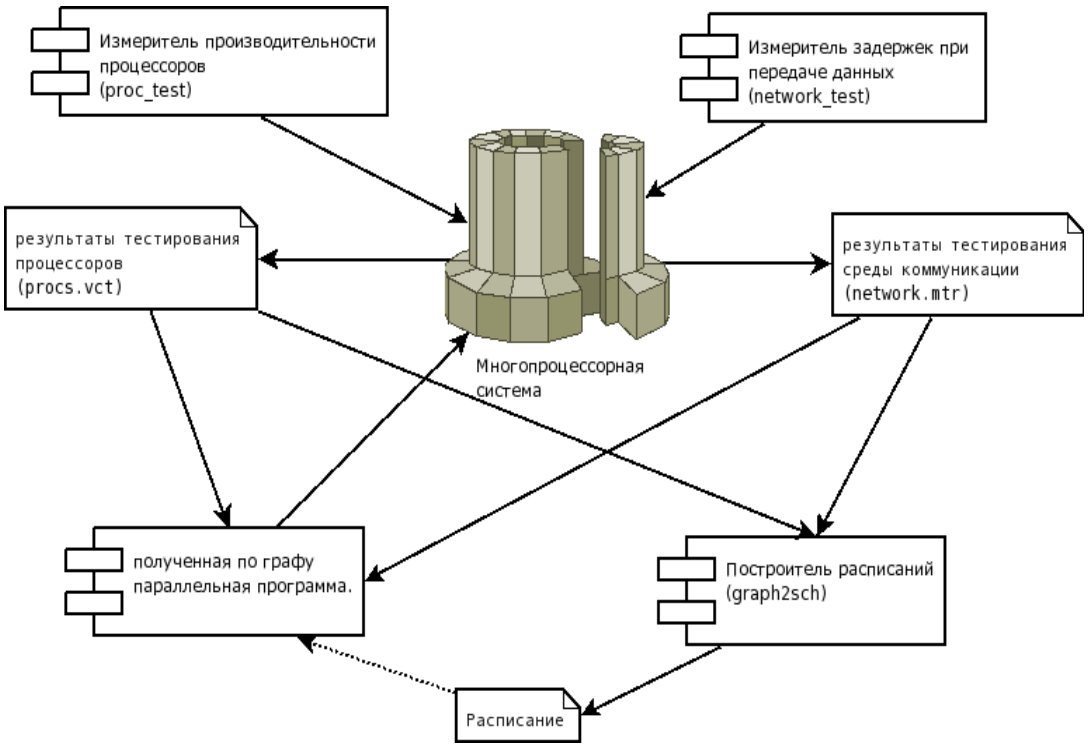

Рисунок 8. Компоненты и связи (необходимое для запуска граф-программы, преобразованной в MPI-программы).

4.2. Механизм преобразования графа зависимости в параллельную программу

После компиляции C++ кода с вызовами MPI-функций граф-программа становится MPI-программой. MPI-программа строится как набор MPI-процессов, взаимодействующих между собой через передачу сообщений. Обычно каждый MPI-процесс, составляющий MPI-программу, привязывается к своему процессору многопроцессорной системы. MPI-процесс может быть также назначен на свой узел в кластере. Под узлом кластера будем понимать отдельный, возможно состоящий из нескольких микропроцессоров, компьютер, который работает под управлением своей

собственной операционной системы и объединён сетью с другими узлами кластера. Среди всех MPI-процессов выделяется один, который в дальнейшем будет выполнять роль координатора в граф-программе. Остальные MPI-процессы исполняют код, приписанный вершинам графа.

Опишем представление параллельной программы графом зависимости по данным. На этапе распараллеливания задачи человек стремится компоненты задачи, наиболее сильно связанные между собой, расположить как можно ближе друг к другу, минимизируя связи между получившимися группами компонентов. Система «PARUS» позволяет реализовывать такой подход. Как было сказано выше, программа представляется как текстовый файл с описанием вершин и рёбер графа, а также с описанием некоторой другой информации (Описание форматов текстовых файлов приведено в приложениях).

В MPI-программе вершины граф-программы становятся функциями, в которых могут быть описаны свои переменные и которые могут использовать описанные глобальные переменные. Каждый MPI-процесс хранит свою собственную копию глобальной переменной. Вершины могут менять переменные и передавать изменённые значения другим вершинам граф-программы по ребрам графа.

Вершина графа в файле представлена следующим образом. Каждой вершине присвоен собственный уникальный номер, который затем используется в расписании исполнения. Для каждой вершины указываются три фрагмента с исходным кодом на языке программирования C++: «голова вершины», «тело вершины» и «хвост вершины». В «голове вершины» сосредоточенны определяемые пользователем действия инициализационного характера, которые не зависят от других вершин. Эти действия всегда предшествуют приёму данных по рёбрам, входящим в вершину, и обычно это действия, подготавливающие вершину к приёму данных от других вершин. «Тело вершины» ответственно за обработку данных, обычно с учётом принятой информации от других вершин. В «хвосте вершины» производятся финальные действия: закрываются открытые в «голове вершины» и «теле вершины» файлы, освобождается динамическая память и тому подобное. Каждой вершине приписан её вес. Вес выражает суммарную вычислительную сложность кода, приписанного вершине. Он определяется как доля эталонных операций, которую нужно выполнить, чтобы затратить столько же времени, сколько было бы затрачено на исполнение кода вершины. Для каждой вершины указывается её уровень относительно источника. Иногда с точки зрения оптимизации времени выполнения полученной параллельной программы выгодно один из уровней в графе разбить на несколько подуровней. Для каждой вершины графа указываются списки номеров входящих в неё и исходящих из неё рёбер. На рисунке 9 приведён пример текстового представления вершины графа.

```
<NODE_BEGIN>
  number 1
  type 1
  weight 100
  layer 2
  num_input_edges 1
  edges ( 1 )
  num_output_edges 1
  edges ( 2 )
  head "head"
  body "node"
  tail ""
 <NODE_END>
```

Рисунок 9. Пример вершины графа.

В файле, наряду с вершинами, описываются и рёбра графа. Рёбра, по существу, содержат информацию о синхронизируемых ребром значениях переменных, где переменные описаны либо глобально, либо в связанных ребром графа вершинах. К ребру приписана весовая метка, характеризующая объём передаваемой по ребру информации в байтах. Для каждого ребра указывается посылающая вершина – источник данных, и принимающая вершина – приёмник данных. Данные синхронизируются: состояние данных в вершине – приёмнике становится идентичным состоянию данных в вершине – источнике. По аналогии с вершинами, для каждого ребра указывается его уникальный для графа номер. Номер ребра указывается в вершинах и используется в дальнейшем при исполнении полученной параллельной программы.

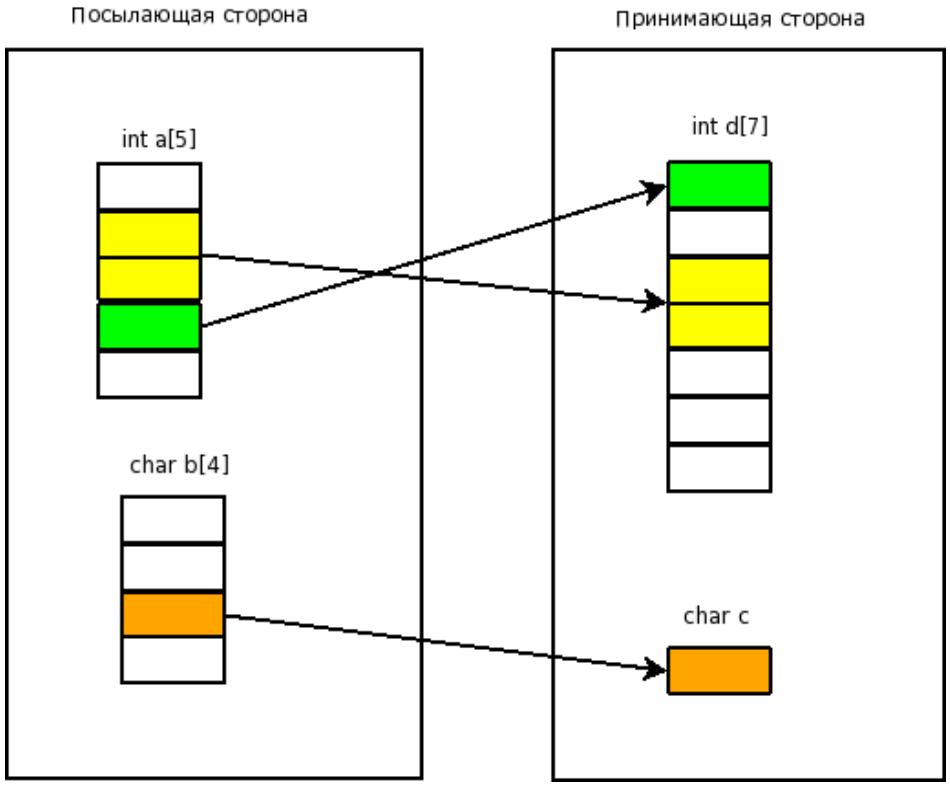

Рисунок 10. Внутренняя структура ребра граф-программы.

С каждым ребром графа сопоставляется набор «чанков». На рисунке 10 представлена иллюстрация процесса выделения чанков на множестве переменных MPI-программы для ребра графа. Чанк задаёт фрагмент в области памяти MPI-процесса, который будет синхронизирован с аналогичным фрагментом в памяти другого MPI-процесса. Один чанк связан с определённой переменной в вершине – источнике данных или в вершине – приёмнике данных. Несколько чанков могут быть связаны с одной и той же переменной. Чанк задаётся указателем и смещениями, которые определяют левую и правую границы фрагмента в области памяти. Имена переменных, которые синхронизируются через ребро графа, могут отличаться с принимающей и посылающей сторон. Всё множество чанков, описанных в текстовом файле для ребра графа, образуют одно ребро графа. В MPI-программе ребро представляются как одно неблокированное MPI-сообщение. В случае, когда принимающая и посылающая вершина оказались на одном MPI-процессе, синхронизация данных производится с помощью механизма упаковки и распаковки данных. Механизм упаковки и распаковки данных определён в MPI стандарте. На рисунке 11 приведён пример текстового представления ребра графа.

```
<EDGE_BEGIN>
  number 2
  weight 2
  type   GRAPH_NONE
  num_var 1
  num_send_nodes 1
  send_nodes ( 2 )
  num_recv_nodes 1
  recv_nodes ( 5 )
  <SEND_BEGIN>

   <CHUNK_BEGIN>
     name "*data_out"
     type GRAPH_DOUBLE
     left_offset "0"
     right_offset "window_size+F_LEN"
   <CHUNK_END>
  <SEND_END>
  <RECIEVE_BEGIN>

   <CHUNK_BEGIN>
     name "*data_out"
     type GRAPH_DOUBLE
     left_offset "0"
     right_offset "window_size"
   <CHUNK_END>
  <RECIEVE_END>
<EDGE_END>
```

Рисунок 11. Пример ребра графа.

Кроме вершин и рёбер, для граф-программы целиком указываются ещё фрагменты кода «голова графа», «корень графа» и «хвост графа». Наличие «головы графа» связано с тем, что граф-программа преобразуется в C++ программу. В «голове графа» указываются все необходимые .h файлы, а также описания всех функций, которые потом понадобятся при исполнении вершины графа. В «голове графа» также содержатся описания глобальных переменных. Однако в силу

специфики MPI-программы, на каждом MPI-процессе окажется свой собственный экземпляр глобальной переменной. «Корень графа» – код, который представляется как отдельная функция. Исполнение граф-программы начинается с исполнения кода «корня графа». Данная функция вызывается на всех MPI-процессах MPI-программы, кроме процесса координатора, ещё до вызова на соответствующем процессе любой вершины графа. Обычно «корень графа» содержит код, настраивающий пространство памяти MPI-процесса для работы с граф-программой вообще. Это действия, результат которых будет виден одинаковым образом всеми вершинами граф-программы. Как правило, это действия инициализационного характера; например, чтение одной и той же матрицы в память каждого MPI-процесса. «Хвост графа» (фрагмент кода, также представленный в виде функции) выполняется после завершения исполнения всех вершин графа на всех MPI-процессах MPI-программы, за исключением процесса координатора. «Хвост графа» предназначен для завершающих программу действий. Например, в «хвосте графа» можно закрыть файл, открытый в «корне графа».

После преобразования графа в код на C++ вершины графа представляются как функции с пустыми аргументами. Все необходимые вершине данные получаются либо из инициализированных заранее в «корне графа» глобальных переменных, либо через передачу их от других вершин графа по рёбрам. Автоматическая сборка исходного кода вершины графа в C++ функцию производится следующим образом. Сперва идёт декларация служебных переменных, затем идёт код «головы вершины», дальше вставляется MPI-код, ответственный за приём данных по входящим в вершину рёбрам, затем идёт «тело вершины», собственно обработка данных, затем код, ответственный за архивацию в память данных, которые будут переданы этой вершиной графа другим вершинам, и наконец идёт «хвост вершины», на этом функция заканчивается (см. рис.12).

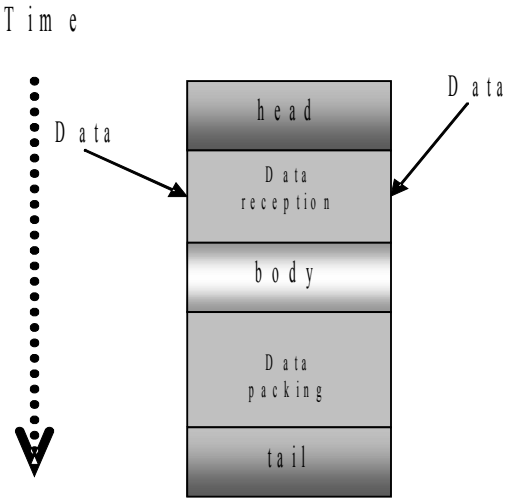

Рисунок 12. Вершина графа.

Действия, связанные непосредственно с передачей данных и управлением вызовами вершин на процессорах, не видны пользователю на этапе создания текстового файла с граф-программой. Они появляются только на этапе автоматического преобразования текстового файла с граф-программой в C++ код с вызовами MPI-функций. Тем самым достигается независимость описания граф-программы от особенностей многопроцессорной системы.

Полученный файл с вершинами компилируется и связывается с подсистемой, осуществляющей управление вызовами вершин на MPI-процессах и назначающей

передачи данных между MPI-процессами по рёбрам графа.

4.3. Организация передачи данных между вершинами графа

Различным MPI-процессам в параллельной программе отводятся различные роли. MPI-процессу с номером 0 отводится роль координатора. Остальные MPI-процессы исполняют вершины и концентрируют данные, наработанные вершинами. Работа 0 процесса (координатора) имеет прямое отношение к алгоритму назначения вершин графа по MPI-процессам и будет рассмотрена позднее. Остановимся на работе остальных процессов.

На этапе начала выполнения параллельной программы все MPI-процессы помечаются как занятые. Поэтому первое, что делает процесс, – это посылает координатору блокированное сообщение о том, что процесс в текущий момент не выполняет никакой работы и свободен. Это делается при помощи сообщения с меткой PX_NODE_CLEAN_TAG. В ответ процесс получает сообщение о режиме своей работы. То есть во всех процессах, кроме 0, выставляется блокированный приём сообщения с меткой PX_REGIME_TAG, где в сообщении могут прийти следующие значения:

1. PX_STOP – означает, что данным MPI-процессом далее не будет производится никаких действий и он должен быть завершен, что собственно в дальнейшем и происходит;
2. PX_EDGE_REGIME – означает, что процесс переводится в режим передачи данных по сохранённым в нём рёбрам другим MPI-процессам;
3. PX_WORK – означает, что процесс переводится в режим исполнения вершины графа.

Рассмотрим для начала режим PX_WORK. После получения соответствующего сообщения MPI-процесс запрашивает у координатора номер вершины, которую нужно выполнить данным MPI-процессом. Запрос осуществляется путём инициализации блокированного приёма сообщения с меткой PX_NODE_QUESTION_TAG, где по окончанию приёма в памяти процесса оказывается номер вершины, которую необходимо выполнить на процессе. Далее идёт вызов функции той вершины графа, номер которой получен от координатора.

Внутреннее устройство вершины таково, что в функцию вершины, помимо кода, определённого в соответствующих полях «head», «body», «tail», вставляется специальный код, необходимый для организации передачи данных между вершинами графа. Первая порция кода вставляется непосредственно в начале функции. Это инициализация служебных переменных, отвечающих за передачу данных. Затем идёт «голова вершины», код которой подставляется из соответствующего файла. Далее следует код, ответственный за приём данных, входящих по рёбрам в вершину.

На начальном этапе происходит выделение памяти для всех служебных массивов, необходимых для приёма данных по ребру. В частности, создаётся массив с номерами входящих в вершину рёбер, а также место под массив с информацией, на каких именно MPI-процессах находятся данные для входящих в вершину рёбер. После создания всех служебных буферов памяти происходит отправка запроса координатору о месте нахождении данных для рёбер (сообщение с меткой PX_RECV_INFO_TAG). Запрос осуществляется с помощью отправки MPI-процессом блокированного сообщения, где в теле указан номер вершины.

Далее выставляется блокирующий приём сообщения. Это приём массива, совпадающего по размерности с числом входящих в вершину рёбер, в который

заносятся номера MPI-процессов, содержащих данные для соответствующих рёбер граф-программы.

Следующим шагом происходит инициализация приёма данных по рёбрам граф-программы, в том случае, если данные расположены в другом MPI-процессе. Приём данных производится с помощью неблокированных обменов, поддерживаемых MPI. Если данные расположены на том же MPI-процессе, то вместо инициализации приёма данных происходит их распаковка из памяти. Затем MPI-процесс ожидает завершения приёма данных по любому из неблокированных обменов. По окончании приёма происходит распаковка соответствующих данных из буфера в переменные программы. Распаковка данных производится в соответствии с механизмом распаковки, определённым в стандарте MPI.

По окончанию всех операций, связанных с приёмом данных по ребру графа, производится отправка блокированного сообщения координатору с меткой PX_RECV_EDGE_FINISHED_TAG и номером ребра.

Особенности реализации процесса приёма данных по рёбрам графа таковы, что в момент исполнения вершины графа порядок прихода данных по рёбрам не определён. Иными словами, если в вершину входит несколько рёбер, то в процессе написания «тела вершины» и «хвоста вершины» нельзя считать, что передача по рёбрам ведётся в каком-либо определённом порядке. Схему обмена сообщениями можно видеть на рисунке 13.

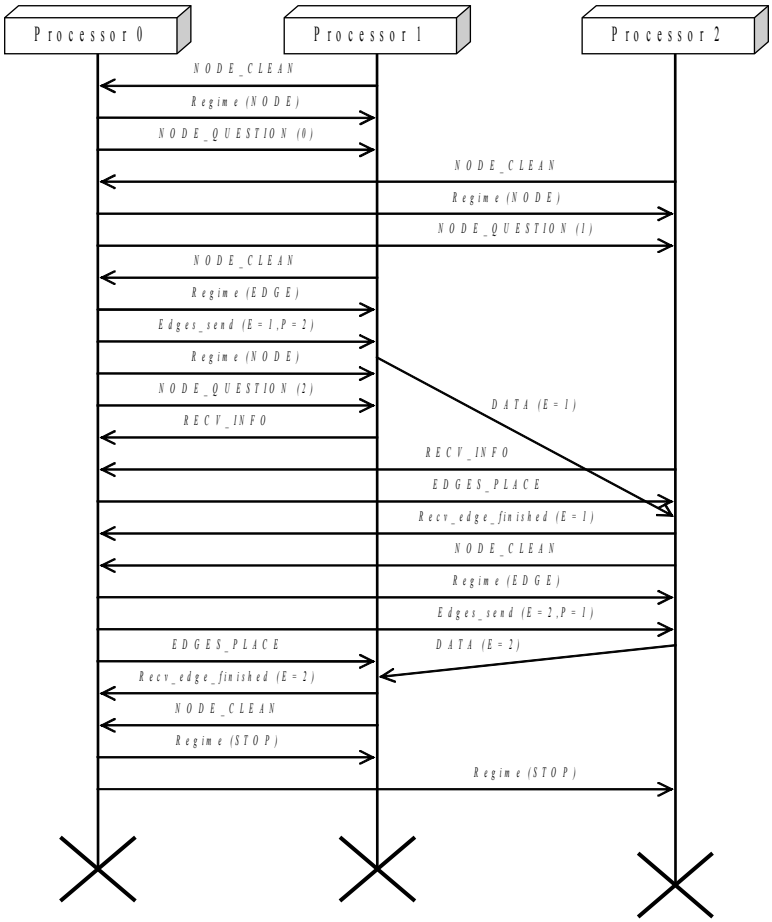

Рисунок 13. Пример протокола обмена сообщениям между MPI-процессами.

По завершении приёма данных происходит их обработка «телом вершины». «Тело вершины» не только обрабатывать данные, поступившие в вершину по рёбрам, но и нарабатывать новые данные. Обработанные данные и новые наработанные

данные затем либо будут отправлены по рёбрам другим вершинам, либо будут записаны как результаты работы программы в файлы.

После исполнения «тела вершины» происходит выделение памяти для буферов, в которые будут скопированы данные для передачи по рёбрам граф-программы другим вершинам. По завершении этого процесса необходимые данные копируются в соответствующие буфера. Данное действие производится при помощи механизма упаковки данных, описанных в стандарте MPI.

Затем выполняется «хвост вершины» с действиями по очистке памяти и закрытию открытых в «теле вершины» и «голове вершины» файлов. На этом работа функции вершины завершается. После этого координатору отсылается блокированное сообщение с меткой PX_NODE_CLEAN_TAG, и MPI-процесс инициализирует блокированный приём сообщения о дальнейшем режиме работы.

Рассмотрим режим работы MPI-процесса, связанный с передачей данных по рёбрам другим вершинам, функции которых запущены на других MPI-процессах.

Передача данных по рёбрам между вершинами в системе «PARUS» реализована через неблокированные MPI-обмены. Так сделано для того, чтобы передача данных, связанных с рёбрами, проходила одновременно с вычислениями в вершинах. Такая схема, очевидно, должна минимизировать простои, вызванные необходимостью передавать данные между MPI-процессами.

Для реализации схемы с неблокированными обменами в каждом MPI-процессе, исключая процесс координатора, существует 2 списка: список рёбер, для которых не инициализирована передача данных другим в другие MPI-процессы, а также список активных рёбер, для которых инициализирована неблокированная передача данных. После каждого приёма сообщения о режиме работы происходит проверка: не закончен ли какой-либо неблокированный обмен, связанный с каким-либо ребром. Если обнаружен завершенный обмен, то буфера, связанные передачей данных, очищаются, и ребро удаляется из обоих списков. Рис. 14. иллюстрирует описанный процесс.

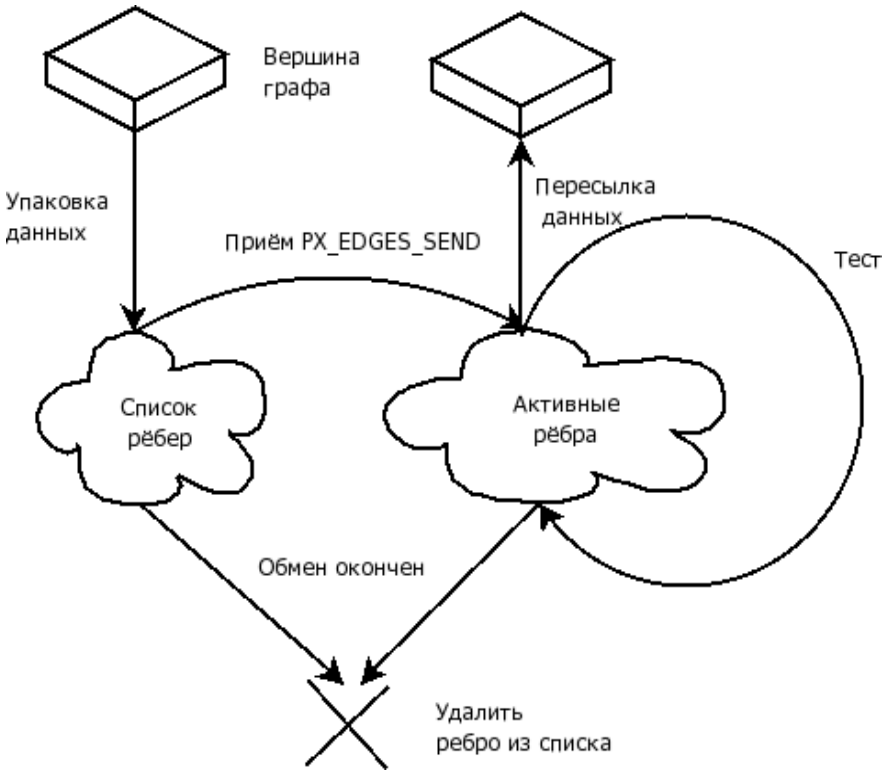

Рисунок 14. Миграция данных внутри MPI-процесса.

После приёма сообщения с режимом PX_EDGE_REGIME происходит инициализация блокированного приёма сообщения с меткой PX_EDGES_SEND_TAG. В сообщении приходит информация с номером ребра и номером MPI-процесса, в который нужно передать данные, связанные с этим ребром. Далее для данного ребра инициализируется неблокированная передача данных, и ребро добавляется в список посылаемых рёбер. Иллюстрацию протокола обмена сообщениями можно видеть на рисунке 13.

4.4. Работа координирующего MPI-процесса

В исполняющийся параллельной программе 0 MPI-процесс выделяется как управляющий (координатор). Данный процесс принимает решения о том, какую вершину в какой момент времени назначить на исполнение, то есть какой MPI-процесс будет исполнять вершину и по какому ребру передавать данные. С этой целью на координаторе на всё время жизни параллельной программы запускается функция px_demon, которая собственно и занимается планированием вычислений.

В данной функции присутствует несколько списков и массивов:

1. Массив layer – множество вершин, приписанных к одному из уровней в графе программы;
2. Массив processors – в данный массив заносятся номера вершин графа, которые в текущий момент исполняются соответствующим MPI-процессом. Номера вершин всегда неотрицательные;
3. Массив loading – показывает, занят или свободен в текущий момент соответствующий MPI-процесс;
4. Массив waiting_recv – массив, показывающий, ожидает ли вершина, будучи запущена на соответствующем MPI-процессе, приёма данных по входящим в неё рёбрам;
5. Список req – список вершин, которые готовы к назначению их на исполнение в какой-либо MPI-процесс;
6. Список sending_edges – список рёбер, по которым в текущий момент времени идёт передача данных;
7. Список unsent_edges – список рёбер, по которым вершины уже наработали данные, но они по каким-то причинам ещё не инициализированы для передачи. На самом деле это не один список, а набор списков. Количество списков в наборе соответствует количеству MPI-процессов в программе.

В начальный момент функцией выясняется число уровней в графе, это существенно, поскольку номера уровней задаются пользователем из собственных соображений, а не автоматически. Дело в том, что меняя номер уровня, можно управлять процессом загрузки вершин на процессоры. Например, если пользователь обладает сведениями об устройстве многопроцессорной системы, он может попытаться оптимизировать передачу данных между вершинами, если известно, что реальный уровень в графе очень большой и не помещается целиком на доступное пространство процессоров. Иллюстрация этой ситуации приведена ниже при анализе особенностей реализации параллельного перцептрона.

В случае если известно, что часть вершин слоя сильно связана с некоторыми вершинами следующего слоя и есть ещё одна группа вершин, также сильно связанная, но уже с другими вершинами следующего слоя, то такие сильно связанные вершины внутри группы стоит отнести к соседним уровням, а сами

группы разделить и «разнести» на разные уровни.

Изначально все MPI-процессы помечаются как занятые; предполагается, что ни один из них не находится в режиме ожидания приёма данных по рёбрам (это достигается заданием начальных значений в массивы loading, processors, waiting_recv).

После этого инициализируется бесконечный цикл, условием остановки которого является одновременное выполнение нескольких условий:

- Отсутствие ещё не запущенных на MPI-процессах вершин, то есть список req должен быть пустым;
- Не должно быть рёбер, по которым в текущий момент времени производится передача данных, список sending_edges должен быть пустым;
- Не должно быть рёбер, по которым ещё не инициализирован обмен, список unsent_edges должен быть пустым;
- Все MPI-процессы, кроме процесса координатора, должны быть свободны, в массиве loading на 0 месте должна стоять 1, в остальных ячейках данного массива должны быть нули.

Первое действие в цикле - это выставление ловушки, которая ловит все сообщения от других MPI-процессов к координатору и, в зависимости от метки сообщения, производит определённое действие. Метки сообщений таковы:

1. PX_NODE_CLEAN_TAG – означает, что MPI-процесс, приславший сообщение, освободился от выполняемой им работой;
2. PX_RECV_INFO_TAG – означает, что MPI-процесс исполняет в текущий момент вершину графа и ожидает от координатора информации о местонахождении входящих в вершину рёбер;
3. PX_RECV_EDGE_FINISHED_TAG – означает, что вершина, запущенная на MPI-процессе, приславшем сообщение, получила данные по соответствующему входящему в неё ребру.

В случае получения PX_NODE_CLEAN_TAG в массиве loading производится тривиальное действие: в ячейке loading, соответствующей номеру MPI-процесса, приславшего сообщение, пишется 0. Для координатора это означает, что процесс освободился.

В случае получения PX_RECV_INFO_TAG в массиве waiting_recv выставляется единица. В случае получения PX_RECV_EDGE_FINISHED_TAG соответствующее ребро удаляется сразу из обоих списков: sending_edges и unsent_edges.

Далее, в случае завершения вершины графа на MPI-процессе, что соответствует неотрицательному значению в массиве processors и 0 в массиве loading, происходит добавление исходящих рёбер вершины к списку ещё не посылаемых рёбер – unsent_edges. Добавляется номер ребра и номер MPI-процесса, на котором находятся данные рассматриваемого ребра. Для избежания повторного добавления ребра в массив processors в нужное место заносится отрицательное значение.

После этого происходит разбор ситуации с рёбрами графа. Для этого вызывается функция try_to_send, которая выясняет, есть ли потребность в инициализации передачи данных, и инициализирует соответствующую передачу для одного из MPI-процессов по одному из рёбер. Данная функция просматривает список unsent_edges и для каждого ребра в этом списке ищет в массиве processors подходящую вершину. Это эквивалентно совпадению номеров, входящих в вершину

рёбер, с одним из рёбер из списка unsent_edges, при условии, что MPI-процесс действительно исполняет вершину; то есть в массиве loading на соответствующем месте стоит 1.

Если такая вершина существует и её расположение отличается от расположения данных по входящему в неё ребру, то производится инициализация передачи данных по ребру графа. Это происходит следующим образом:

- От управляющего процесса (координатора), на котором хранятся, данные передаваемые по ребру, передаётся сообщение с меткой PX_REGIME_TAG и значением PX_EDGE_REGIME, что означает перевод MPI-процесса в режим передачи данных по ребру;
- Далее вершине в сообщении с меткой PX_EDGES_SEND_TAG передаётся номер ребра, по которому необходимо инициировать неблокированную передачу данных, и номер MPI-процесса, в который эту передачу нужно производить. Подробнее о том, что происходит в MPI-процессе, принимающем данные, было сказано выше. Пример протокола обменов можно видеть на рисунке 13;
- После этого ребро, по которому был инициирован обмен, добавляется в список sending_edges и удаляется из списка unsent_edges.

По окончанию работы с рёбрами происходит добавление вершин графа со следующего слоя в список вершин, требующих выполнения. Условием наступления данного события является обнуление списка вершин, требующих назначения на один из MPI-процессов, и факт наличия следующего уровня в графе в принципе.

Далее следует процесс информирования вершин, ожидающих сведений о том, на каких MPI-процессах находятся данные по входящим в вершину рёбрам. Иными словами, они находятся в блокированном состоянии до тех пор, пока не получат сообщения с меткой PX_EDGES_PLACE_TAG. В массиве waiting_recv управляющего процесса для процессов, на которых запущены вершины, находящиеся в соответствующей стадии, должна быть выставлена 1.

Для каждого такого процесса выясняется, какая вершина на нём запущена, какие рёбра входят в данную вершину. Затем определяется наличие наработанных данных для каждого входящего в вершину ребра. Для этой цели ребро ищется в списках sending_edges и unsent_edges. Если все рёбра найдены в этих списках, то MPI-процессу, на котором запущена вершина, посылается блокированное сообщение PX_EDGES_PLACE_TAG с массивом номеров MPI-процессов, на которых находятся необходимые данные рёбер. Размерность массива совпадает с числом входящих в вершину рёбер. Порядок совпадает с порядком указанным для вершины в текстовом файле с описанием графа.

Наконец, следует процесс назначения вершины на процессор. Это происходит в том случае, если список req не пустой, т.е. когда есть ещё не распределённые по MPI-процессам вершины и имеются свободные MPI-процессы, то есть для некоторого процесса в массиве loading написан 0.

Для этой цели координатором вызывается функция to_processors. Данная функция возвращает либо номер вершины, которую нужно назначить на указанный MPI-процесс, либо информацию о том, что процесс нужно оставить свободным (в этом случае в массив processors на соответствующее место заносится отрицательное значение, а в массиве loading оставляется 0). В случае возвращения номера загружаемой вершины вершина удаляется из списка req, затем в массив processors на соответствующее место заносится номер вершины, а в массив loading на соответствующее место заносится 1. Подробно работа функции to_processors будет рассмотрена ниже.

После выбора номера вершины для назначения на MPI-процесс соответствующему MPI-процессу отправляется сообщение с меткой PX_REGIME_TAG с содержимым PX_WORK. Данное сообщение переводит процесс в режим работы над вершинами графа, а затем сообщение с меткой PX_NODE_QUESTION_TAG, в котором указывается номер вершины, которую нужно запустить на данном MPI-процессе. На этом работа процесса координатора заканчивается.

4.5. Алгоритм выбора назначаемой вершины графа на MPI-процесс

Для работы функции, выдающей номер вершины, загружаемой на MPI-процесс, предусмотрено несколько режимов работы.

4.5.1. Статический режим

Решение о назначении вершины принимается, опираясь на сформированное заранее расписание. Расписание представляется как набор массивов с номерами вершин и набор счётчиков, показывающих, какая по счёту вершина в массиве в текущий момент должна быть исполнена. Расписание может быть создано при помощи генетического алгоритма, о чём написано ниже.

Когда необходимо получить номер запускаемой вершины для указанного MPI-процесса, просматривается один из массивов с вершинами графа, и в нём происходит смещение на число ячеек, указанное в счётчике, после этого выбирается номер указанной там вершины. Далее происходит поиск выбранной вершины в списке вершин, требующих выполнения; если вершина там не представлена, то в качестве возвращаемой вершины выбирается отрицательное значение – это означает, что MPI-процесс нужно оставить свободным.

4.5.2. Динамический режим

Идея работы в динамическом режиме заключается в поиске подходящей вершины графа для назначения её на свободный в текущий момент MPI-процесс. Вершина выбирается из списка заявленных на выполнение вершин. Для выбранной вершины время выполнения на данном MPI-процессе должно быть минимально среди других вершин.

Для определения такой вершины необходимо вычислить следующую формулу:

$$number = argmin\left(ExecTime(node_i, p) + TransferTime(InputEdges(node_i), p)\right),$$
$$\forall i : node_i \in Nodes, \quad number \in Nodes, \quad p \in Procs.$$

Здесь *number* – номер вершины, которая должна быть выполнена на MPI-процессе. *ExecTime* – функция для вычисления времени исполнения вершины на определённом MPI-процессе. Параметр p задаёт номер MPI-процесса. *TransferTime* – функция для вычисления времени передачи данных по всем входящим в вершину рёбрам, если вершина помещена на соответствующий MPI-процесс. *Nodes* – множество вершин, которые требуют своего выполнения каким-либо MPI-процессом; данное множество соответствует списку req управляющего MPI-процесса.

ExecTime вычисляется следующим образом:

$$time = \frac{weight(node) \cdot antiperfomance(p)}{StandardOperations}$$

Здесь *weight* – вес вершины, выраженный в количестве эталонных операций. Функция *antiperfomance* – время работы MPI-процесса с номером p над эталонной

задачей (в данном случае перемножение двух квадратных матриц 1000x1000); размерность задачи выражается в эталонных операциях. Размер эталонной задачи в эталонных операциях – *StandardOperations*.

Время передачи данных по рёбрам вычисляется так: $time = max(transfer(edge_i, p)), \forall i : edge_i \in InputEdges(node)$. Здесь считается, что передача данных, связанных с рёбрами графа, производится не блокированно, асинхронно и одновременно. Таким образом, передачи не мешают друг другу. В этих предположениях время передачи данных по рёбрам, входящим в вершину, можно считать как время самой медленной передачи данных по одному из рёбер.

К сожалению, предположение о том, что передачи не мешают друг другу, в общем случае неверно, но если с этим возникают проблемы, то предлагается протестировать многопроцессорную систему с помощью “шумящих” тестов. О них речь пойдёт далее.

В рассматриваемой формуле *transfer* – функция, определяющая время передачи данных по одному из рёбер, входящих в вершину графа, при условии, что вершина назначена на MPI-процесс с номером p. Рассмотрим подробно, как вычисляется *transfer*.

С ребром связано 2 характеристики: p_i – номер MPI-процесса, на котором находятся данные для ребра, и w_i – вес ребра, сколько байт необходимо по нему передать. Для определения времени передачи данных происходит обращение к нескольким матрицам с временами передачи данных. Каждая матрица соответствует длине передаваемого MPI-сообщения. Длины сообщений и размер шага приращения длины выбираются человеком в момент тестирования производительности коммуникационной среды многопроцессорной системы.

Таким образом, задержки при передаче сообщений в коммуникационной среде можно представить как набор дискретных функций от длины передаваемого сообщения: $ExactDelay_{i,j}(length_k),\ length_k \in Length$ со множеством указанных пользователем при тестировании длин сообщений – *Length.*

Возникает вопрос: каким образом получать значения для промежуточных точек, а также точек, которые находятся за пределами диапазона тестирования? Данная проблема довольно сложна, поскольку наиболее популярные алгоритмы квадратичной и кубической интерполяции могут давать весьма далёкие от истинной характеристики кривой результаты. Из соображений простоты реализации и предположении о ступенчатом характере поведения функции временной задержки от длины сообщения была принята следующая схема интерполяции:

$$delay_{i,j}(length) = \begin{Bmatrix} ExactDelay_{i,j}(l_{min}), & length < l_{min} \\ ExactDelay_{i,j}(l_m), & l_{min} < length < l_{max}, l_m < length < l_{m+1} \\ ExactDelay_{i,j}(l_{max}), & l_{max} < length \end{Bmatrix},$$

$\forall k : length_k \in Length$
$l_{min} = min(length_k),$
$l_{max} = max(length_k);$
$\exists m : l_m < length < l_{m+1}.$

Данная функция используется следующим образом: $transfer(edge, proc) = delay_{proc(edge), proc}(weight(edge))$. Здесь *proc(edge)* – номер MPI-процесса, на котором находятся данные ребра $p_i = proc(edge_i)$, weight(edge) – вес ребра, количество байт, которое по нему нужно передать, $w_i = weight(edge_i)$.

4.5.3. Комбинированный режим

Цель **комбинированного режима** - дать возможность человеку «помочь» планировщику в распределении вершин по MPI-процессам, указав информацию о желаемой привязке вершины к процессу. В комбинированном режиме используется тот же файл с расписанием, но он трактуется другим способом. Если для MPI-процесса в расписании указана вершина, то этот факт понимается так, что при прочих равных условиях данную вершину рекомендуется назначить на указанный процесс.

Выбор назначаемой вершины сначала происходит по аналогии с выбором в динамическом режиме. Далее, если для нескольких вершин получаются одинаковые времена их исполнения данным MPI-процессом, то происходит обращение к файлу в формате расписания. По файлу выбирается та вершина, которая рекомендована к назначению на соответствующий MPI-процесс.

Для предлагаемого алгоритма одна и та же вершина может быть рекомендована к назначению на несколько MPI-процессов; в случае статического расписания вершина обязана быть строго только на одном из MPI-процессов.

4.6. Генетический алгоритм построения расписания назначений вершин графа по MPI-процессам

Как было сказано выше, статический режим назначения вершин графа по MPI-процессам предполагает наличие файла расписания. В системе «PARUS» предусмотрена специальная программа, которая строит данный файл.

Под расписанием будем понимать вектор S, состоящий из элементов $s_{node_i} = (proc_j, rank_{node_i})$, где $proc_j \in Procs$, $node_i \in Nodes$, $rank_{node_i}$ – порядковый номер вершины при вызове её на MPI-процессе. Размерность вектора совпадает с числом вершин в графе.

Для расписания определена функция, вычисляющая время исполнения полученной параллельной программы. Данная функция вычисляется следующим образом: $GlobalTime(S) = max(T_{node_i} + \tau_{node_i}),\ \forall node_i \in Nodes$. T_{node_i} – момент времени, с которого необходимо начать исполнять вершину графа на MPI-процессе с номером $proc_j$. $\tau_{node_i} = NodeTime(node_i, proc_j, T_{node_i})$ – функция, определяющая как долго будет исполнятся вершина, если она будет запущена в момент времени T_{node_i} MPI-процессом $proc_j$.

$T_{node_i} = StartTime(node_i, rank_{node_i}, proc_j)$ – функция, имеющая рекурсивный характер. Поскольку назначение вершины на MPI-процесс необходимо производить строго по окончанию работы тех вершин, от которых данная вершина зависит, то для подсчёта времени старта вершины мы вынуждены вычислить все времена окончания этих вершин. Для вычисления *StartTime* определим множество «предков» $Parents(node_i)$ для вершины. В это множество входят все вершины, из которых существуют рёбра, входящие в вершину $node_i$. Предполагается, что родительские вершины уже закончили свою работу к моменту распределения $node_i$ на MPI-процесс, в противном случае множество *Parents* не определено и расписание не допустимо. *StartTime* для вершины графа вычисляется следующим образом:

$$StartTime = \begin{Bmatrix} Parents(node_i) = \varnothing, 0 \\ Parents(node_i) \neq \varnothing, max(T_{parent_j} + \tau_{parent_j}) \end{Bmatrix},$$

$$\forall\ parent_j \in Parents(node_i).$$

Порядок вызовов вершин на MPI-процессах влияет на то, возможно ли вообще

вычислить функцию *StartTime*. В случае невозможности вычисления данной функции расписание считается недопустимым.

Время исполнения вершины графа вычисляется следующим образом: $NodeTime = ExecTime(node_i, proc_j) + TransferTime(InputEdges(node_i), proc_j)$, *ExecTime* и *TransferTime* были рассмотрены ранее. Основная особенность – неявная зависимость функции *TransferTime* от момента назначения вершины на MPI-процесс T_{node_i}. Функция TransferTime вычислима только в том случае, если все необходимые данные по входящим в вершину рёбрам уже наработаны другими вершинами. Иными словами, все вершины графа, от которых зависит данная вершина, к моменту вычисления функции должны быть назначены на один из MPI-процессов и должны там доработать, то есть получить результаты и сформировать данные, передаваемые по рёбрам.

Цель алгоритма – каким-либо образом приблизится к минимуму функции *GlobalTime,* перебирая всевозможные допустимые расписания, то есть найти $S_{opt} = argmin(GlobalTime(S_i)),\ S_i \in \Xi$ (здесь Ξ – множество допустимых расписаний). При поиске расписания, близкого к оптимальному, мы варьируем порядок запуска вершин: параметр $rank_{node_i}$, а также MPI-процесс $proc_j$, на котором будет запущена вершина.

Задача построения расписания, близкого к оптимальному, решается при помощи генетического алгоритма. В качестве хромосомы, или особи, используется одно из допустимых расписаний. Популяция – некоторое подмножество множества допустимых расписаний. Функция качества одного из расписаний – это предполагаемое время исполнения программы по данному расписанию, для рассматриваемой реализации функции *GlobalTime*. Мутации производятся следующим образом: случайным образом в случайной позиции хромосомы изменяется либо порядковый номер на MPI-процессе, либо сам номер MPI-процесса, либо и то, и другое вместе. Количество мутаций регулируется параметрами программы. Скрещивание особей производится для одной точки, но в нескольких позициях. Каждая точка ищется при помощи датчика равномерно распределённых на отрезке случайных величин. Количество позиций, в которых производится скрещивание, определяется параметрами программы. Скрещивание производится таким образом, что хромосомы меняются элементами расписания, относящимися целиком к одной вершине графа. На основе 2-х расписаний S_i, S_j, принадлежащих к текущему поколению, формируются новые расписания S'_i, S'_j, принадлежащие следующему поколению. Случайным образом выбирается номер вершины графа k, после чего производится обмен информацией о привязке вершины к MPI-процессу:

$$\begin{Bmatrix} s'_{i,l} = s_{j,l},\ l = node_k \\ s'_{i,l} = s_{i,l},\ \forall l \neq node_k \end{Bmatrix},\ \begin{Bmatrix} s'_{j,l} = s_{i,l},\ l = node_k \\ s'_{j,l} = s_{j,l},\ \forall l \neq node_k \end{Bmatrix}.$$

Здесь $s_{i,l}$ – ген, соответствующий вершине с номером l и хромосоме с номером i. Номер вершины графа можно выбирать несколько раз. Число скрещиваний на фиксированную пару хромосом задаётся в параметрах программы.

В силу сравнительно высокой вычислительной сложности функции качества, полностью заменять следующее поколение особей и вычислять для всех них значения функции качества – довольно тяжёлая задача. Вместо этого используется параметризованный генетический алгоритм, где на каждом шаге порождается случайное число особей, полученных путем скрещивания или мутации родителей (родителя).

В момент старта алгоритма случайным образом порождается некоторое количество особей. Начальное количество можно задавать параметрами программы.

Далее над частью особей популяции производится операция мутации. Затем в популяции выбирается множество пар особей для проведения скрещивания. В результате такой последовательности действий алгоритма мутантные особи могут стать родителями для скрещиваемых особей. Старые, неизменённые хромосомы оставляются наряду с появившимися новыми. Из всей пополненной популяции выбираются лучшие особи. Лучшие особи – это те, у которых значение функции качества меньше, чем у остальных особей. Число отобранных особей не должно превышать значение, указанное в настройках программы. Для сохранения в популяции в том числе особей, не обладающих наилучшей функцией качества, процесс отбора ведётся с учётом случайного штрафа. Такая политика отбора позволяет легче выводить алгоритм из точек локального экстремума функции качества.

Алгоритм останавливается в случае незначительного изменения значения функции качества между 2-мя соседними по времени популяциями. Величина изменения функции качества задаётся в настройках программы. Другой причиной остановки может служить исчерпание отведённого количества итераций. Как результат работы алгоритма выбирается особь с наилучшей функцией качества в последнем поколении.

Все генетические операции производятся таким образом, чтобы не получалось недопустимых расписаний. Все особи с недопустимым расписанием удаляются, и операция повторяется снова.

Все параметры, указываемые программе, строящей расписание, указываются в .ini файле, формат которого будет рассмотрен в приложениях.

4.7. Система тестирования многопроцессорной системы

Цель системы тестирования заключается в предоставлении наиболее полной информации о многопроцессорной системе для этапа планирования назначений вершин на MPI-процессы. Здесь выясняется производительность процессоров многопроцессорной системы и пропускная способность коммуникаций. Физически коммуникационная среда многопроцессорной системы может быть гетерогенна. Система тестирования устроена таким образом, чтобы информация о несбалансированности пропускной способности коммуникаций попала на этап планирования в форме, унифицированной для произвольной многопроцессорной системы. Это достигается за счёт определения статистическим образом задержек при передаче сообщений через коммуникационную среду многопроцессорной системы от каждого процессора к каждому в различных режимах для определённого диапазона длин сообщений.

Система тестов в PARUS содержит два вида тестов: производительности процессоров и пропускной способности коммуникационной среды. Производительность процессоров определяется по времени перемножения 2-х матриц фиксированного размера. Пропускная способность коммуникационной среды определяется на основе системы тестов, которые измеряют время передачи MPI-сообщений между MPI-процессами. Время измеряется с точностью, предоставляемой функцией MPI_Wtime. Эта функция выдаёт текущее время в секундах, но в формате числа с плавающей точкой и с учётом долей секунды.

Библиотека MPI позволяет передавать сообщения между MPI-процессами в разных режимах. Как правило MPI-процесс соответствует одному фиксированному процессору многопроцессорной системы в момент исполнения MPI-программы. Предусмотрен блокированный режим, когда процесс блокируется на время передачи сообщения, и неблокированный, когда процесс может производить какие-либо действия на фоне передачи данных другим MPI-процессам. Для обеспечения неблокированных обменов в MPI предусмотрено несколько способов синхронизации

MPI-процессов. На основе предлагаемых в MPI средств по передаче сообщений и синхронизации MPI-процессов, а также предполагая, что MPI-процесс соответствует фиксированному процессору многопроцессорной системы, можно строить систему тестирования коммуникационной среды. Она будет таким образом нагружать коммуникационную среду, чтобы выявлять особенности оборудования и помогать делать предположения о способе соединения процессоров в многопроцессорной системе (администраторы не всегда предоставляют такую информацию).

Системе тестирования коммуникационной среды указываются следующие параметры: диапазон длин сообщений, шаг приращения длины сообщения, число повторов на фиксированную длину, способ тестирования. Для группы шумящих тестов, описанных ниже, дополнительно указывается размер шумового сообщения, число шумящих MPI-процессов. Тесты, зависимым от способа тестирования образом, измеряют задержки при передаче сообщений между всеми возможными парами MPI-процессами, участвующими в тестировании. Результаты измерений формируют матрицу размерности NxN, где N – число задействованных MPI-процессов. Элемент (i, j) матрицы соответствует времени передачи сообщения от i-го процесса к j-ому. Для каждой длины сообщения тест «прогоняется» некоторое количество раз (число прогонов задаётся как параметр программы), а затем берется среднее арифметическое по этим значениям, – это делается для обеспечения большей точности измерений. По результатам тестов формируется файл, в который сохраняются полученные матрицы для всех протестированных длин сообщений.

По способу передачи сообщений между MPI-процессами реализовано несколько видов тестов:

- ***all_to_all*** – тест основан на одновременной неблокированной передаче данных от каждого к каждому и предназначен для выяснения «стрессоустойчивости» коммуникационной среды. По идее должен вызывать множество коллизий в коммуникационной среде. Измеряется время между инициализацией неблокированного приёма *MPI_Irecv()* и соответствующем ему *MPI_Waitany()*. Данное значение вносится в матрицу, соответствующую тестируемой длине сообщения, в позицию (i, j), где j – принимающий MPI-процесс, i – процесс, из которого было отправлено сообщение.
- ***one_to_one*** – тест, производящий блокированную передачу от i-ого процесса к j-ому. Во время передачи данных остальные процессы (кроме i-го и j-ого) «молчат». Измеряется время исполнения функции *MPI_Recv()*, после чего это время считается временем передачи сообщения от i-ого процесса к j-ому. Такая передача производится для всех процессов, то есть i и j пробегают значения от 1 до n , где n - число задействованных процессов. Исключение составляет ситуация, когда i=j, в этом случае в матрицу записывается 0. При помощи данного теста можно определить пропускную способность каналов связи между процессорами (узлами) многопроцессорной системы при условии, что в момент исполнения теста в коммуникационной среде не ведётся никаких посторонних передач данных, которые могут существенно снижать пропускную способность.
- ***async_one_to_one*** – тест, осуществляющий неблокированный обмен MPI-сообщениями навстречу друг другу между i-ым и j-ым процессом. Два процесса одновременно вызывают *MPI_Isend()*, затем с обеих сторон одновременно инициируется *MPI_Irecv()* и *MPI_Waitany()*. После выполнения *MPI_Waitany()* i-ым процессом измеряется время. Разница между измеренным временем и моментом инициализации

соответствующего неблокированного приёма сообщения вносится как время передачи от j-ого MPI-процесса к i-ому. При помощи рассмотренного теста можно определить, является ли канал между двумя процессорами полнодуплексным. Если канал полудуплексный, то полученные результаты будет в 2 (и более) раза меньше, чем результаты, полученные при помощи теста *one_to_one.*

- ***send_recv_and_recv_send*** – тест, предназначенный для выяснения усредненной производительности обменов в коммуникационной среде многопроцессорной системы. Блокированное сообщение передаётся от i-ого процесса к j-ому, а затем сразу ретранслируется обратно от j-ого к i-ому. Время измеряется между первой отправкой сообщения и получением ответа обратно. После этого оно делится на 2 и в таком виде вносится в матрицу, как время передачи от i-ого MPI-процесса к j-ому.
- ***test_noise*** – работает аналогично тесту *async_one_to_one*, но добавлен параметр «шума». Все MPI-процессы разбиваются на две группы. В первую группу выделяется пара MPI-процессов, между которыми затем будет измеряться время передачи «целевого» сообщения. Во вторую группу входят все остальные процессы. Среди процессов второй группы случайным образом выбираются «фоновые» процессы. Количество «фоновых» процессов задается в командной строке. После этого инициализируются неблокирующие передачи сообщений. Между парой MPI-процессов, принадлежащих к первой группе, передаётся «целевое» сообщение определённой длины; для «фоновых» процессов инициализируется передача «шумового» сообщения. Размер сообщения и есть уровень шума. Уровень шума задаётся в командной строке, как один из параметров программы. MPI-процессы, не выбранные в качестве «шумовых» и не относящиеся к тем, время передачи между которыми измеряем, простаивают.
- ***test_noise_blocking*** – тест, измеряющий время блокирующих передач на фоне «шума». Выбор «фоновых» процессоров и уровня «шума» аналогичен тесту *test_noise*.

Некоторые результаты проведения экспериментов на различных многопроцессорных системах будут приведены ниже. «Шумящие» тесты можно использовать для прогнозирования задержек при передаче сообщений в случае, если многопроцессорная система не полностью отдана в распоряжение одной пользовательской задачи. На фоне исполнения задачи в коммуникационной среде наблюдается дополнительная активность с прогнозируемым уровнем (например, активность от других задач).

4.8. Анализатор зависимостей по данным в С-программе

4.8.1. Общее описание

Целью является разработка программного средства, преобразующего текст последовательной программы в граф зависимостей по данным. Виды зависимостей были формально описаны и обсуждены во введении к данной работе. В качестве языка выбран язык **C** (стандарт **ANSI ISO**). С помощью полученной реализации предполагается исследовать возможности и перспективы, связанные с применением полностью автоматического подхода к созданию параллельных программ. Построенный граф может использоваться как независимый от архитектуры машины способ представления программы.

Анализ текстов проводится в предположении статического распределения памяти. В программе не должно быть динамического выделения памяти, например системного вызова malloc. Конечно, при статическом анализе невозможно полностью учесть всех особенностей языка программирования С из-за разнообразия функциональных возможностей и широко применяемых программистами приемов программирования в своей повседневной практике. Например, очень сложно проводить статический анализ, если включить в представление тип данных *указатель* и операции над объектами этого типа, предусмотренные в языке **C**. Поэтому в текущей версии разработанного анализатора эта и подобные ей особенности языка не анализируются, и при их обнаружении программа прекращает свою работу выдачей диагностического сообщения. Проверка на наличие в С-программе не поддерживаемых в текущий момент особенностей проводится на этапе определения информационных зависимостей, который происходит после распознавания текста.

Основной стратегией при построении графа по С-программе , является стратегия максимального разделения операций с целью получения наиболее широкого представления графа, то есть максимизации количества независимых между собой операций. Арифметические выражения переводятся в древовидную структуру, над которой реализован базовый набор операций и специальные преобразования. Например, можно вычислить значение выражения, если значения переменных в нем присутствующих определены, преобразовать в линейную форму и упростить индексное выражение и др. Далее эта же структура используется в представлении информации о зависимостях. Стоит заметить, что при работе не происходит анализа выражений на распараллеливаемость, поскольку изменение последовательности вычислений может привести к некорректным или «неправильным» результатам (обычные арифметические операции для чисел с плавающей запятой не коммутативны). Однако перераспределение вычислений в выражении возможно. Это может быть достигнуто путем предварительного вычисления результатов, используемых функций, с последующей подстановкой результата в место вызова. Заметим, что такая функция не должна иметь побочных эффектов.

Для каждой функции определяется набор глобальных переменных, с которыми она оперирует. Определяется вес функции, характеризующий её вычислительную сложность. В теории возможно сделать подстановку тела функции в место вызова, с подстановкой актуальных параметров. В **C++** такие функции помечаются ключевым словом **inline**. К сожалению, в текущей реализации такая возможность не реализована в настоящий момент. На такие функции накладывается некоторые существенные ограничения, в нашем случае самым серьезным является требование единственности оператора **return** у функции. Совершенно ясно, что оператор **return** задает т.н. зависимость по управлению и напрямую не выразим в терминах информационных зависимостей. Единственный (безусловный) **return** может быть заменен на присвоение некоторой переменной возвращаемого значения. Если тело функции содержит условные операторы с **return** в качестве выполняемого действия, то такой текст нельзя подставлять в место вызова данной функции. Подобный стиль написания функций довольно широко распространен, благодаря его удобству и наглядности.

4.8.2. Анализ зависимостей

Обратимся к тому, как происходит построение дерева зависимостей для блока операторов С-программы. Для каждого оператора в блоке определяется следующая информация: множество входных параметров, множество выходных параметров, вес оператора. В дальнейшем данная информация будет называться контекстом

оператора. Вес - неотрицательное число, характеризующее вычислительную сложность оператора, выраженный, например, в количестве тактов абстрактной машины. Для входных и выходных параметров оператора указываются имя встречающейся в операторе переменной, тип переменной, признак вхождения переменной в условный оператор и, если это переменная – массив, список диапазонов - интервалов, определяющих используемую часть массива по каждому измерению, с описанием размерностей.

Поскольку в идеале для каждой функции программы происходит разворачивание её структуры, с подстановкой в место вызова, то для каждой переменной, указанной в блоке, происходит преобразование её имени с целью получения уникального имени переменной для программы целиком. Всякое имя переменной в программе получается следующим образом: удваивается количество подчеркиваний, и к нему добавляется суффикс <_число>, где число - номер блока операторов языка С, в котором объявлено имя переменной. Нумерация блоков сквозная и начинается с нуля. Для глобальных имен суффикс не используется, это позволяет при чтении легко отличить глобальные имена от остальных имен программы.

Анализ блочного оператора характеризуется наличием локальных переменных, использующихся только внутри этого оператора и невидимых другим операторам программы, расположенным на верхних уровнях. Кроме того, возможно перекрытие имен переменных, объявленных ранее в объемлющих блоках или глобально, поэтому следует отличать различные упоминания одного и того же имени. Данные проблемы решаются описанным ранее способом.

Блочный оператор состоит из списка деклараций локальных переменных и списка операторов. Для каждого блочного оператора порождается новый контекст, в котором регистрируются локальные переменные. Контекст объемлющего блока становится его родителем. Далее происходит анализ последовательности операторов блочного оператора на основе нового контекста. Входные и выходные параметры блока определяются как объединение входных и выходных параметров всех внутренних операторов. После проведения операции объединения контекстов внутренних операторов для описания внешних зависимостей из построенного объединённого контекста удаляются упоминания о локальных переменных блочного оператора. Результат такого просеивания будет входом/выходом для блочного оператора в целом. В дальнейшем при анализе зависимостей в родительском блоке по отношению к переменным внутреннего блока будет использоваться только построенный таким образом контекст.

Зависимости для внешних переменных, использующих внутренние переменные блока, строятся следующим образом. Скалярные переменные и массивы сначала исключаются из списков входных и выходных зависимостей, затем анализируются индексы, и если они содержат вхождения локальных переменных (неважно, массивов или скаляров), то индексы заменяются на диапазон, описывающий длину соответствующего этому индексу измерения. Если элементы массива из диапазона входят в условные операторы, то такие диапазоны помечаются. Так поступают, поскольку невозможно гарантировать изменение всех элементов массива, соответствующих данному диапазону, и нельзя указать точно, какой элемент меняется.

Далее обратимся к методу анализа зависимостей в циклах. В виду сложности анализа производится только анализ цикла ***for*** при наложении на него следующих условий:

1. границы цикла должны быть заданы константными выражениями.
2. индексные переменные должны меняться только в заголовке цикла строго с шагом единица и не должны меняться внутри тела цикла.

3. Внутри тела цикла не должно быть операторов ***break/continue/return***.

Первое условие связано вообще со сложностью анализа преобразований переменных. Второе и третье условия гарантируют, что индекс пробежит все значения из диапазона определённого границами цикла. Игнорирование этих требований может привести к некорректному результату. Например, какой-то элемент массива из диапазона может остаться не инициализированным, но будет помечен как получивший новое значение. Последующие операторы будут думать, что значение этого элемента они должны получить от этого цикла, но в действительности они получат «мусор», а не то значение, какое бы получили при последовательном исполнении. Логика исходной программы будет нарушена. При этом требование единичного шага итерации цикла не критично и легко может быть снято, а в текущей реализации присутствует из соображений простоты.

Анализ зависимостей по данным в цикле разбивается на несколько этапов. На первом этапе, для каждой итерации цикла производится анализ зависимостей переменных, встречающихся в теле цикла, от других итераций данного цикла. На следующем этапе производится преобразование зависимостей для тела цикла в зависимости от внешних по отношению к циклу переменных. Далее определяется выход цикла, то есть та часть данных, которая была изменена во время исполнения цикла. Далее во всех операторах, где будут задействованы переменные, которые использовались циклом, будет присутствовать в зависимостях только рассматриваемая изменяемая часть цикла. Например для цикла

```
for(i=1; i<=10; i++)
   a[i][i+1] = 1;
```

выход будет a[1][2], a[2][3] … a[10][11], а не a[1][2], a[2][2], a[1][3] …

Анализ условных операторов (***if***,***switch***) производится следующим образом. Для построения зависимостей строится объединение In и Out от всех ветвей условного оператора. К сожалению, в статике редко удаётся предсказать, какая именно ветка условного оператора выполнится. Поэтому затем, при преобразовании зависимостей в граф, весь условный оператор представляется как одна вершина.

4.8.3. Построение графа

Далее по всем найденным зависимостям строится граф. Просмотр начинается с функции main С-программы, в которой производится проход по всем операторам функции. Если среди операторов встречается вызов функции, или цикл, или условный оператор, то по ним принимается решение о том, нужно ли рассматривать данный оператор как единое целое, или можно его разделить на части, и каждую часть рассматривать отдельно. Операция такого разбиения оператора на составные части в дальнейшем будет называться операцией раскрытия оператора. В результате операции раскрытия вместо одной вершины, соответствующей оператору, получается группа вершин, состоящая из операторов, входящих в раскрываемый оператор.

Последовательность операторов, которые следуют естественным порядком, то есть когда операторы выполняются строго последовательно один за другим, без операторов, в которых есть операторы переходов, будем называть линейным участком.

Таким образом, построение графа зависимости производится рекурсивно по всем составным операторам, которые преобразуются либо в одну вершину графа, если оператор не допускает операцию раскрытия, либо в множество вершин, если операция раскрытия допустима. Процесс добавления очередной вершины в граф

состоит из определения типа зависимости от других вершин для данной вершины, и определения того, какие данные необходимо передать между вершинами, в случае наличия между ними зависимости. В случае обнаружения прямой зависимости по данным между 2-мя вершинами проводится ребро, для которого указываются переменные, участвующие в зависимости, и объём пересылаемых данных. Если Out(i) и In(j) пересекаются, то создаётся ребро, направленное от i-ой вершины к j-ой. В случае обнаружения обратной зависимости между 2-мя операторами вершины аналогично соединяются ребром, но поскольку в данном случае важен только порядок следования операторов, а данные не имеют значения, то строится ребро с 0 весом. В этом случае ребро направлено таким образом, чтобы не нарушать порядок следования операторов. Тем самым декларируется важность самого факта инициализации передачи данных, а что именно будет передано по ребру, не имеет значения.

С целью оптимизации все дуги между парой вершин, независимо от их типа, объединяются в одну дугу, с конкатенацией данных. Такой подход позволяет передавать требуемые данные за одну операцию передачи данных, позволяя тем самым снизить расход времени, связанный с латентностью коммуникационной среды.

На начальном этапе производится раскрытие всех составных операторов до состояния, когда получаются либо атомарные, либо нераскрываемые операторы. Раскрытие различных составных операторов производится по-разному. Блочный оператор раскрывается всегда. Раскрытие цикла происходит только в том случае, если удается определить границы изменения итератора и в теле отсутствует передача управления (**return/break/continue**). При выполнении этих условий цикл преобразуется в последовательность итераций, где для каждой итерации явно прописывается значение индекса цикла. Пример раскрытия итераций цикла **for** приведён на рисунке 15. К сожалению, в текущей реализации нет возможности раскрытия циклов **while** и **do while**, главным образом из-за сложности определения границ цикла и отсутствия итератора цикла. По схожим причинам не производится и разделение на ветки операторов **switch** и **if else**.

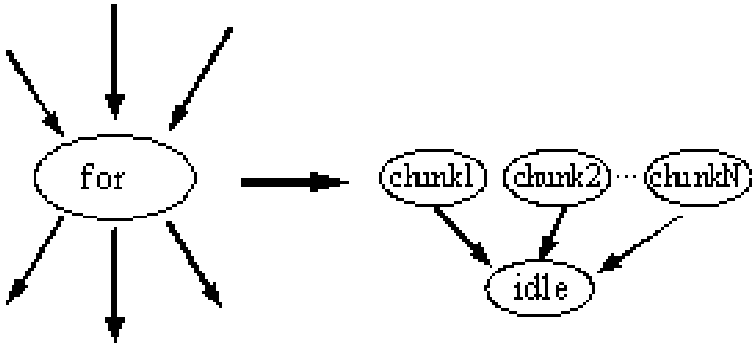

Рисунок 15. Разбитие цикла for на итерации.

Применительно к вложенным циклам можно сказать, что после раскрытия всех итераций получается огромное количество вершин; так, для простого алгоритма перемножения матриц (3 вложенных цикла) и матрицы 10x10 получается порядка 1100 вершин.

```
for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
                for (k = 0; k < 10; k++)
                        Result[i][j] += Left [i][k] + Right [k][j];
```

В приведённом примере видно, что раскрытие только верхнего цикла дает максимальную степень параллельности 10, тогда как раскрытие еще 2-го цикла увеличивает эту цифру в 10 раз. При таком подходе, т. е. раскрытии всех итераций, необходима дополнительная процедура «свертки» графа в соответствии с реальным числом процессоров и минимизацией пересылок, что, к сожалению, в текущий момент не реализовано.

Рассмотрим способ построения фрагмента графа для линейного участка. Пусть $S_1,S_2,...,S_i,...,S_N$ – последовательность операторов линейного участка. Между 2-мя операторами S_i и S_j, i<j имеется зависимость во всех случаях, если результаты исполнения i-го оператора нужны j-ому оператору. В данной ситуации, казалось бы, два данных оператора необходимо соединить ребром, однако если между этими операторами присутствует оператор S_k, который модифицирует переменную, требуемую на вход j-ому оператору, то в этом случае будет проведено не одно ребро, а 2 ребра между вершинами, соответствующими операторам S_i, S_k и S_k, S_j.

Процесс построения начинается от последнего оператора и продолжается в обратную сторону. Для оператора, у которого переменная $var \in \mathrm{In}(i)$ входит во множество входов оператора, ищется ближайший к нему оператор, у которого данная переменная входит во множество выходов оператора $var \in \mathrm{Out}(i)$. Весом ребра становится количество байт, занимаемое переменной, участвующей в зависимости. Так продолжается до тех пор, пока не будет достигнут первый оператор. В случае, если для пары операторов прямая зависимость не выявлена, ищется обратная зависимость. Безотносительно от наличия прямой или обратной зависимости для последовательности операторов ищутся зависимости по выходам.

4.8.4. Определение весов операторов

Каждой вершине графа приписан её вес. Как уже говорилось ранее, это число, характеризующее вычислительную сложность вершины. В данном случае это оператор, как атомарный, так и составной (цикл, условный оператор, вызов функции). Данная информация необходима в дальнейшем для алгоритмов планирования вычислений. В данной реализации вес оператора считается как число эталонных операций, и в качестве эталонной операции выбрана операция сложения 2-х чисел с плавающей точкой. Таким образом, для атомарных операций можно произвести очень приблизительный подсчёт их времени исполнения относительно вычисления операции сложения. Для составных операторов всё обстоит несколько сложнее, поскольку они состоят из множества операторов. Среди множества операторов могут встречаться вызовы функций, о времени вычисления которых на этапе статического анализа программы может не оказаться сведений.

Для условных операторов вес вычисляется как сумма веса условия оператора и максимального значения веса одной из альтернатив. Вес линейного участка кода вычисляется как сумма весов, составляющих его операторов.

Обратимся к вычислению весов циклов. Для определения веса цикла for необходимо знать численные значения границ изменений параметра цикла и вес его

тела. Для нескольких вложенных операторов **for**:

```
for (I1=L1 ; I1<=U1 ; I1++)
{
        ...
        for(I2=L(I1) ; I2<=U(I1) ; I2++)
        {
                ...
                for(IK=L(I1,I2,...,IK-1) ; IK <=U(I1,I2,...,IK-1) ; IK ++)
                {
                        Body(I1,I2,...,IK)
                }
                ...
        }
        ...
}
```

вес будет вычисляться по формуле:

$$Weight = \sum_{I_1=L_1}^{U_1} (... + \sum_{I_2=L_2(I_1)}^{U_2(I_1)} (... + \sum_{I_k=L_k(I_1,...,I_{k-1})}^{U_k(I_1,...,I_{k-1})} BodyWeight(I_1,...,I_k) + ...) + ...)$$

Здесь тело и диапазоны изменения границ вложенного цикла зависят от параметров объемлющих циклов. Как видно из приведённой формулы, большая степень вложенности влечёт большие вычислительные затраты при вычислении веса.

К сожалению, про число итераций цикла **while,** как правило, ничего сказать нельзя, поэтому вес цикла считается как вес его тела. Тоже самое относится и к циклу **do while**.

4.9. Редактор графа и расписаний

Редактор позволяет создавать новый граф, редактировать существующий, а также изменять расписание выполнения графа программы на многопроцессорной системе. Редактор реализован на языке ***Java2*** из соображений переносимости. В проекте была использована компонента ***JGraph***, которая была взята с одноименного открытого проекта.

Граф представляется на экране как набор прямоугольников, соединённых стрелками; стрелки обозначают зависимость по данным между вершинами. Чем темнее цвет прямоугольника, тем больший вес сопоставлен соответствующей ему вершине графа, и тем более он трудоёмок в вычислительном смысле.

Рассмотрим подробнее работу с узлами и ребрами. На вершине графа можно открыть форму, представленную на рисунке 16, в которой пользователь может изменить атрибуты вершины графа.

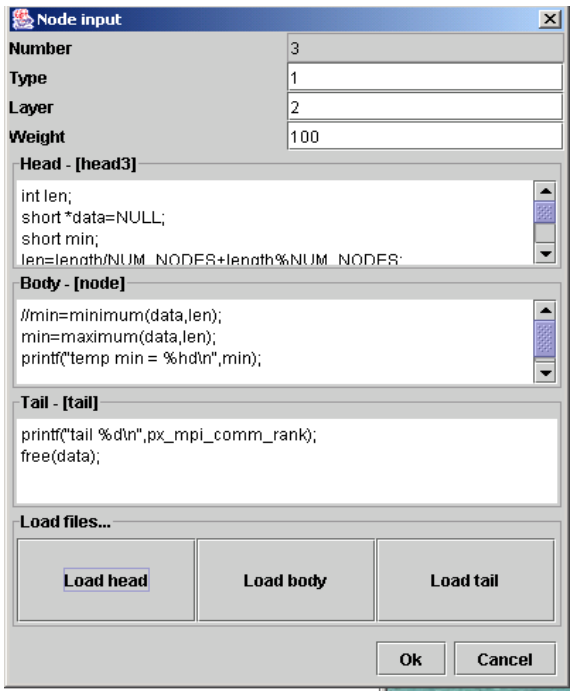

Рисунок 16. Представление вершины графа.

К атрибутам вершины относятся: номер, уровень в графе, вес, тип, а также программный код, выполняемый в вершине. Изменение текста создаваемой граф-программы можно произвести вручную, отредактировав соответствующий текстовый файл. Аналогичные действия можно провести с ребрами. Здесь атрибутами ребер будут его номер, тип, вес, число массивов посылаемых данных и сами массивы, нарезанные на чанки. (рис. 17.)

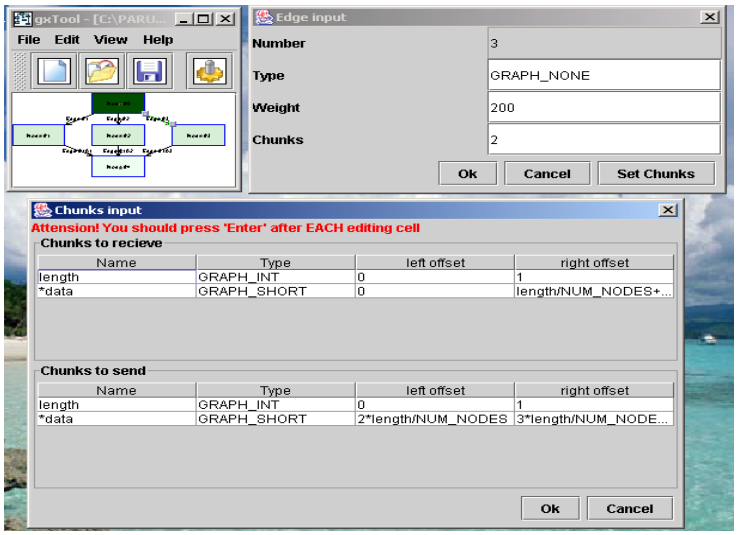

Рисунок 17. Представление рёбер в графе.

У пользователя также есть возможность отредактировать файл с расписанием. Таких файлов может быть несколько. Пользователь должен выбрать нужный ему файл. Визуализация расписания производится по аналогии с визуализацией графа. Пример расписания представлен на рисунке 18.

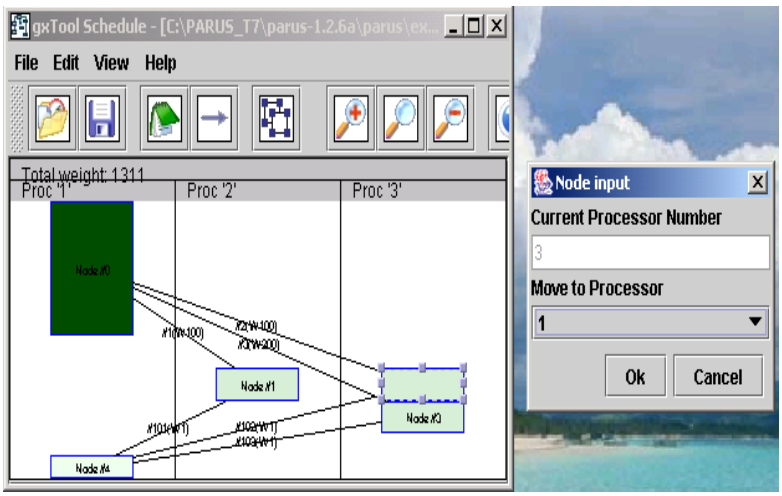

Рисунок 18. Представление расписания.

Вертикальные разделения окна обозначают процессоры, на которые распределены вершины графа. Временем выполнения каждой вершины на каком-либо процессоре считаем вес этой вершины. Редактором считается максимальное из времён выполнения вершин на процессорах - это есть предположительное относительное время выполнения программы. Имеются возможности просмотреть текст файла с описанием расписания, переносить вершины графа на другой процессор, уменьшать/увеличивать изображение, включить режим просмотра расписания без ребер.

4.10. Визуализатор данных о производительности сети и процессоров

Как показывает практика, данные, которые были получены как результат тестирования многопроцессорной системы, оказываются интересны не только для построителя расписаний исполнения вершин графа на многопроцессорной системе и динамическому планировщику, но и человеку для понимания деталей поведения многопроцессорной системы. В случае с процессорами всё обстоит сравнительно просто. По получаемому вектору с числами можно делать грубые выводы о том, насколько один процессор или одна машина производительнее другой. В случае с коммуникационной средой дело обстоит значительно сложнее. Полученные данные имеют четырёх мерный характер.

Для облегчения навигации в соответствующем многомерном пространстве были созданы специальные приложения, визуализирующие результаты тестирования многоцессорной системы.

- paMonitor – программа визуализации задержек при передаче данных по коммуникационной среде многопроцессорной системы.
- ppMonitor – программа визуализации производительности процессоров многопроцессорной системы или отдельных компьютеров многопроцессорной системы.

Программы полностью реализованы на языке ***Java2*** Standart Edition для поддержания свойства независимости от операционной системы. В проекте был использован пакет ***Java3D***, который был взят с сайта разработчика ***java.sun.com***.

В программе ppMonitor, наряду с демонстрацией разницы в производительности отдельных процессоров или компьютеров, производится слежение за текущей их загруженностью, для этого через определённые промежутки времени производится удалённый вызов функции операционной системы, которая показывает, насколько к текущему моменту времени использовался процессор.

К сожалению, у различных операционных систем соответствующие функции сильно отличаются, и ppMonitor может отобразить только данные, полученные с этапа тестирования многопроцессорной системы. ppMonitor может отобразить четыре параметра: антипроизводительность (*antiperfomance)*, загруженность (*loading)*, используемая производительность, доступная, или незанятая, производительность. Загруженность измеряется в процентах, а антипроизводительность – в секундах. Антипроизводительность – это число секунд, которое было затрачено процессором на работу над вычислительным тестом – задачей фиксированной размерности. Используемая производительность вычисляется по формуле:

$$\frac{N}{antiperfomance} * \frac{loading}{100}$$

и указывает, какое количество эталонных операций в секунду исполняет процессор в текущий момент времени. Доступная производительность вычисляется по формуле:

$$\frac{N}{antiperfomance} * (1 - \frac{loading}{100})$$

и указывает, сколько ещё эталонных операций в секунду способен производить процессор. В обоих формулах N – число эталонных операций, производимых тестом производительности процессора в момент тестирования. Пример работы ppMonitor приведён на рисунке 19.

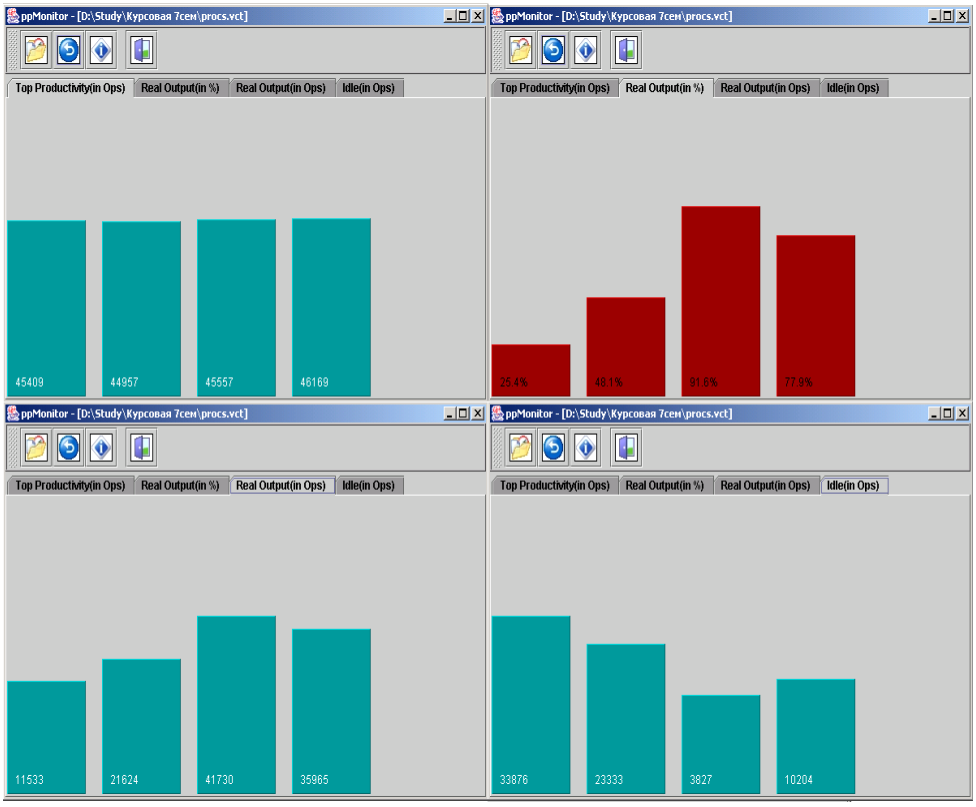

Рисунок 19. Пример работы визуализатора производительности и загруженности процессоров.

В программе ***paMonitor*** имеется несколько различных средств и возможностей для просмотра информации о загруженности сетевого интерфейса вычислительной системы.

По умолчанию загруженность сети изображается программой в виде трехмерной диаграммы, где в горизонтальной плоскости лежат соответственно посылающие и получающие сообщение MPI-процессы. По желанию пользователя

может быть установлен режим просмотра задержек при передаче данных в виде поверхности, а так же как один или несколько двумерных графиков. Представление данных как поверхности или как набор столбцов приведено на рис. 20.

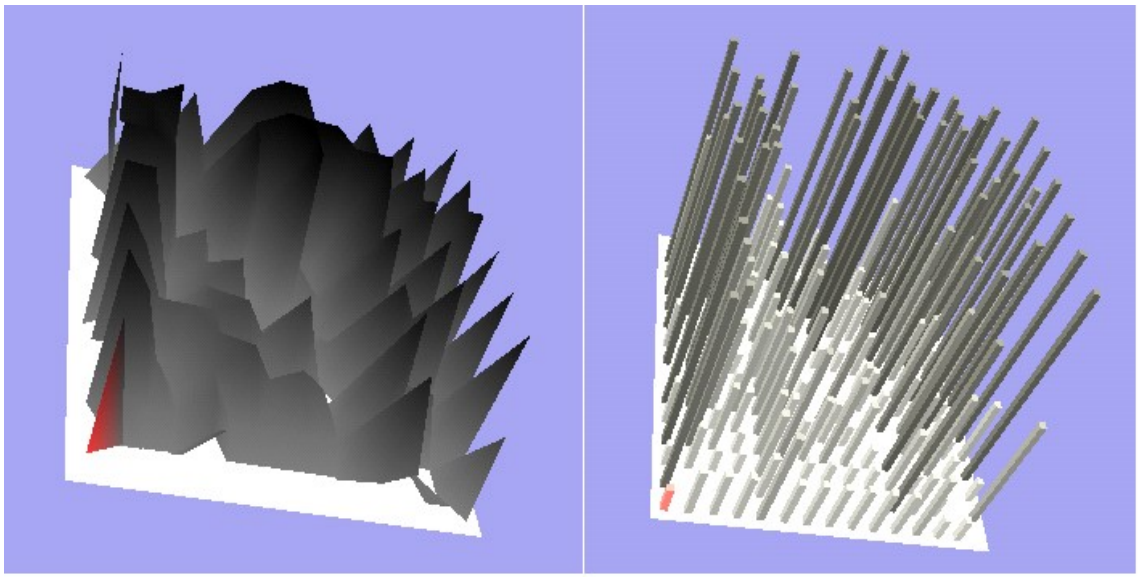

Рисунок 20. Представление информации о задержках при передаче данных в коммуникационной среде многопроцессорной системы.

В случае просмотра результатов как поверхности или как диаграммы с набором столбцов фиксируется одна из длин сообщений, и для всех пар MPI-процессов, передававших сообщение фиксированной длины, по оси Z откладывается время, затраченное на передачу сообщения. Затем, если это необходимо, через полученные таким образом точки проводится поверхность. В программе предусмотрен специальный слайдер, который позволяет человеку перейти к выбранной длине сообщения.

Просмотр результатов в виде двухмерного графика (хронология загруженности), в отличие от предыдущего случая, реализован для фиксированной пары MPI-процессов для всех длин передававшихся между ними сообщений. MPI-процессы фиксируются при помощи специального навигатора, оформленного как квадратная таблица с активными ячейками. Выбор ячейки означает выбор пары MPI-процессов, для которых строится график задержек. По горизонтали откладывается длина сообщения, по вертикали - время передачи сообщения. В промежуточных точках производится аппроксимация значений.

Интенсивность цвета столбцов диаграммы, цвета соответствующих областей поверхности и цвета соответствующей ячейки на навигаторе зависят от времени передачи сообщений: чем дольше передавалось сообщение, тем насыщеннее цвет визуализируемого объекта. Для визуализации этих объектов используется комбинация цветов белый-серый-черный. При фиксации пары MPI-процессов на навигаторе столбец, отвечающий этим процессам, выделяется красным цветом. В момент просмотра «хронологии» загруженности для выделенной пары MPI-процессов точка, соответствующая матрице (длине сообщения), в которой выделяли столбец, будет выделена красным цветом. См. рисунок 21.

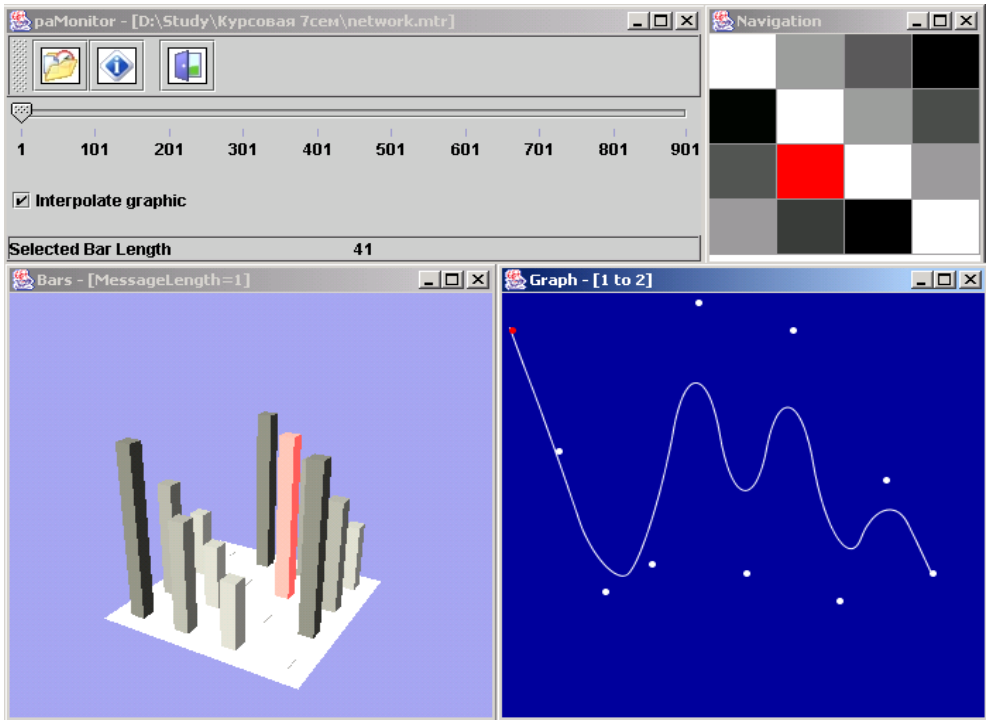

Рисунок 21. Внешний вид программы paMonitor.

Рассмотренный способ визуализации данных о коммуникационной среде позволяет человеку выяснить для себя особенности поведения многопроцессорной системы при передаче данных. Например, на рисунке 20 приведена диаграмма, из которой видно, что на данной многопроцессорной системе, при заданной длине сообщения выгоднее назначать на исполнение код, интенсивно обменивающийся данными между MPI-процессами, на процессы с меньшими номерами. В этом случае данные будут передаваться быстрее. Этот факт можно учесть при ручном составлении расписания для системы «PARUS». Вершины можно назначать таким образом, чтобы они попадали на MPI-процессы, для которых столбцы диаграммы имеют минимальную высоту.

5. Примеры использования системы «PARUS»

Работоспособность системы «PARUS» была протестирована в процессе решения нескольких модельных задач. Система «PARUS» используется в текущий момент для решения некоторых научно-прикладных задач. Далее рассмотрены сами задачи и способы их распараллеливания.

5.1. Распределённая операция над массивом (модельная задача)

Довольно часто над каждым элементом большого массива приходится производить одну и ту же операцию, а полученный результат помещать в одну скалярную переменную. Например, можно вычислить максимальное значение в массиве. Однако если массив имеет большой размер, то данная операция может занимать большое количество времени. К счастью, такие операции могут быть легко распараллелены. Массив можно распределить по процессорам многопроцессорной системы, в результате чего операция станет распределённой. Базовый набор распределённых операций входит в стандарт MPI. В статье [28] описана реализация распределённых операций в одной из наиболее популярных реализаций стандарта

MPI – mpich. Также распределённые операции входят в стандарт OpenMP и его реализацию, например в компиляторах Intel [29].

Была создана распределённая реализация поиска максимума и минимума с одновременным суммированием ячеек массива действительных чисел. В данной реализации имеет место рекурсивная процедура разделения массива на фрагменты. Массив делится на m фрагментов размера в n ячеек. Каждый фрагмент обрабатывается независимо, в результате получается m новых значений, над которыми в свою очередь необходимо провести операцию. Полученный массив опять разбивается на к-частей размера n, и процедура повторяется. Таким образом, очередной результат зависит от предыдущих результатов вычисления. Данный процесс распределения и вычисления может быть представлен как граф. Операции над каждым фрагментом становятся вершиной графа, а рёбра направлены от фрагмента текущей итерации к собираемому по результатам текущей итерации фрагменту следующей итерации. Таким образом будет получено дерево, где в последней вершине будет вычислено искомое значение максимума, минимума и суммы ячеек массива. На рисунке 22 приведён пример распределения операции суммирования над массивом. Массив из 10 элементов поделён на группы по 2 элемента, для каждой группы суммирование фрагмента массива производится независимо.

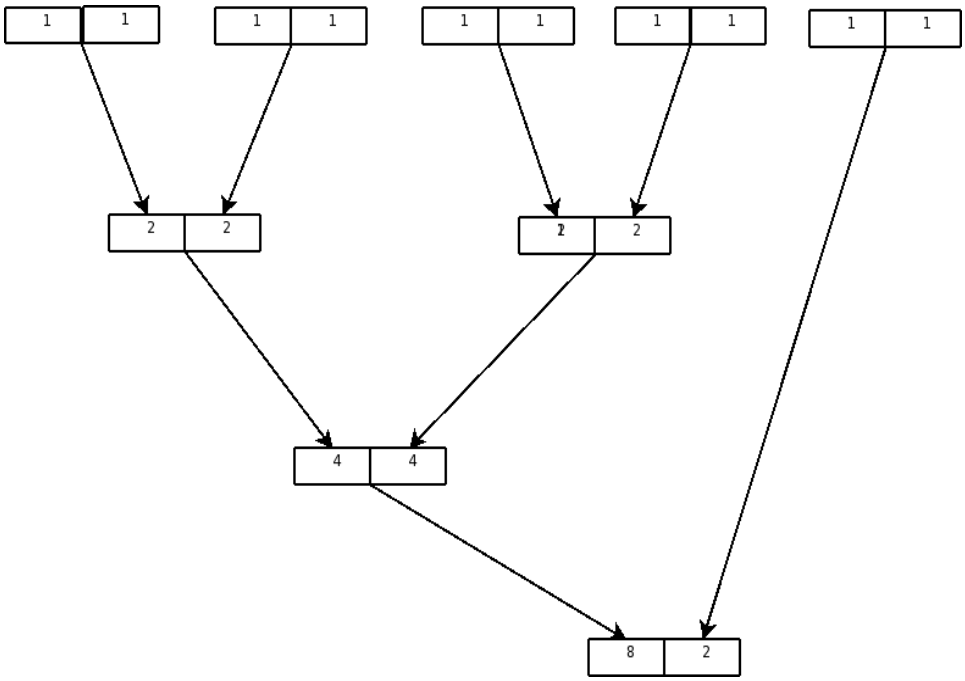

Рисунок 22. Распределение операций суммирования над массивом по вершинам графа.

5.2. Параллельная реализация перцептрона (модельная задача)

Довольно часто создают параллельную реализацию перцептрона [30] большой размерности. В частности, этому посвящена статья [31]. Перцептроны активно используются в задачах распознавания образов, например, для определения по изображению глазного яблока, болен ли человек сахарным диабетом [32]. Для ускорения работы программной реализации перцептрона был предложен следующий алгоритм распараллеливания.

Распараллеливание нейронной сети производилось независимо от вычислительной платформы следующим образом: на каждом слое перцептрона выделяются группы нейронов. Каждая группа нейронов относится к одной вершине граф-программы. Все имеющиеся связи между нейронами слоёв, отнесённых к

соседним группам, объединяются в одно ребро. В результате взаимодействие происходит уже не между отдельными нейронами, а между группами нейронов. Таким образом должен достигаться баланс между передачами данных и вычислениями. Подход к распараллеливанию проиллюстрирован на рис 23.

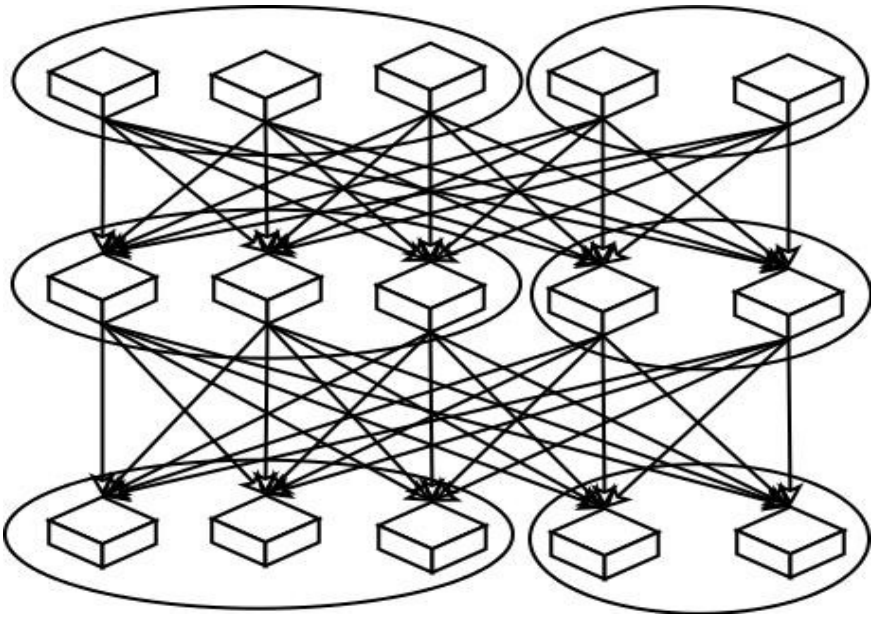

Рисунок 23. Распараллеливание перцептрона, объединение нейронов в группы.

5.3. Частотный фильтр звуковых сигналов

Достаточно популярен подход в обработке сигналов, когда из сигнала удаляются нежелательные частоты. С целью фильтрации частот была написана параллельная реализация Windowed-Sinc фильтров, принадлежащих к классу FIR-фильтров (фильтры с Конечной Импульсной Характеристикой). Эти фильтры хорошо подходят для разделения частот в сигнале, а также выделения нужной частотной полосы. Подробное описание соответствующих фильтров приведено в книгах [33,34]. Как и все FIR-фильтры, Windowed-Sinc фильтры реализуются через операцию свёртки и вычисление быстрого преобразования фурье, которые значительно нагружает процессор вычислениями. Стоит также отметить, что свёртка сигнала из N отсчётов с ядром свёрткой длиной в M отсчётов даст на выходе сигнал в N + M – 1 отсчёт. Оценив время работы последовательного алгоритма, а также обнаружив, что наиболее трудоёмкой операцией является построение быстрого преобразования фурье, которую можно производить независимо для каждого отсчёта исходного сигнала, а зависимость появляется только из-за необходимости накладывать результаты фильтрации и из-за удлинения сигнала на размер ядра свёртки, было принято решение распараллелить алгоритм.

Была принята следующая схема распараллеливания. Исходный файл разрезается на несколько фрагментов большей длины, чем удвоенный размер ядра свёртки. Каждый из фрагментов обрабатывался независимо как одна вершина графа. Затем фрагмент передаётся другой вершине, осуществлявшей сложение “хвоста” предыдущего фрагмента с “головой” текущего фрагмента и осуществлявшей запись полученного нового фрагмента в файл. В случае обработки короткого файла остальные вершины граф-программы, кроме обрабатывающей файл, ничего не делают. Для создания соответствующих графов было создано специальное приложение на языке C, которое осуществляет генерацию граф-программы с заданной степенью параллелизма. Такой способ распараллеливания не является

единственным возможным, в работе [35] обсуждается способ распараллеливания быстрого преобразования фурье. Схема распараллеливания представлена на рис 24.

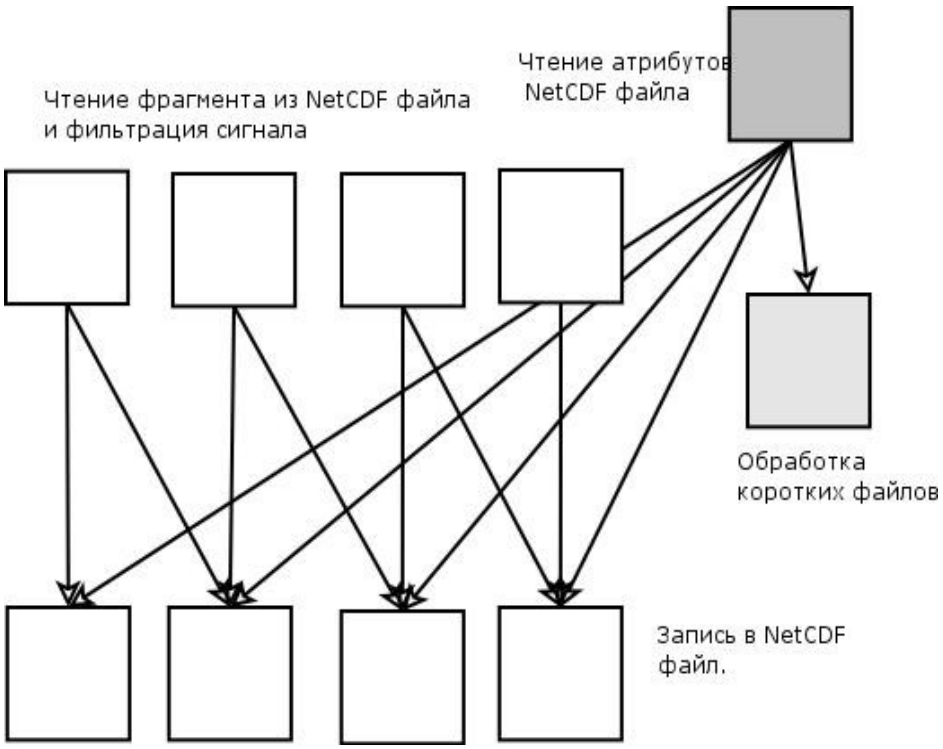

Рисунок 24. Метод распараллеливания для частотного фильтра звуковых сигналов.

5.4. Построение множественного выравнивая нуклеотидных и белковых последовательностей

5.4.1. Общие сведения о выравниваниях

Актуальным для молекулярной биологии и биоинженерии является изучение нуклеотидных и белковых последовательностей. При анализе белковых и нуклеотидных последовательностей важным является инструмент множественных выравниваний. Его цель – таким образом расположить последовательности символов друг под другом, вставляя специальные «пробельные» символы, чтобы наиболее похожие фрагменты последовательностей выровнялись относительно аналогичных фрагментов других последовательностей. Анализируя множественные выравнивания, можно получить сведения об эволюции генов и белков, а также предсказывать функциональные и структурные особенности белков. Первые алгоритмы построения выравниваний строились для пары последовательностей (парные выравнивания). Наиболее популярный метод построения парных выравниваний – это метод динамического программирования, применение которого впервые было описано в статье [20]. Значение выравниваний в реконструкции эволюции биологических последовательностей обсуждается в статье [19]. Важность построения множественных выравниваний для сравнения третичных структур белков освещена в статье [21].

Наиболее популярным средством сравнения 2-х последовательностей является алгоритм BLAST, описанный в статье [24], а наиболее известным методом для построения множественных выравниваний аминокислотных и нуклеотидных последовательностей на текущий момент является алгоритм ClustalW [22], входящий

в пакет программ EMBOSS [26] как программа emma.

5.4.2. Парное выравнивание

Формально определим парное выравнивание. Пусть представлены 2 последовательности символов $S_1 = s_{1,1} s_{1,2} \ldots s_{1,L_1}$ и $S_2 = s_{2,1} s_{2,2} \ldots s_{2,L_2}$ из символов алфавита $A = \{\alpha_1, \ldots, \alpha_n\}$. Парным выравниванием последовательностей S_1 и S_2 назовём матрицу $R = ((r_{i,j}))$ размерности $2 \times L$. В ячейке $r_{i,j}$ может стоять либо символ алфавита A, либо символ «-», который ниже будет называться символом «индел» (объединение слов «инсерция», т.е. вставка, и «делеция», т.е. удаление символов; по английски «indel» см. [23]). Потребуем выполнения трёх условий:

1. Строка $R_1 = r_{1,1} r_{1,2} \ldots r_{1,L}$ матрицы R представляется последовательностью S_1, в которую в некоторых позициях вставлены символы индел «-», следовательно $L_1 \le L$.
2. Строка $R_2 = r_{2,1} r_{2,2} \ldots r_{2,L}$ матрицы R представляется последовательностью S_2, в которую в некоторых позициях вставлены символы индел «-», следовательно $L_2 \le L$.
3. Среди столбцов матрицы R нет таких, которые состоят из одних символов индел «-».

На рисунке 25 можно видеть простейший пример парного выравнивания.

```
aa--ttgttt
||  ||  |||
aaggttattt
```

Рисунок 25. Пример элементарного парного выравнивания.

Для каждого возможного выравнивания последовательностей S_1 и S_2 определяется весовая функция (score – функция). С математической точки зрения цель заключается в поиске выравнивания максимизирующего score функцию.

Конкретный вид весовой функции отражает биологическое содержание выравнивания. Проиллюстрируем смысл выравнивания на примере последовательностей аминокислот (белков). Белок является линейным гетерополимером, состоящим из звеньев 20 типов – 20 типов аминокислотных остатков. Каждому из типов аминокислотных остатков сопоставим символ в алфавите. Таким образом, последовательность символов введённого алфавита однозначно определяет химическую формулу белка. Выравнивание последовательностей белков имеет смысл только в том случае, когда они происходят от одной последовательности предка.

Предполагается, что наблюдаемое состояние аминокислотной последовательности сложилось в процессе эволюции как результат множества локальных мутаций последовательности предка. Локальные мутации бывают трех типов:

1. замена одного символа на другой;
2. делеция (удаление) символа из последовательности;

3. вставка символа алфавита между двумя символами последовательности или на одном из концов последовательности.

Таким образом, если обладать полной информацией о процессе эволюции, то каждый символ последовательности-потомка либо имеет «предка» - символ из последовательности-предка, либо он возник в данной позиции в результате вставки и для него можно указать позицию в последовательности-предке, где произошла вставка; кроме того, часть символов последовательности предка были делетированы, то есть утеряны в процессе эволюции, и не имеют потомка. Таким образом, в идеале, для двух разных последовательностей-потомков, произошедших от одного предка, биологически правильное выравнивание отражает эволюцию последовательностей. В тех случаях, когда заметная часть букв не мутировали ни в одной из двух сравниваемых последовательностей, можно восстановить, или попытаться восстановить, правильное выравнивание (которое практически никогда не известно, так как эволюция белков происходила на протяжении десятков и сотен миллионов лет), основываясь на сходстве последовательностей или их фрагментов. Именно эта цель преследуется при формализации задачи построения оптимального выравнивания.

Для последовательностей белков играет роль также то, что не все замены равновероятны: аминокислоты классифицируются по сходству физико-химических свойств (по заряду, гидрофильности и гидрофобности, размеру и т.д.); замена аминокислотного остатка на сходный ему более вероятна, чем замена на отличающийся по свойствам.

Обычно стремятся максимизировать количество одинаковых символов в одинаковых позициях строк выравнивания и минимизировать штрафы за несовпадения и вставки заглушек из нескольких подряд символов-«индел». В статье [25] более подробно обсуждается принцип построения весовых функций выравнивания.

Рассмотрим более подробно одну из возможных score-функций:

$$score(R_1, R_2) = \sum_{i=1}^{L} substitute(r_{1,i}, r_{2,i}) - \sum_{j=1}^{G} \text{gap\_penalty}(g_j) .$$

Здесь функция $substitute(r_{1,i}, r_{2,i})$ задаёт степень приемлемости того, что один символ алфавита находится напротив другого символа алфавита в определённой позиции выравнивания. Пусть P_a - вероятность встретить символ a в последовательности, а $P_{a,b}$ - вероятность замены символа a на символ b в процессе эволюции последовательности. Тогда функция подстановок задаётся следующим образом: $substitute(a, b) = \log\left(\frac{p_{a,b}}{p_a \cdot p_b}\right)$, где $P_{a,b}$ - элемент матрицы.

$$P_{\alpha_i,\alpha_j} \in \begin{bmatrix} p_{\alpha_1,\alpha_1} & p_{\alpha_1,\alpha_2} & \cdots & p_{\alpha_1,\alpha_n} \\ p_{\alpha_2,\alpha_1} & p_{\alpha_2,\alpha_2} & \cdots & p_{\alpha_2,\alpha_n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{\alpha_n,\alpha_1} & p_{\alpha_n,\alpha_2} & \cdots & p_{\alpha_n,\alpha_n} \end{bmatrix}, \quad где \quad \forall i, j \in \{1, \ldots, n\} \quad \alpha_i \in A, \alpha_j \in A.$$

Другая функция $\text{gap\_penalty}(g_j)$ задаёт размер штрафа за так называемые «гэпы» – последовательности инделов. Функция штрафов за гэпы вычисляется для всех гэпов в обоих строках выравнивания. Здесь G общее число гэпов для обоих строк выравнивания, а g_j – один из гэпов. Функция штрафов вычисляется следующим образом: $\text{gap\_penalty}(g) = cost_{begin} + cost_{end} - length(g) \cdot cost_{extention}$. Здесь

$cost_{begin}$, $cost_{end}$ – штрафы за инициацию гэпа. Следующий параметр, присутствующий в формуле: $cost_{extension}$ – штраф за удлинение гэпа на один символ. От штрафа за удлинение, как правило, требуется, чтобы он был меньше, чем штрафы за инициацию гэпа. Таким образом исключается появление большого количества коротких гэпов.

После задания score-функции ищется такая пара строк R_1 и R_2 в матрице R, на которых данная функция принимает максимальное значение. Поиск соответствующих строк обычно осуществляется методом динамического программирования.

5.4.3. Множественное выравнивание

По аналогии с парным выравниванием определяют понятие множественного выравнивания. В случае множественного выравнивания рассматривается не пара последовательностей, а сразу же группа последовательностей. Для этой группы аналогично с парным выравниванием строится матрица $R=((r_{i,j}))$ размерности MxL, где M - число выравниваемых последовательностей.

Если для парного выравнивания метод динамического программирования даёт точное решение, то для множественного выравнивания многомерный аналог метода динамического программирования имеет экспоненциальную временную сложность от числа последовательностей. Задача построения множественного выравнивания является NP-трудной. Обычно в случае построения множественного выравнивания используют эвристические алгоритмы, которые не дают точного решения в смысле оптимизации некоторой весовой функции, однако имеют меньшую сложность и выдают приемлемый в биологическом смысле результат. Обычно для упрощения стремятся максимизировать число одинаковых символов в столбце матрицы при минимизации числа и длин вставляемых в строки заглушек из символов болванок. Далее будет рассмотрен алгоритм, описанный в статье [23]. Этот алгоритм реализован в программном средстве MUSCLE, которое можно получить с сайта http://www.drive5.com/muscle.

Алгоритм MUSCLE разбит на несколько стадий. На первой стадии создаётся матрица похожести всех последовательностей друг на друга, по которой затем будет построено множественное выравнивание. Далее производится кластеризация последовательностей по степени схожести. В результате строится двоичное кластерное дерево. Для построения дерева используются алгоритмы neighbour-joining [27] и UPGMA [80]. Степень похожести между отдельными последовательностями вычисляется описанным ниже способом. Далее по дереву строятся сперва парные выравнивания последовательностей, приписанных листьям, а затем выравнивания профилей выравнивания для всех внутренних вершин дерева. Выровненные последовательности, входящие в профиль, нельзя двигать друг относительно друга, при этом гэпы вставляются сразу для всех строк, входящих в профиль выравнивания. На второй стадии производится улучшение выравнивания, построенного первой стадией алгоритма MUSCLE. Улучшение выравнивания производится на основе улучшения построенного кластерного дерева. По улучшенному кластерному дереву, в тех местах, где это необходимо, производится перевыравнивание соответствующих профилей выравнивания. Вторая стадия может быть повторена несколько раз. На третьей стадии производится окончательное построение выравнивания по профилям и дереву.

Схожесть между последовательностями, входящими в выравнивание на первой стадии алгоритма, вычисляется следующим образом. Все последовательности разделяются на множество фрагментов длины k (k-меры), далее вычисляется встречаемость каждого k-мера во всех последовательностях выравнивания. Схожесть

между последовательностями вычисляется по формуле:

$$similarity(S_i, S_j) = \sum_{m=1}^{M} \frac{min(frequensy_{S_i}(\tau_m), frequensy_{S_j}(\tau_m))}{min(length(S_i), length(S_j)) - k + 1}$$

где τ_m - один из k-меров, выделенных среди последовательностей, $frequensy_S(\tau)$ число раз, которое τ встречался в последовательности S , а M - общее число различных выявленных k-меров для всех последовательностей, составляющих выравнивание.

С целью распараллеливания алгоритма MUSCLE была внесена модификация в оригинальный алгоритм на стадии построения профилей выравнивания по кластерному дереву. На основе кластерного дерева строится граф-программа. Вершинам-истокам соответствуют одна или несколько последовательностей. Эти вершины создают профиль попавшего к ним подмножества последовательностей. От них направляется дуга к внутренним вершинам. Внутренние вершины выравнивают пару профилей между собой, в результате получается новый профиль, который затем отправляется очередной внутренней вершине или в вершину-сток. В вершину-сток алгоритм попадает в случае, когда очередной построенный профиль выравнивания совпадает по числу последовательностей с общим числом выравниваемых последовательностей. Таким образом, граф-программа по отношению к кластерному дереву строится от листьев кластерного дерева к корню. Для повышения эффективности работы параллельной программы производится сжатие нижних уровней кластерного дерева (близких к листьям) в один уровень граф-программы. Для этого в алгоритм вводится специальный параметр, который задаёт число сжимаемых уровней кластерного дерева. В результате в создаваемой граф-программе, в отличии от исходного кластерного дерева, в вершине-истоке может оказаться не одна последовательность, а целая группа последовательностей. Выигрыш во времени работы программы достигается за счёт того, что каждую вершину граф-программы можно исполнять на своём процессоре. Максимальное количество одновременно обрабатывающихся вершин будет ограничиваться числом вершин, приписанных одному слою граф-программы, и числом процессоров в многопроцессорной системе, доступных для запуска созданной параллельной программы.

6. Тестирование системы «PARUS»

6.1. Описание машин, на которых производилось тестирование

Система «PARUS» была установлена на следующие многопроцессорные вычислительные системы: МВС-1000М, IBM eServer pSeries 690 Regatta и Fujitsu Sun PRIMEPOWER 850. Примеры, описанные выше, были откомпилированы на этих машинах, и далее приведены результаты измерения времени работы программных реализаций. Архитектурные особенности данных многопроцессорных систем рассмотрены в приложениях.

6.2. Результаты тестирования коммуникационной среды

В процессе настройки «PARUS» на многопроцессорную систему запускаются тесты, измеряющие в разных режимах задержки при передаче MPI-сообщений между процессорами в многопроцессорной системе. Для рассматриваемых многопроцессорных систем накопилось некоторое количество тестовых данных, часть из которых хотелось бы проиллюстрировать здесь. Результаты нашей

деятельности по тестированию коммуникационных сред многопроцессорных вычислительных комплексов обсуждены в работах [15, 36, 37].

На первом этапе исследовался характер влияния длины сообщения на время его доставки принимающей стороне. На рисунке 26 приведён график зависимости времени передачи сообщения от размера коротких сообщений. График, помеченный ромбами, соответствует машине Regatta, а график, помеченный квадратами, - МВС-1000М. Здесь рассмотрены результата теста all_to_all, который запускался на 16 процессорах в случае Regatta и на 64 в случае МВС-1000М.

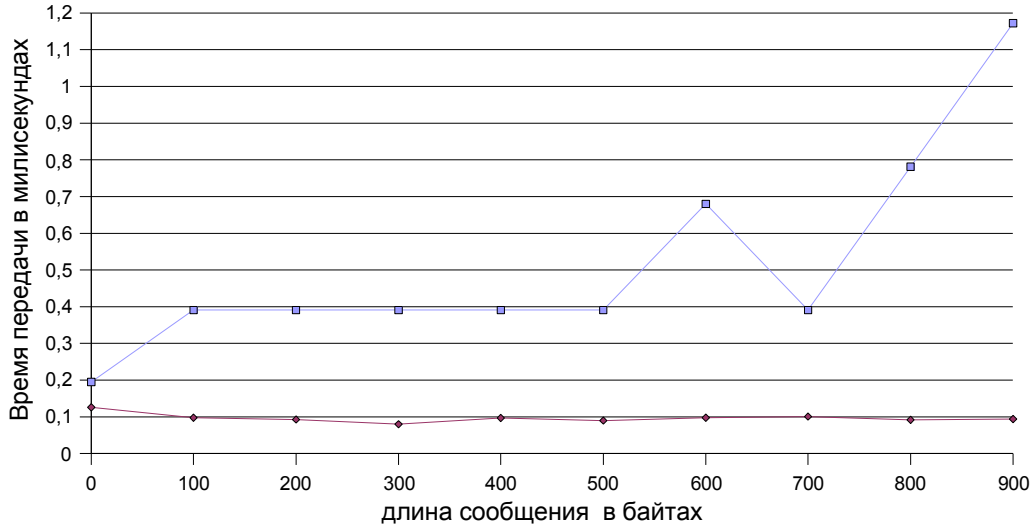

Рисунок 26. Зависимость времени передачи от размера коротких сообщений для теста all_to_all.

Как иллюстрирует график, задержки при передаче коротких сообщений по сети Myrinet 2000 в МВС-1000М больше, чем при передаче через многоуровневую систему доступа в память IBM pSeries 690 «Regatta», однако разница не столь велика, как должна бы быть. Это объясняется тем, что накладные расходы на инициацию передачи коротких сообщений много больше самого времени передачи сообщения.

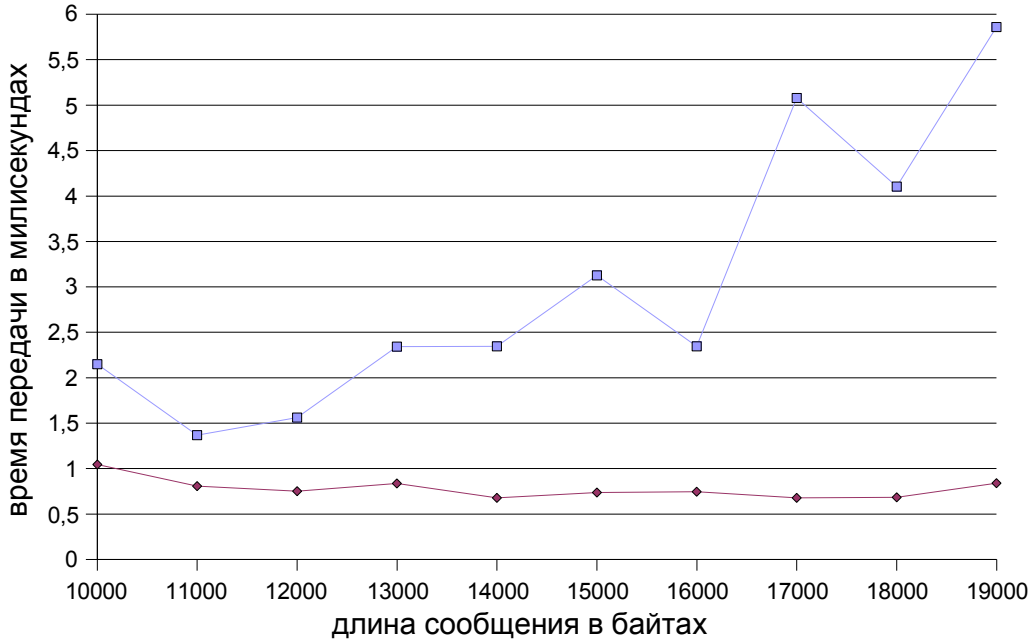

Рисунок 27. Зависимость времени передачи от сообщений среднего размера для теста all_to_all.

На рисунке 27, также, как и на рисунке 26, приведён график зависимости времени передачи от сообщений среднего размера. Здесь график, помеченный ромбами, соответствует машине Regatta, а график, помеченный квадратами, - МВС-1000М. Рассмотрены результаты теста all_to_all, который так же, как и в предыдущем случае, запускался на 16 процессорах в случае машины regatta и на 64 случае МВС-1000М. Интересной представляется следующая особенность: в то время как задержки при передаче сообщений на Regatta практически не растут, задержки на МВС-1000М начинают активно расти с ростом размера сообщения. По сравнению с короткими сообщениям задержки выросли в 2 раза, но сам характер зависимости остался прежним.

Видимо, в этом месте можно ощутить разницу в архитектуре Regatta и МВС-1000М. Для Regatta основной причиной задержек при передаче данных будет являться переполнение кэш второго и третьего уровней. Для МВС-1000М источником задержек будет пропускная способность канала и то, какой объём данных помещается в кадр при передаче пакета между 2-мя модулями. Поэтому в случае с Regatta кэш не успевает переполнится. В случае с МВС-1000М реализация MPI, по-видимому, устроена таким образом, что для коротких пакетов поле MTU выставляется приблизительно в 500 байт (поле в заголовке пакета, отвечающее за фрагментацию данных), что приводит к тому, что до 500 байт всё помещается в один кадр, а с увеличением размера начинает дробиться, и это приводит к началу роста времени передачи.

Ещё одна интересная особенность работы МВС-1000М была выявлена при помощи «шумящих»тестов. Фиксируем длину сообщения, время передачи которого мы измеряем, и на фоне процесса передачи этого сообщения инициализируем передачу набора фоновых сообщений. Будем постепенно увеличивать уровень фона, увеличивая размер «шумового» сообщения. На рисунке 28 приведён пример такой зависимости для МВС-1000М, где задействовано 128 процессоров при длине целевого сообщения 5000 байт.

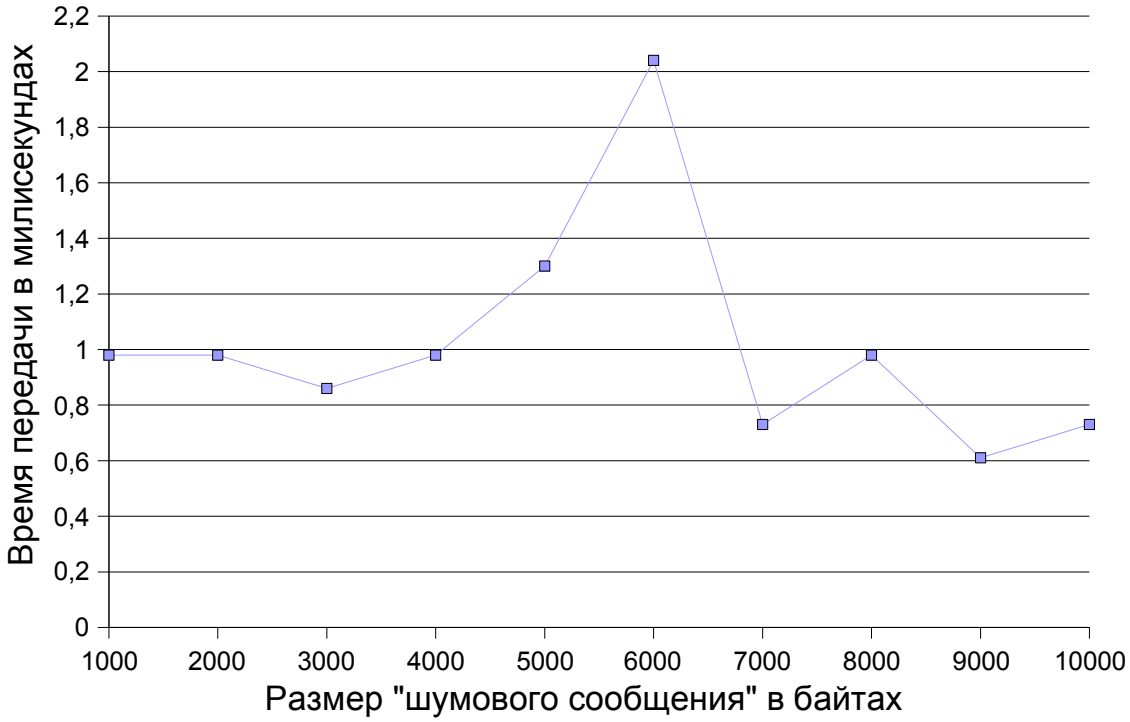

Рисунок 28. Тест test_noise, зависимость времени передачи от размера "фонового" сообщения, длина "целевого" сообщения – 5000 байт.

Достаточно интересен пик на графике, который возникает при размере фонового сообщения, близком к 6000 байт. К сожалению, выяснить причину такого поведения не удалось. Для выяснения деталей поведения необходимо учитывать большое число параметров, таких, как особенности реализации MPI на Regatta, детали реализации протокола Myrinet2000, а также особенности поведения коммуникационного оборудования. В качестве одной из гипотез можно предположить следующее: в начальный момент, когда «шумовые» сообщения короче целевых, «шумовые» сообщения мешают. Далее, с увеличением размера «шумового» сообщения, коммуникационная среда принимает решение отложить передачу длинного «шумового» сообщения и передаёт короткое «целевое». Этим объясняется уменьшение времени передачи «целевого» сообщения с ростом длины «шумового» сообщения, а так же наличие пика на графике.

В следующем тесте на МВС-1000М был фиксирован размер «шумовых» сообщений в 1000 байт и зафиксировано число процессоров в 64, при этом размер «целевого» сообщения менялся. Независимо от этого теста был проведён тест, когда «целевые» сообщения передавались без «шума». Результаты соответствующего тестирования приведены на рисунке 29. Здесь график, помеченный ромбами, – время передачи сообщений без «шума». График, помеченный квадратами, – время передачи на фоне шума.

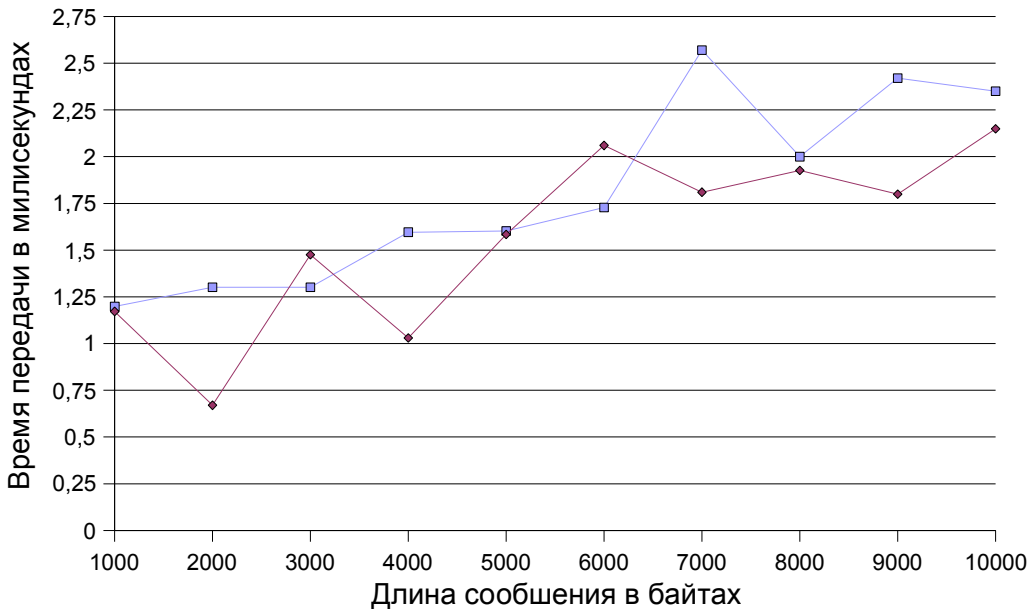

Рисунок 29. Влияние "шума" на скорость.

По-видимому, можно считать, что для 64-х процессоров на МВС-1000М наличие «шума» замедляет скорость передачи «целевого» сообщения, однако для рассмотренных длин сообщений влияние незначительно.

Рассмотренные примеры показывают, насколько сложно поведение многопроцессорных систем при передаче данных по каналам связи между процессорами многопроцессорной системы. Примеры показывают, что этап предварительного тестирования многопроцессорной системы на предмет скоростей передачи данных по каналам связи совершенно необходим. Получается, что в общем случае нельзя построить функцию, описывающую зависимость времени передачи сообщения от номера процессора и длины сообщения, пригодную для многопроцессорных систем произвольной архитектуры.

6.3. Особенности реализаций примеров использования «PARUS» на многопроцессорных системах

Некоторые результаты измерений производительности были обсуждены в работе [38]. Перцептрон был протестирован на машине Regatta.

Для тестирования такого подхода к созданию параллельных программ написана параллельная программа как граф зависимости по данным, где в качестве базовой модели нейронной сети для тестирования был выбран трехслойный перцептрон с большим количеством вершин (до 18500 в одном слое). Для генерации граф-программы было написано специальное приложение на языке С, которое по заданным характеристикам нейронной сети, а также параметрам группировки нейронов создаёт текстовый файл с описанием графа.

6.3.1. Особенности исполнения параллельной реализации перцептрона на машине Regatta

На рисунке 30 приведены результаты измерений времени работы параллельной реализации 3-слойного перцептрона на многопроцессорной системе Regatta. Разделение на группы производилось так, чтобы один слой сети целиком занимал всё множество доступных процессоров.

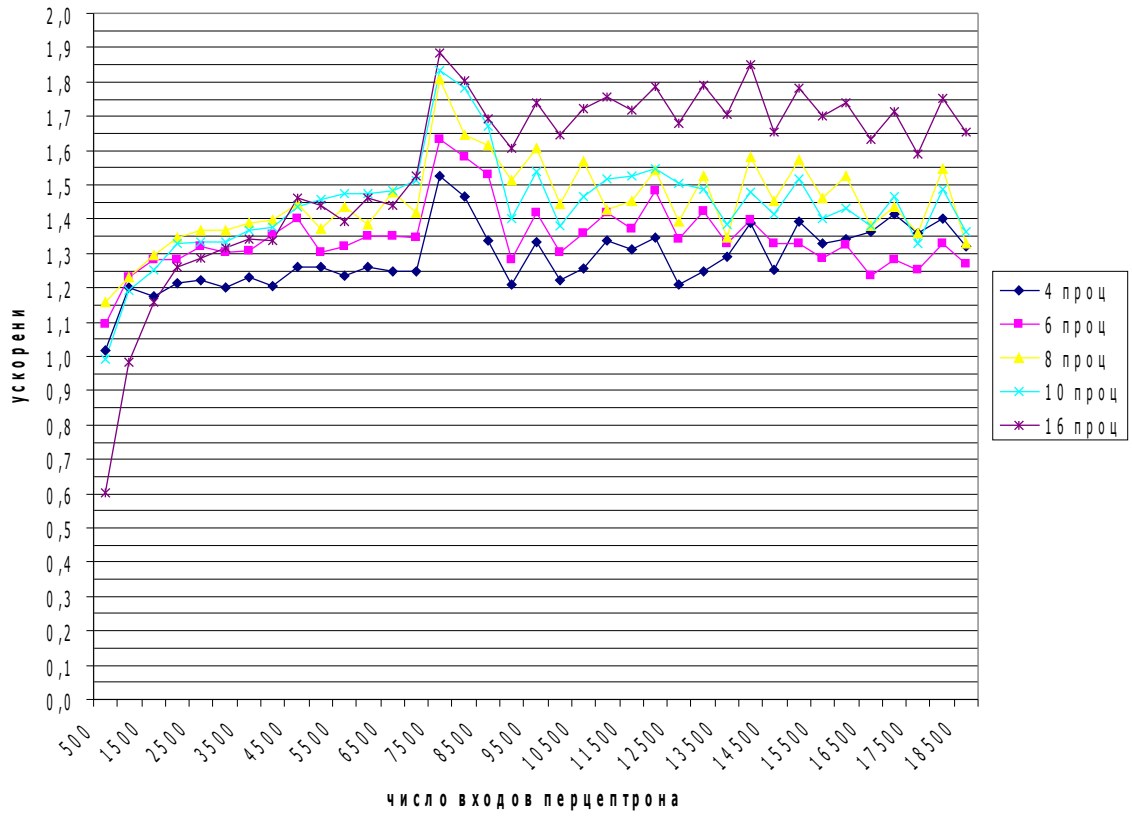

Рисунок 30. Зависимость ускорения от числа входов по отношению к 2-процессорной реализации параллельного перцептрона.

Как показывают графики, приведённые на рисунке 30, в целом добиться масштабируемости не получается. Максимальное ускорение относительно 2-процессорной реализации получается при 7500 входах нейронной сети на 16 процессорах. Как видно из графиков, при числе входов нейронной сети от 500 до 4500 16-ти процессорная реализация проигрывает реализациям с меньшим числом процессов. Это связано с тем, что на процессор приходится небольшой объём

вычислений по сравнению с объёмом передач данных.

Однако если нейроны объединять в группы так, что на процессоры многопроцессорной системы помещается сразу 2 слоя нейронной сети, ситуация довольно сильно меняется, что показано на рисунке 31.

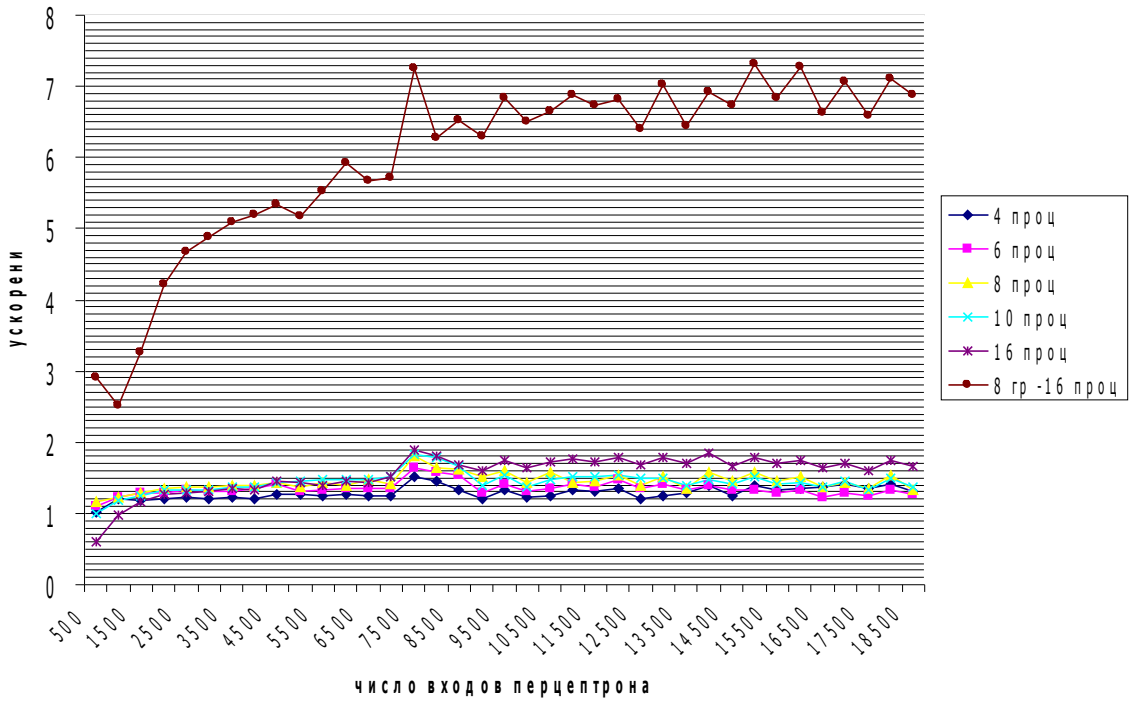

Рисунок 31. Зависимость ускорения от числа входов по отношению к 2-процессорной реализации параллельного перцептрона (на процессоры помещается сразу 2 слоя).

На рисунке 31 представлен способ разбиения на группы для машины Regatta, когда каждый слой нейронной сети делится на 8 групп, а полученное параллельное приложение запускается на 16 процессорах. Как и в случае, представленном на рисунке 30, ускорение относительно двухпроцессорного варианта растёт вместе с ростом числа входов нейронной сети от 1000 до 7500 нейронов в слое, затем, с дальнейшим ростом числа нейронов в слое перцептрона, стабилизируется приблизительно на уровне семи. Эта ситуация говорит о том, что при небольшом числе нейронов, приходящихся на слой нейронной сети, процессоры оказываются недогруженными.

6.3.2. Исследование эффективности реализации распределённой операции над массивом для МВС-1000М

Следующая задача, эффективность реализации которой на «PARUS» была исследована – это модельная задача одновременного поиска максимума, минимума и суммы элементов массива чисел с плавающей точкой. По алгоритму, описанному ранее для данной задачи, создаётся граф-программа. Полученная граф-программа компилировалась и исполнялась на МВС-1000М. Параметры, по которым строилась граф-программа, таковы: размерность массива – 1 миллиард ячеек, размер одного фрагмента 1 миллион ячеек. На рисунке 32 приведён график зависимости ускорения от числа процессоров для серии запусков программы на МВС-1000М .

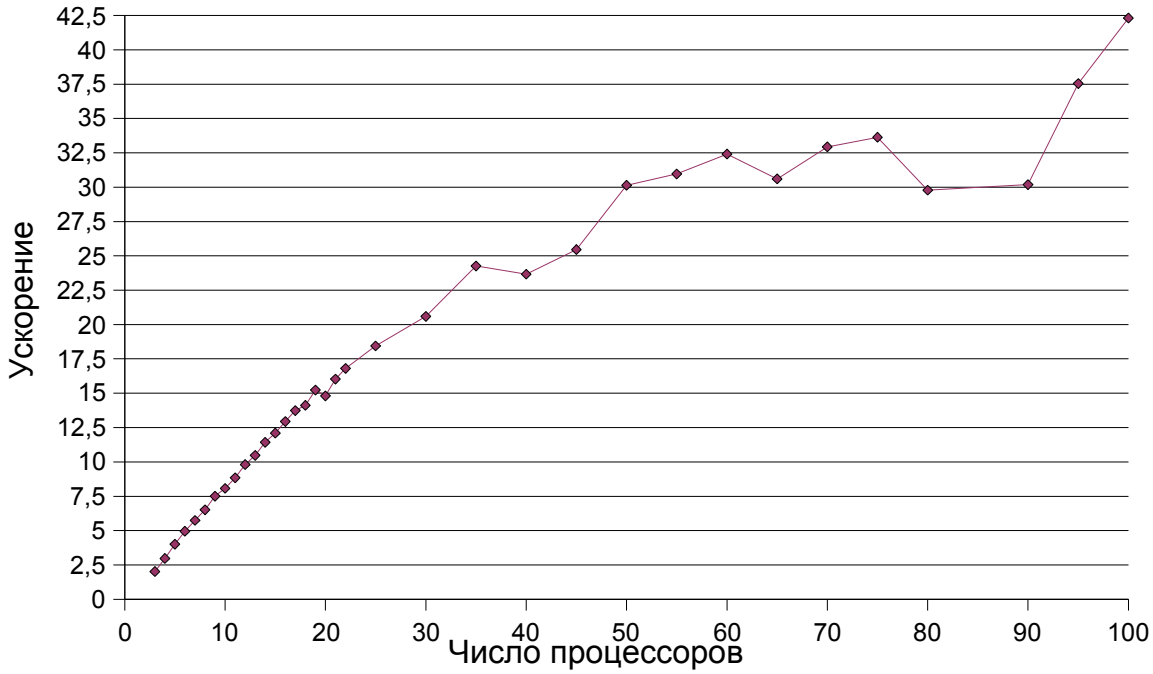

Рисунок 32. Зависимость ускорения от числа процессоров для распределённой операции над массивом. (Размерность массива 100 млн. ячеек.)

Как видно на рисунке 32, до 30 процессоров ускорение можно считать линейным, причём с коэффициентом наклона больше 0,7. Однако далее кривая становится более пологой и, хотя ускорение растёт, но значительно медленнее, то есть эффективность падает. По-видимому, это связано с тем, что число процессоров увеличивается, а размерность задачи остаётся той же. Процесс падения эффективности показан на рисунке 33.

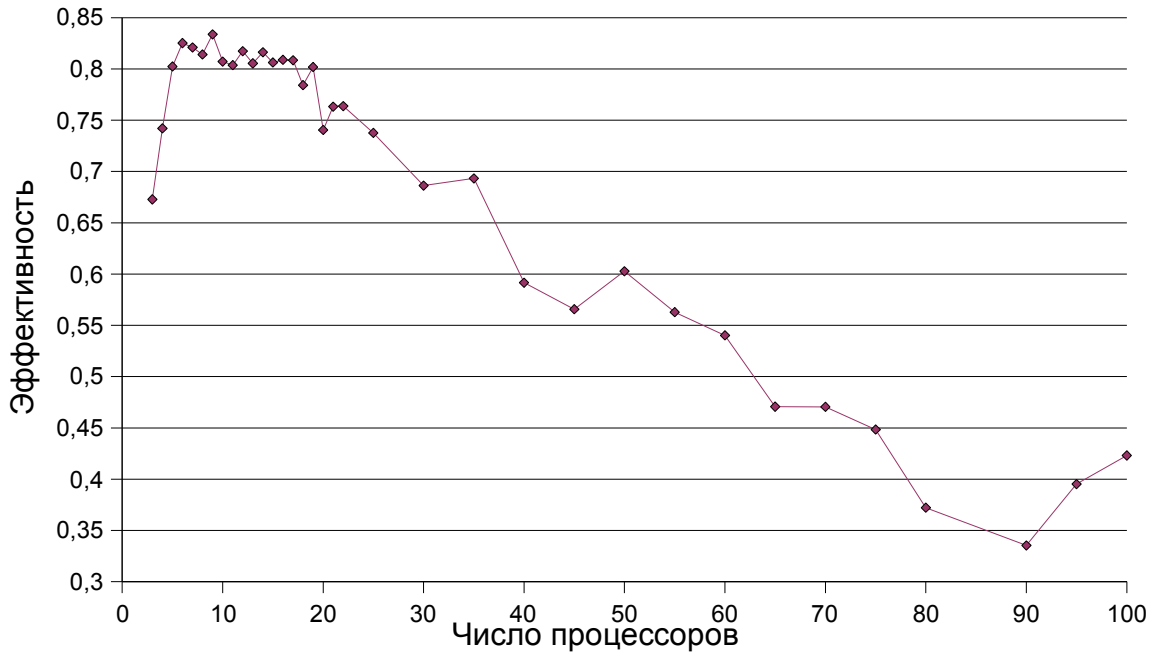

Рисунок 33. Зависимость эффективности от числа процессоров для распределённой операции над массивом. (Размерность массива 100 млн. ячеек.)

На графике можно выделить группу от 5 до 20 процессоров, для которой

эффективность наиболее высока. Следующая группа – от 20 до 90 процессоров. Здесь эффективность падает, а затем опять начинает расти. Несмотря на падение эффективности, ускорение продолжает расти, тем самым точка, когда большее число процессоров для решения задачи привлекать не имеет смысла, ещё не достигнута.

6.3.3. Параллельный способ выравнивания всех LTR5 в человеческом геноме

Результатом многолетних исследований в молекулярной биологии стало открытие того факта, что размер генома слабо коррелирует со сложностью организма. К примеру, геном человека в 200 раз меньше генома амёбы. Причина такого противоречия объясняется тем, что размер кодирующей белок части генома человека не превосходит 1,5% от размера всего генома. То есть большая часть генного материла не кодирует белок и «вообще не нужна», однако в геноме существуют неоднократно повторяющиеся участки – повторы. Повторы занимают приблизительно 50% от всего размера человеческого генома. Сведения о структуре генома и повторах приведены в статье [39]. Как часть работ по совместному проекту Института физико-химичесой биологии им. А.Н. Белозерского МГУ и Людвиговского института раковых исследований "Ludwig Institute for Cancer Research" (грант CRDF RB01277-MO-2), проводилось исследование одного из типов повторов в человеческом геноме, а именно так называемого LTR класса 5. Для более детальной классификации данного класса повторов необходимо строить множественное выравнивание по всем известным в человеческом геноме LTR5, что было сделано с использованием «PARUS». Одним из важных этапов было создание собственной базы данных с информацией по LTR5 [40,41,43].

Последовательность повтора LTR (Long Terminal Repeat) содержит всю необходимую информацию для инициации процесса трансляции (синтеза матричной РНК) с того участка генома, где встречается данный повтор. LTR является одной из частей ретровируса. На рисунке 34 представлена схема генома ретровируса, или схема участка генома человека после встраивания в него ДНК, синтезированной ретровирусом. Повторы LTR ограничивают участок с генами, куда входят: gag – ген, отвечающий за создание собственного механизма синтеза белка по матричной РНК, pol – ген, кодирующий белок для синтеза ДНК по РНК, а также env – ген, отвечающий за создание оболочки ретровируса для встраивания в клетку. Также имеется ряд вспомогательных генов.

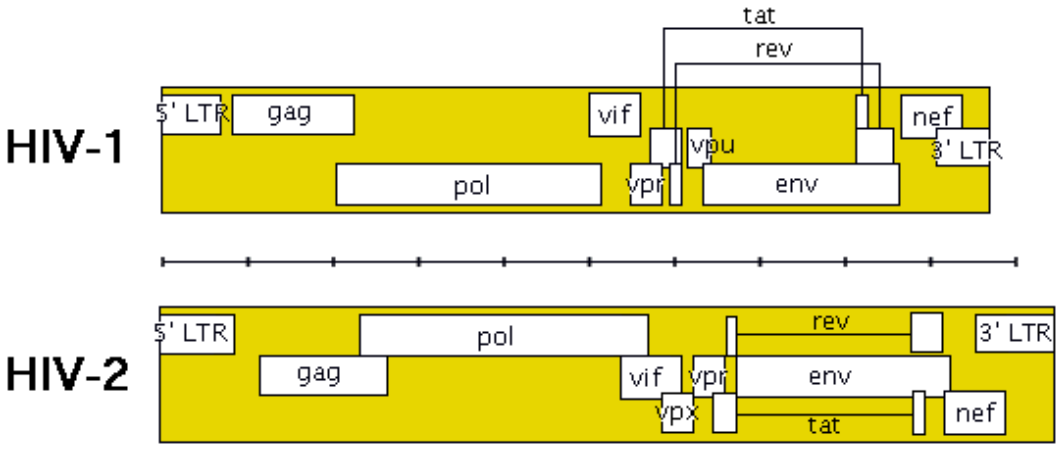

Рисунок 34. Роль LTR. Окрестности LTR в РНК ретровируса.

Повторы чаще всего не являются чем-то важным для функционирования

генома (иными словами, они были выключены в процессе эволюции), однако при этом они содержат очень важную информацию о биологических процессах, происходивших и происходящих с данным геномом. Повторы несут на себе важный «палеонтологический» отпечаток и могут способствовать пониманию процессов эволюции генома. Как активные элементы, которыми они могут стать вследствие мутаций и перестроек генома, повторы могут приводить к появлению новых генов, модификации существующих и изменению порядка генов. Таким образом, изучая повторы в человеческом геноме, можно найти причины и, возможно, способы лечения наследственных и раковых заболеваний [42].

С целью дальнейшего изучения LTR5 и их классификации на машине PRIMEPOWER 850 была установлена система PARUS. Затем с её помощью по методу, описанному ранее, была создана параллельная программная реализация для выравнивания последовательностей.

В однопроцессорном варианте выравнивание всех известных LTR5 в человеческом геноме занимает 1 час 8 минут. В многопроцессорном варианте на двенадцати процессорах оно же занимает 28 минут, что в 2,4 раза быстрее, чем последовательный вариант. На первый взгляд, эффективность параллельной реализации не очень высокая. Однако, ускорение параллельной реализации для конкретного множества выравниваемых последовательностей будет очень сильно зависеть от сбалансированности построенного кластерного или филогенетического дерева. Кроме всего прочего, в случае уточнения информации о последовательности LTR5 или обнаружения мировой общественностью нового экземпляра LTR5 в геноме придётся перестраивать выравнивание. С большим количеством последовательностей или с последовательностями большой длины алгоритм MUSCLE, изначально непараллельный, может работать значительный промежуток времени – до нескольких суток; даже незначительное ускорение для параллельной программы в этом случае может значительно ускорить работу человека.

6.3.4. Web интерфейс к построителю выравниваний

Для облегчения доступности параллельного способа выравнивания нуклеотидных и аминокислотных последовательностей был создан специальный сайт, воспользовавшись которым любой желающий может построить выравнивание своего собственного множества последовательностей на многопроцессорной системе PRIMEPOWER 850. На рисунке 35 приведён внешний вид web интерфейса.

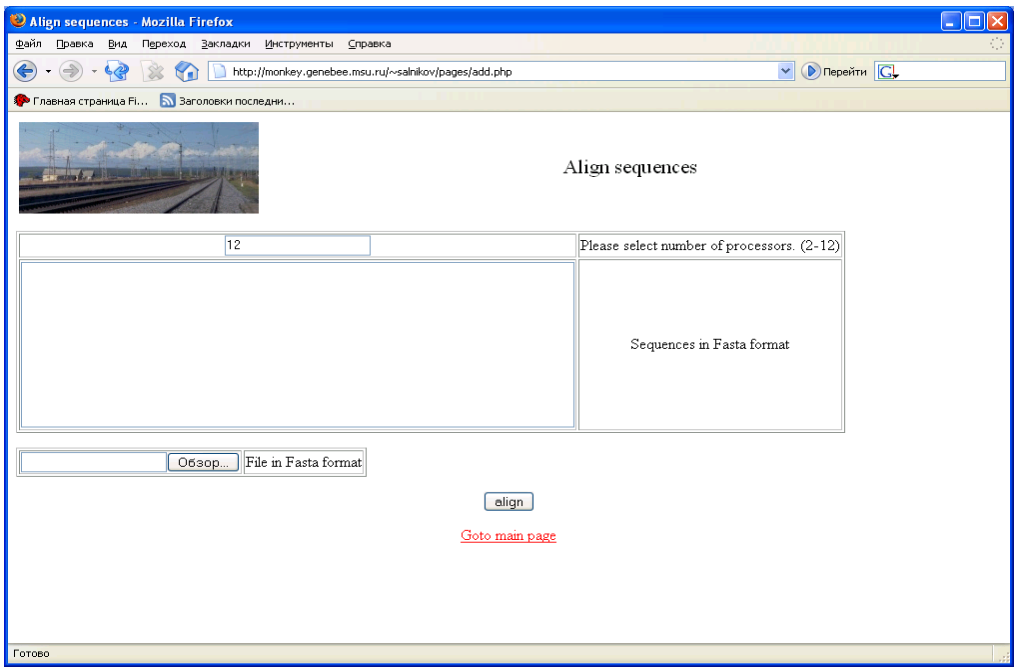

Рисунок 35. Web интерфейс к параллельной программе, предназначенной для построения множественного выравнивания последовательностей.

Web интерфейс к программе, осуществляющей выравнивание белковых и нуклеотидных последовательностей параллельным образом на многопроцессорной системе доступен по адресу: **http://monkey.genebee.msu.su/~salnikov**.

7. Результаты и выводы

7.1. Достоверность и практическая значимость результатов диссертационной работы

Практическая применимость и эффективность созданной системы «PARUS» для разработки и исполнения параллельных программ показана на примере решения ряда модельных и практических задач на современных параллельных вычислительных системах (IBM pSeries 690, МВС-1000м, и др.).

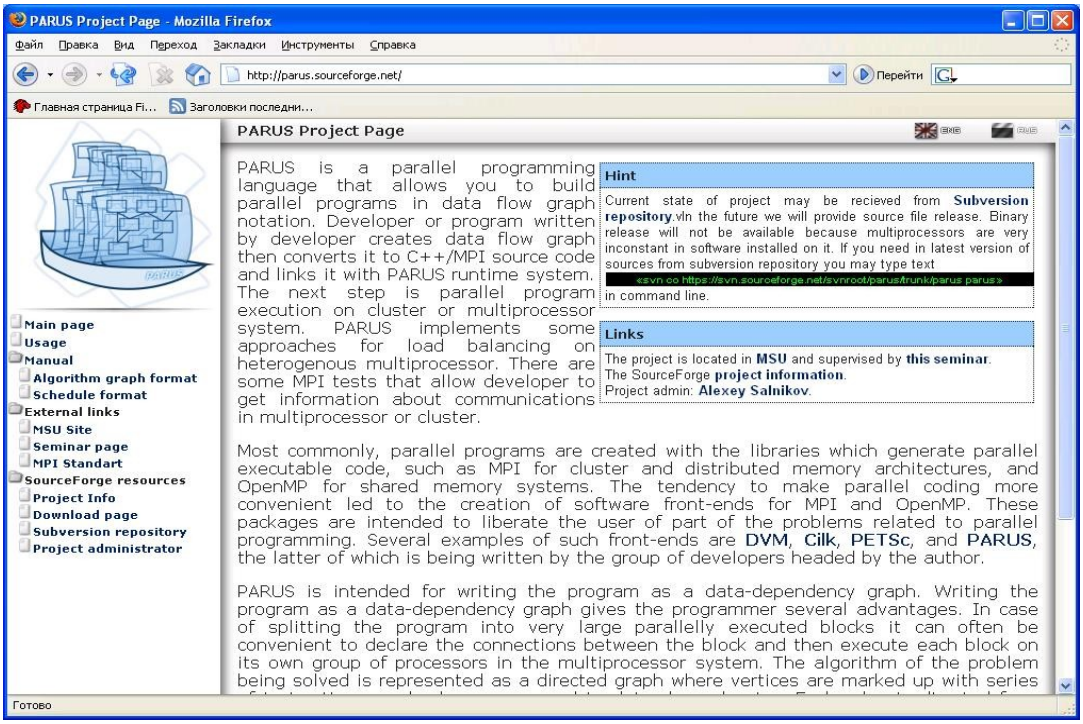

Рисунок 36. Стартовая страница проекта PARUS на sourceforge.net.

Система «PARUS», построенная на основе технологию MPI, оформлена как проект с открытым исходным кодом и доступна в Internet (**http://parus.sf.net**). На рисунке 36 показан внешний вид заглавной страницы проекта PARUS.

7.2. Основные результаты диссертационной работы

Разработан метод построения параллельных программ и язык их описания как граф-схемы потока данных, позволяющие абстрагироваться от конкретной технологии передачи сообщений.

Разработаны и реализованы алгоритмы управления процессом исполнения параллельной программы с учётом структуры программы, динамики обменов данными и текущего состояния многопроцессорной системы.

На основе предложенных метода, языка и алгоритмов создана система поддержки этапов разработки и исполнения параллельных программ.

Список литературы

[1] Коновалов Н.А., Крюков В.А., Сазанов Ю.Л.. “C-DVM - язык разработки мобильных параллельных программ”. Программирование N 1, 1999. стр. 54-65.

[2] Supercomputing Technologies Group MIT Laboratory for Computer Science "Cilk 5.3.2 Reference Manual" 2001. стр. 1-42.

[3] Satish Balay and Kris Buschelman and Victor Eijkhout and William D. Gropp and Dinesh Kaushik and Matthew G. Knepley and Lois Curfman McInnes and Barry F. Smith and Hong Zhang “{PETS}c Users Manual”, ANL-95/11 - Revision 2.1.5, Argonne National Laboratory 2004

[4] Satish Balay and Victor Eijkhout and William D. Gropp and Lois Curfman McInnes and Barry F. Smith "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries", "Modern Software Tools in Scientific Computing"

Birkh@user Press, pp. 163-202.

[5] A.Lastovetsky. Parallel Computing on Heterogeneous Networks. John Wiley & Sons, 423 pages, 2003, ISBN: 0-471-22982-2.

[6] IBM “Performance Tools Guide and Reference” SC23-4859-03 pp. 3-56.

[7] Julian Seward, Nicholas Nethercote. “Using Valgrind to detect undefined value errors with bit-precision.” “Proceedings of the USENIX'05” Annual Technical Conference, Anaheim, California, USA, April 2005.

[8] Nicholas Nethercote,Jeremy Fitzhardinge “Bounds-Checking Entire Programs Without Recompiling” Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004), Venice, Italy, January 2004.

[9] Nicholas Nethercote, Julian Seward “Valgrind: A Program Supervision Framework” Electronic Notes in Theoretical Computer Science 89 No. 2, 2003.

[10] James Reinders “VTune™ Performance Analyzer Essentials” Measurement and Tuning Techniques for Software Developers Intel-Press ISBN 0-9743649-5-9

[11] S. Graham, P. Kessler, and M. McKusick “Gprof: A Call Graph Execution Profiler” “Proceeedings of the SIGPLAN '82 Symposium on Compiler Construction” volume 17 number 6 pp 120-126.

[12] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Debra Hensgen and Sahra Ali. “Representing task and machine heterogeneities for heterogeneous computing systems.“ Tamkang Journal of Science and Engineering. 2000. Vol. 3, №3, P. 195-207.

[13] Julita Corbalan, Xavier Martorell and Jesus Labarta. “Improving Gang scheduling through job performance analysis and malleability” Proceedings of the 15th international conference on Supercomputing pp. 303 - 311 Sorrento, Italy 2001. ISBN:1-58113-410-X

[14] Сальников А.Н. “Некоторые технические аспекты инструментальной системы для динамической балансировки загрузки процессоров и каналов связи” “Программные системы и инструменты” тематический сборник N 3 факультета ВМиК МГУ им. Ломоносова 2002г. ISBN 5-89407-149-6 стр. 152-164.

[15] Булочникова Н.М., Горицкая В.Ю., Сальников А.Н. “Методы тестирования производительности сети с точки зрения организации вычислений” труды Всероссийской научной конференции “Научный сервис в сети Интернет 2004” Издательство Московского университета 2004г. ISBN 5-211-05007-X стр. 221-223.

[16] Булочникова Н.М., Сальников А.Н., “Разработка прототипа CASE средства создания программ для гетерогенных многопроцессорных систем “PARUS””, “программные системы и инструменты” тематический сборник N4 факультета ВМиК МГУ им. Ломоносова 2003г. Издательский отдел факультета ВМиК МГУ. ISBN-5-89407-170-4 стр. 203-209.

[17] Сальников А.Н., Сазонов А.Н., Карев М.В. “Прототип системы разработки приложений и автоматического распараллеливания программ для гетерогенных многопроцессорных систем.” “Вопросы Атомной Науки и Техники” серия: “Математическое моделирование физических процессов” министерство российской федерации по атомной энергии ФГУП, Российский федеральный ядерный центр – ВНИИЭФ, научно-технический сборник, выпуск N1 2003 г. стр. 61-68.

[18] Сальников А.Н. “Разработка инструментальной системы для динамической

балансировки загрузки процессоров и каналов связи” ”Высокопроизводительные параллельные вычисления на кластерных системах.” Материалы Международного научно-практического семинара. Изд-во Нижегородского университета, 2002г. ISBN 5-85746-681-4 стр. 159-167.

[19] Feng D.F., Doolittle R.F. “Progressive sequence alignment as a prerequisite to correct phylogenetic trees” Journal of molecular evolution. 987; Volume 25, Issue 4, pp. 351-360. ISSN: 0022-2844.

[20] Saul B. Needleman and Christian D. Wunsch “A general method applicable to the search for similarities in the amino acid sequence of two proteins” Journal of Molecular Biology Volume 48, Issue 3, 1970, pp. 443-453, ISSN: 0022-2836.

[21] Geoffrey J. Barton and Michael J. E. Sternberg “A strategy for the rapid multiple alignment of protein sequences : Confidence levels from tertiary structure comparisons” Journal of Molecular Biology Volume 198, Issue 2, 1987, pp. 327-337, ISSN**:** 0022-2836.

[22] Julie D. Thompson, Desmond G. Higgins, Toby J. Gibson “CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice” Nucleic Acids Research, 1994, vol 22 No 22 , pp. 4673-4680. ISSN**:** 0305-1048 (Print), ISSN: 1362-4962 (Electronic).

[23] Robert C Edgar “MUSCLE: a multiple sequence alignment method with reduced time and space complexity” BMC Bioinformatics 2004, 5:113, ISSN: 1471-2105 (Electronic).

[24] S F Altschul, T L Madden, A A Schäffer, J Zhang, Z Zhang, W Miller, and D J Lipman “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs” Nucleic Acids Research,Volume 25(17); September 1, 1997 ISSN 1362-4962 (Electronic) , ISSN 0305-1048 (Print)

[25] T.F. Smith, M.S. Waterman and W.M. Fitch “Comparative biosequence metrics” Journal of Molecular Evolution Volume 18, Number 1, pp. 38-46 ISSN: 0022-2844 (Paper) , ISSN: 1432-1432 (Online)

[26] Rice,P. Longden,I. and Bleasby,A. “*EMBOSS: The European Molecular Biology Open Software Suite* (2000)” Trends in Genetics 16, (6) pp. 276-277 ISSN: 0168-9525

[27] N Saitou and M Nei “The neighbor-joining method: a new method for reconstructing phylogenetic trees” *Molecular Biology and Evolution* 1987 4: 406-425. Online ISSN 1537-1719, Print ISSN 0737-4038

[28] Rajeev Thakur and William Gropp, "Improving the Performance of Collective Operations in MPICH," Recent Advances in Parallel Virtual Machine and Message Passing Interface 10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings Series: Lecture Notes in Computer Science , Vol. 2840, pp. 257-267 ISBN: 3-540-20149-1

[29] Xinmin Tian, Aart Bik, Milind Girkar,Paul Grey, Hideki Saito, Ernesto Su “Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance” “Intel Technology Journal” Q1, Volume 1, Issue 1, 2002 ISSN: 535-864X

[30] Rosenblatt, Frank, The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review,1958, v65, No. 6, pp. 386-408. ISSN: 0033-295X

[31] Andrea Di Blas, Arun Jagota and Richard Hughey “Optimizing neural networks on

SIMD parallel computers" journal Parallel Computing, 2005,Vol 31, Issue 1, pp. 97-115 ISSN: 0167-8191

[32] A Osareh, M Mirmehdi, B Thomas, and R Markham
Automated identification of diabetic retinal exudates in digital colour images
Br. J. Ophthalmol., Oct 2003; 87: 1220 – 1223. ISSN: 0007-1161 (Print), ISSN: 1468-2079 (Electronic)

[33] Steven W. Smith, "The Scientist and Engeneer's Guide to Digital Signal Processing", California Technical Publishing, pp. 285-296 ISBN: 0-9660176-7-6

[34] Методы компьютерной обработки изображений / Под. ред. В.А. Сойфера, М.: ФИЗМАТЛИТ, 2003, 784с., ISBN 5-9221-0270-2

[35] P. Wapperom, A.N. Beris and M.A. Straka "A new transpose split method for three-dimensional FFTs: performance on an Origin2000 and Alphaserver cluster" Volume 32, Issue 1, Jan 2006, pp 1-13, ISSN: 0167-8191

[36] Булочникова Н.М., Горицкая В.Ю., Сальников А.Н. "Система поддержки сбора и анализа статистики о работе вычислительной системы" "Методы и средства обработки информации" Труды второй всероссийской научной конференции. - Москва, издательский отдел факультета ВМиК МГУ, Стр. 136-141, 2005г. ISBN: 5-89407-230-1.

[37] Булочникова Н.М., Горицкая В.Ю., Сальников А.Н. "Некоторые аспекты тестирования многопроцессорных систем" "Программные системы и инструменты" тематический сборник факультета ВМиК МГУ им. Ломоносова: N5. Москва, издательский отдел факультета ВМиК МГУ, Стр. 73-82., 2005г., ISBN: 5-89407-216-6.

[38] Колпаков Р.В., Сальников А.Н. "Аспекты параллельного программирования при задании программы как графа зависимости по данным на примере написания тестов для системы "PARUS"" "Методы и средства обработки информации" Труды второй всероссийской научной конференции. - Москва, издательский отдел факультета ВМиК МГУ, Стр. 269-275. 2005г., ISBN: 5-89407-230-1.

[39] Eric S. Lander, Lauren M. Linton, Bruce Birren, Chad Nusbaum, Michael C. Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, Roel Funke, Diane ... «Initial sequencing and analysis of the human genome» Nature 409, 860-921 (15 Feb 2001) Human ISSN: 0028-0836, EISSN: 1476-4687

[40] Alexeevski A.V., Lukina E.N., Salnikov A.N., Spirin S.A. "Database of long terminal repeats in human genome: structure and synchronization with main genome archives" // "Proceedings of the fours international conference on bioinformatics of genome regulation and structure", Volume 1. BGRS 2004, Novosibirsk, редакционно-издательский отдел ИциГ СО РАН, стр. 28-29.

[41] Алексеевский А.В., Лукина Е.Н., Спирин С.А., Сальников А.Н. "Структура базы данных длинных терминальных повторов в человеческом геноме и синхронизация данных с основными архивами генной информации" труды Всероссийской научной конференции "Научный сервис в сети Интернет 2004" Издательство Московского университета 2004г. ISBN 5-211-05007-X стр. 219.

[42] Свердлов Е.Д. «Ретровирусные регуляторы экспрессии генов в геноме человека как возможные факторы его эволюции» Биоорганическая Химия, 1999, том 25, N 11, с. 821-827. ISSN: 0132-3423.

[43] Сальников А.Н. "Разработка структуры хранения и организация синхронизации информации по повторяющимся нуклеотидным последовательностям в человеческом геноме." Сборник тезисов

Международной научной конференции студентов, аспирантов и молодых ученых "Ломоносов - 2004". Том 1. 12-15 апреля 2004 г. МГУ им. М.В. Ломоносова, стр. 28

[44] С.М. Абрамов, А.И. Адамович, А.В. Инюхин, А.А. Московский, В.А. Роганов, Ю.В. Шевчук, Е.В. Шевчук. 2004. «Т-система с открытой архитектурой» Proc. Суперкомпьютерные системы и их применение SSA'2004. Труды Международной научной конференции, 26-28 октября 2004 г. Минск, ОИПИ НАН Беларуси Минск, с.18-22

[45] A.Lastovetsky, "Adaptive parallel computing on heterogeneous networks with mpC", Parallel Computing, Elsevier Publisher, 28, 2002, pp. 1369-1407.

[46] Экономико математическая библиотека: «Теория расписаний и вычислительные машины» под редакцией Э. Г. Кофмана Москва изд. «Наука» 1984г. 336стр.

[47] M. Saleh, Z. Othman, and S. Shamala «FCFS Priority-based: An Adaptive Approach in Scheduling Real-Time Network Traffic» Networks and Communication Systems proceeding 527, 2006, ISBN 0-88986-590-6.

[48] Marcia C. Cera, Guilherme P. Pezzi, Elton N. Mathias, Nicolas Maillard, Phillipe O.A. Navaux «Improving the Dynamic Creation of Processes in MPI-2» Lecture Notes in Computer Science LNCS 4192 Recent Advantages in Parallel Virtual Machine and Message Passing Interface, Volume 4192, pp. 247-255, 2006, ISBN-10: 3-540-39110-X ISBN-13: 978-3-540-39110-4.

[49] Josh Aas «Understanding the Linux 2.6.8.1 CPU Scheduler» online publication: (http://citeseer.ist.psu.edu/aas05understanding.html) pp. 38.

[50] O. Sename, D. Simon and D. Robert: "Feedback scheduling for real-time control of systems with communication delays" ETFA'03 9th IEEE International Conference on Emerging Technologies and Factory Automation, Lisbonne. Volume 2, 16-19 Sept. 2003 Page(s):454 - 461 vol.2

[51] Уссермен Ф. Нейрокомпьютерная техника. - М.: Мир, 1992.

[52] S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983.

[53] Костенко В.А., Калашников А.В. Исследование различных модификаций алгоритмов имитации отжига для решения задачи построения многопроцессорных расписаний// Дискретные модели в теории управляющих систем. Труды VII Международной конференции. М.: МАКС Пресс, 2006. - С.179-184.

[54] John H. Holland «Adaptation in Natural and Artificial Systems» April 1992 7 x 9, 228 pp. ISBN-10: 0-262-08213-6, ISBN-13: 978-0-262-08213-6.

[55] E.S.H. Hou, N. Ansari, H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 05, no. 2, pp. 113-120, Feb., 1994. ISSN: 1045-9219

[56] Michelle Moore, "An Accurate and Efficient Parallel Genetic Algorithm to Schedule Tasks on a Cluster," *ipdps*, p. 145a, International Parallel and Distributed Processing Symposium (IPDPS'03), 2003. ISBN: 0-7695-1926-1

[57] Костенко В.А., Смелянский Р.Л., Трекин А.Г. Синтез структур вычислительных систем реального времени с использованием генетических алгоритмов// Программирование, 2000., №5, С.63-72. **(Kostenko V.A., Smeliansky R.L., and Trekin A.G. Synthesizing Structures of Real-Time Computer Systems Using Genetic Algorithms. Programming and Computer Software, Vol. 26, No. 5, 2000, pp. 281-288.)**

[58] Garey, M. R. and Johnson, D. S. 1990 «Computers and Intractability; a Guide to the Theory of Np-Completeness.» W. H. Freeman & Co., pages: 338, ISBN: 0716710455.

[59] David Jackson, Quinn Snell, Mark Clement «Core Algorithms of the Maui Scheduler» Lecture Notes in Computer Science Job Scheduling Strategies for Parallel Processing : 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001. Revised Paper, Volume 2221, pp. 87-102, 2001, ISSN: 0302-9743.

[60] Helen D. Karatza «Perfomance Analysis of Gang Scheduling in a Distributed System under Processors Failuries» «International Journal of Simulation Systems, Science and Technology» Volume 2, Number 1, pp. 14-23, 2001, ISSN: 1473-804x, ISSN: 1473-8031.

[61] IBM LoadLeveler for AIX 5L using and Administrating Version 3 Release 1, pp.381-394, 2001. http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IBMp690/IBM/usr/lpp/LoadL/pdf/lluadmin.pdf

[62] P. Bjorn-Jorgensen, J. Madsen, "Critical path driven cosynthesis for heterogeneous target architectures," *codes*, p. 15, 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE '97), 1997.

[63] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7.

[64] Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, Chao-Chin Wu, "Dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems," *icpads*, pp. 365-374, 12th International Conference on Parallel and Distributed Systems - Volume 1 (ICPADS'06), 2006, ISBN: 0-7695-2612-8.

[65] David L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *Computer*, vol. 23, no. 5, pp. 35-43, May, 1990, ISSN: 0018-9162.

[66] Dali Wang, Eric Carr, Louis J. Gross, Michael W. Berry Toward «Ecosystem Modeling on Computing Grids» Computing in Science and Engineering, volume 7, number 5, 2005, pp. 44-52, ISSN: 1521-9615

[67] P.B. Duffy, P.G. Elgroth «Detection of Anthropogenic Climate Change: A Modeling Study», LLNL, 1999, National Technical Information Service U.S. Department of Commerce, Springfield, VA 22161

[68] S.I. Simdyankin and M. Dzugutov, *Case Study: Computational Physics - The Molecular Dynamics Method* , TRITA-PDC-2003:1, ISSN 1401-2731, ISRN KTH/PDC/R—02/1--SE

[69] Takashi Amisaki, Shin-Ichi Fujiwara, "Grid-Enabled Applications in Molecular Dynamics Simulations Using a Cluster of Dedicated Computers," *saint-w*, p. 616, 2004 Symposium on Applications and the Internet-Workshops (SAINT 2004 Workshops), 2004, ISBN: 0-7695-2068-5

[70] Wai-Sum Lin, Rynson W.H. Lau, Kai Hwang, Xiaola Lin, Paul Y.S. Cheung, "Adaptive Parallel Rendering on Multiprocessors and Workstation Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 3, pp. 241-258, Mar., 2001, ISSN: 1045-9219

[71] Voevodin Vl.V. Use of the V-Ray system for optimization of serial programs for parallel computers. - In Proc. of the 2nd Intern. Workshop SMS TPE'94, Moscow,

1994.

[72] Voevodin V.V., Voevodin Vl.V. V-Ray Technology: a New Approach to the Old Problems. Optimization of the TRFD Perfect Club Benchmark to CRAY Y-MP and CRAY T3D Supercomputers.- Proc. of the High Performance Computing Symposium'95, Phoenix, Arizona, USA, 1995, p.380-385.

[73] Voevodin V.V., Voevodin Vl.V. The V-Ray Technology of Optimizing Programs to Parallel Computers //Proc. of the 1st workshop on numerical analysis and applications, Russe, Bulgary, 24-27 June, 1996.

[74] Steven S. Muchnick «Advanced Compiler Design and Implementation» Morgan Kaufmann Publishers Inc., 1997, ISBN: 1-55860-320-4.

[75] Randy Allen, Ken Kennedy «Optimizing Compilers for Modern Architectures: A Dependence-based Approach» Morgan Kaufmann Publishers Inc., 2001, ISBN: 1-55860-286-0

[76] Воеводин В.В. «Информационная структура алгоритмов» Москва: Изд-во МГУ, 1997. - 139с.

[77] Фуругян М.Г. «Некоторые алгоритмы распределения ресурсов в многопроцессорных системах реального времени» Российская Академия Наук Вычислительный центр сообщения по прикладной математике, Москва 1996г., 21 стр.

[78] Гуз Д.С., Красовский Д.В., Фуругян М.Г. «Эффективные алгоритмы планирования вычислений в многопроцессорных системах реального времени» Российская Академия Наук Вычислительный центр сообщения по прикладной математике, Москва, 2004г., 66 стр.

[79] Гречук Б.В., Фуругян М.Г. «Составление оптимальных расписаний с прерываниями в многопроцессорных системах с неполным графом связей» Российская Академия Наук Вычислительный центр сообщения по прикладной математике, Москва, 2004г., 66 стр.

[80] P.H.A. Sneath, Robert R. Sokal «Numerical Taxonomy» *Nature* 193, 855 - 860 (03 March 1962), ISSN: 0028-0836, EISSN: 1476-4687.

[81] Alexey N. Salnikov «PARUS: A Parallel Programming Framework for Heterogeneous Multiprocessor Systems» Lecture Notes in Computer Science (LNCS 4192) Recent Advantages in Parallel Virtual Machine and Message Passing Interface, Volume 4192, pp. 408-409, 2006, ISBN-10: 3-540-39110-X ISBN-13: 978-3-540-39110-4.

Приложения

(Приложение А) Формат текстового представления граф-программы

Инструментальная система рассчитана на двоякое представление графа алгоритма: в виде класса Graph и в виде текстового файла. Приведём формат текстового файла.

Файл графа алгоритма содержит ключевые слова, которые являются обрамляющими тегами, именами полей графа и их значениями. В файле графа алгоритма также могут встречаться комментарии в формате C++**.** В дальнейшем между '<' и '>' скобками будут содержаться понятия, а всё, что содержится между '{' и '}', может быть повторено несколько раз. Символ ::= означает расшифровку понятия.

Текстовое представление графа алгоритма состоит из 2-х блоков, блока узлов и блока рёбер.

```
<GRAPH_BEGIN>
        header "<имя файла заголовка>"
        root "<имя файла с корневым узлом графа>"
        tail "<имя файла с заключительным кодом>"
        num_nodes <число вершин>
    <nodes_block>
        num_edges <число рёбер>
   <edges_block>
<GRAPH_END>
```

- **header** – ссылка на файл, содержащий все переменные, функции, описания типов данных, которые используются узлами графа алгоритма. Следует избегать пользоваться именами, начинающимися с MPI или **px**. Данные имена зарезервированы системой и MPI.
- **root** – ссылка на файл, в котором находится код, выполняющийся в первую очередь на всех процессорах одновременно и одинаково. Обычно это необходимо для проведения начальной инициализации данных, которые затем будут использоваться всеми узлами на всех процессорах. Имя файла в этом поле указывать не обязательно. Можно указать только *""* вместо, например *"root.cpp"*.
- **tail** – ссылка на файл, содержащий код, который необходимо выполнить после работы параллельной части программы. Обычно содержит операции по освобождению памяти и закрытию всё ещё открытых файлов.
- **num_nodes** содержит число узлов в графе.
- **num_edges** содержит число рёбер в графе.

```
<nodes_block>::=<NODES_BEGIN>
{<node>}
<NODES_END>
```

```
<edges_block>::=<EDGES_BEGIN>
{<edge>}
<EDGES_END>
```

Блок вершин графа задаёт список вершин графа:

<*node*>::=<NODE_BEGIN>
number <*номер вершины*>
type <*тип вершины*>
weight <*вес вершины*>
layer <*уровень вершины в графе*>
num_input_edges <*число рёбер, входящих в вершину*>
edges *({<номер рёбра>})*
num_output_edges <*число исходящих из вершины рёбер*>
edges *({<номер ребра>})*
head *"<имя файла с кодом заголовка вершины>"*
body *"<имя файла с кодом тела вершины>"*
tail *"<имя файла с кодом эпилога вершины>"*
<NODE_END>

- **number** – уникальный номер вершины. Для нумерации вершин допускается использовать только неотрицательные значения.
- **type** – целое значение; используется при визуализации графа.
- **weight** – характеристика вычислительной сложности вершины, объём вычислений, выраженный в условных операциях.
- **layer** – целое число, определяющее слой вершин графа.
- **num_input_edges** – число ребер, входящих в вершину графа.
- **num_output_edges** – число ребер, исходящих из вершины графа.
- **edges** – список номеров рёбер графа; число исходящих из вершины ребер равняется **num_input_edges**; число входящих в вершину ребер – **num_output_edges** соответственно.
- **head** – файл с переменными, доступными для текущей вершины, и кодом, который следует исполнять вершине до момента получения данных от других вершин. Значение может быть пустым.
- **body** – файл с кодом, который исполняется вершиной после получения данных от других вершин.
- **tail** – файл с кодом, который исполняется вершиной после окончания упаковки данных в системные буфера, из которых затем после инициации передачи будет вестись передача данных другой вершине на другой MPI-процесс.

<*edges_block*>::=<EDGES_BEGIN>
{<*edge*>}
<EDGES_END>

Блок описания ребер графа:

<*edge*>::=<EDGE_BEGIN>
number <*номер ребра*>
weight <*вес ребра*>
type <*тип ребра*>
num_var <*число передаваемых фрагментов массивов*>
num_send_nodes <*число вершин, где предполагается посылка*>
send_nodes (<*номер вершины*> *{*<*номер вершины*>*}*)
num_recv_nodes <*число вершин, где предполагается приём*>
recv_nodes (<*номер вершины*> *{*<*номер вершины*>*}*)
<*send_block*>
<*recieve_block*>
<EDGE_END>

- **weight** – целочисленная характеристика объема переданной информации по отношению к эталонной операции передачи сообщений.
- **type** – тип ребра. В настоящий момент поддерживается значение GRAPH_NONE, означающее передачу данных от процессора к процессору.
- **num_var** – целое число, равное числу нарезаемых фрагментов массива для передачи данных по ребру. Все переменные упаковываются в массив и пересылаются вершине одновременно; при получении происходит распаковка массива и присваивание полученных значений переменных.
- **num_send_nodes**, **num_recv_nodes** – число вершин, передающих или принимающих информацию по данному ребру. В данной реализации значение поля равно 1.

<*send_block*>::=<SEND_BEGIN>
{<*chunk*>}
<SEND_END>

<*receive_block*>::=<RECEIVE_BEGIN>
{<*chunk*>}
<RECEIVE_END>

<*chunk*>::= <CHUNK_BEGIN>
name *"<адрес переменной>"*
type <*тип переменной*>
left_offset *"<левая граница; смещение адреса>"*
right_offset *"<правая граница; смещение адреса>"*
<CHUNK_END>

- **name** – имя переменной
- **type** – тип переменной. Не допускается использовать сложные структуры. Тип аналогичен GRAPH_INT, GRAPH_FLOAT и т.п.
- **left_offset, right_offset** – смещения, описывающие сегмент хранения переменной. Информация в этом диапазоне пересылается другой вершине графа. Значения данных полей – целые числа в используемом языке программирования.

(Приложение Б) Формат текстового представления расписания

Текстовое представление расписания устроено аналогично текстовому представлению файла с графом. Нумерация процессоров начинается с 1, процессор с номером 0 зарезервирован для системы времени запуска, осуществляющей планирование назначений вершин графа на MPI-процессы, инициализацию передач данных между вершинами графа и управление хранением данных, передаваемых по рёбрам.

<*schedule*>::= <SCHEDULE_BEGIN>
num_processors <*число MPI-процессов*>
{<processor>}
<SCHEDULE_END>

<*processor*>::=<PROCESSOR_BEGIN>
name <*номер MPI-процесса*>
num_nodes <*число вершин, распределённых на MPI-процесс*>
nodes *({<номер вершины>})*
<PROCESSOR_END>

- **num_processors** – число процессоров, меньше на 1 реального числа процессоров, на которых была запущена задача.
- **name** – имя процессора. Не может быть пустым.
- **num_nodes** – число вершин, которые должны выполнятся на процессоре.

(Приложение В) Параметры построителя расписаний

В ini-файле для построителя расписаний могут быть заданы некоторые параметры алгоритма. Данные параметры сильно влияют на скорость работы программы и точность генетического алгоритма. Проблема подбора параметров алгоритма для каждого конкретного набора входных данных является сложной задачей. Предполагается, что параметры выбираются на основании эмпирических соображений или тестовых данных.

Параметры программы разделены на секции, название секции записывается в квадратных скобках перед определением параметров, принадлежащих этой секции.

[creaturs]	**Секция описания особей**
max=100	Максимальное количество особей в популяции генетического алгоритма
start_part=1	Количество особей в начальной популяции
random_selecting=0.1	Вероятность, с которой особи удаляются из популяции с учётом добавления штрафа.

[stop]	**Секция параметров остановки алгоритма**
max_cycles=1024	Максимальное количество итераций алгоритма (число создаваемых популяций)
min_difference=0	Минимальная разница в значениях лучшей целевой функции, при которых итерация не считается бесполезной
idle_iterations=20	Количество бесполезных итераций до остановки

[mutate_possibilities]	**Секции параметры операции мутации**
avg=0.5	Доля мутирующих особей по отношению к размеру всей популяции
processors=0.6	Вероятность миграции вершины с одного процессора на другой
priority=0.2	Вероятность изменения порядкового номера вершины в расписании для фиксированного процессора
avg_gens=0.2	Доля мутирующих генов особи

[recombination]	Секция параметры операции скрещивания
avg_possibility=0.6	Вероятность скрещевания для пары случайно выбранных особей
avg_points=3	Число точек, в которых происходит скрещивание в хромосоме

(Приложение Г) Описание некоторых архитектур многопроцессорных систем

Описание IBM pSeries 690

IBM eServer pSeries 690 Regatta – машина, установленная на ВМиК МГУ. Она имеет следующую конфигурацию: архитектура ccNUMA, 16 процессоров Power4 с тактовой частотой 1,4Ghz, 64gb оперативной памяти, операционная система AIX5L. Рассмотрим более подробно архитектуру pSeries690. Сервер pSeries 690 Regatta – это сервер, в который можно установить до 32-х процессоров на базе кристалла IBM Power4.

Система состоит из процессорной подсистемы и до восьми корпусов ввода-вывода. Процессорная подсистема состоит из 1-4 многокристальных модулей (MCM), каждый из которых содержит четыре 2-процессорных кристалла Power4, образующих конструктивный блок процессорной подсистемы с 8 CPU. Каждый корпус ввода-вывода содержит 20 слотов ввода-вывода PCI и до16 отсеков для дисковых накопителей.

На одном полупроводниковом кристалле POWER4 размещены два 64-разрядных процессорных ядра PowerPC с 64kb кэш инструкций и 32kb кэш данных (L1), кэш второго уровня (L2 1,5mb) с общим доступом для обоих процессорных ядер, контроллер кэша L3 и контролер взаимодействия кэш L3 с оперативной памятью, контроллер шины периферийных устройств (GX controller). IBM предоставляет технологию многопроцессорной обработки на кристалле (Chip Multi-Processing – CMP), которая фактически реализует на одном кристалле SMP систему с двумя процессорами. Такая технология дает преимущества в достигаемой ширине полосы пропускания при обмене сигналами между отдельными элементами чипа, в плотности упаковки данных при передаче, за счёт уменьшения расстояния между элементами. Как следствие, получается выигрыш в производительности чипа и выигрыш в потребляемой чипом мощности. На рисунке 37 проиллюстрирована архитектура полупроводникового кристалла POWER4.

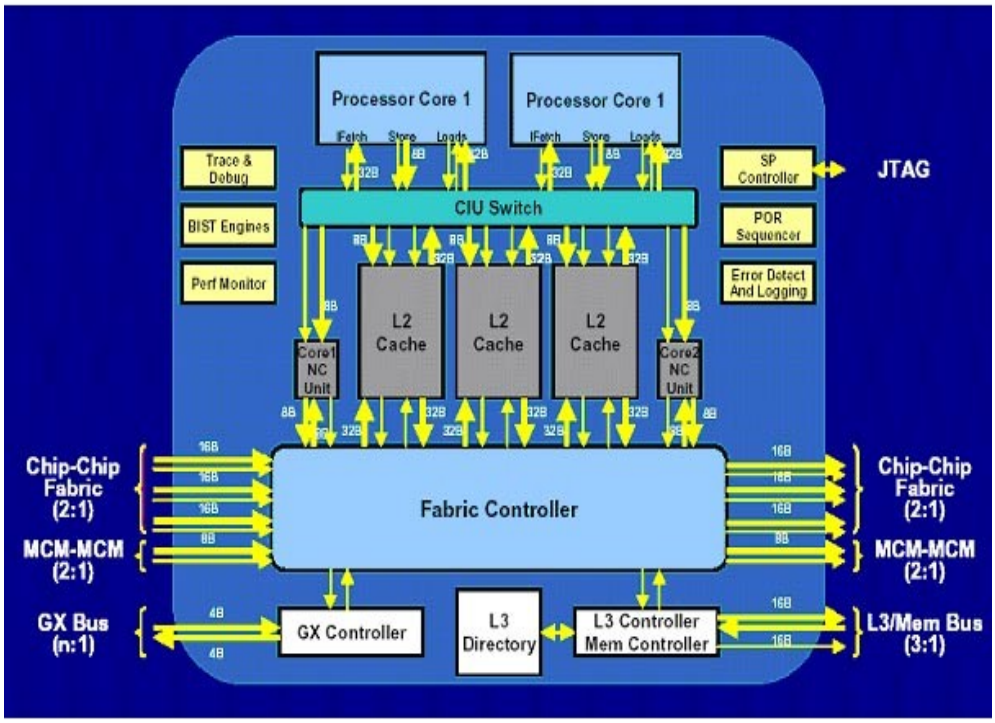

Рисунок 37. архитектура чипа POWER4.

Ключевым аспектом конструкции процессорной подсистемы является упаковка полупроводниковых кристаллов POWER4 на одну керамическую подложку. На многокристальном модуле (MCM) смонтировано четыре процессорных кристалла POWER4, составляющие вместе 8-процессорный конструктивный блок SMP. Каждый модуль реализует полносвязанный граф коммуникаций между чипами POWER4. Это показано на рисунке 38.

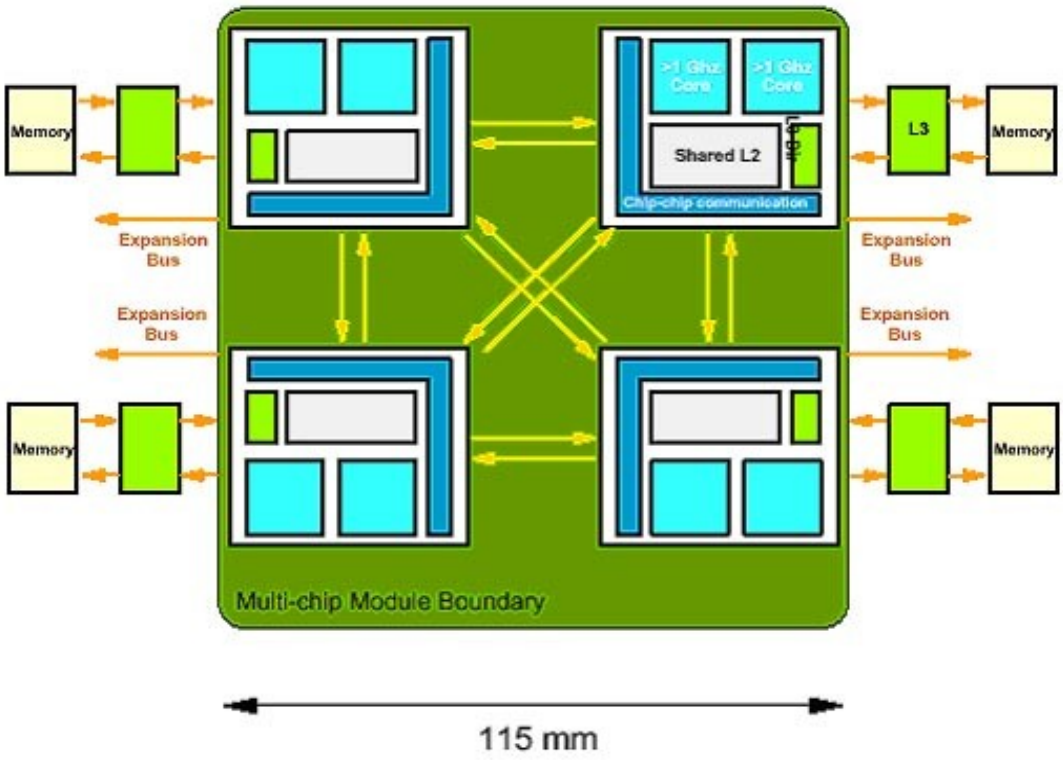

Рисунок 38. MCM Модуль.

В максимальной 32-процессорной конфигурации четыре модуля MCM, каждый из которых насчитывает четыре кристалла POWER4, то есть 8 процессоров, соединены вместе с помощью каналов связи модуль-модуль. Каждый модуль MCM имеет доступ к кэшу L3 объемом до 128 Мбайт. Поддерживается синхронная динамическая оперативная память с коррекцией ошибок ECC SDRAM объемом до 256 Гбайт. Окончательный вариант сборки системы представлен на рисунке 39.

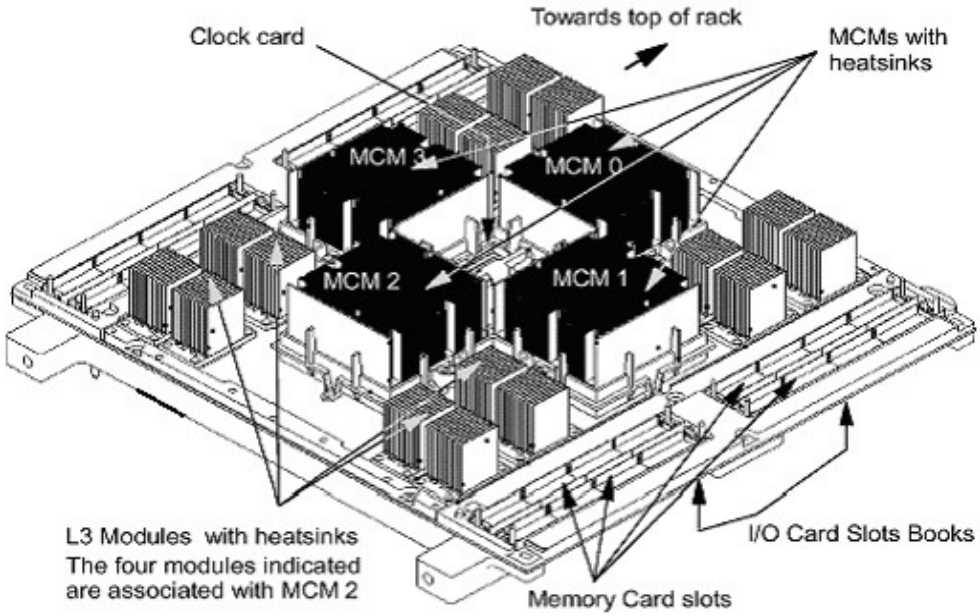

Рисунок 39. Системная плата pSeries 690.

Таким образом, с точки зрения классификации архитектур вычислительных систем по доступу в память pSeries 690 как целое является системой типа ccNUMA с довольно сложной архитектурой. Сложность архитектуры приводит к тому, что при обмене данными между процессорами, возможно, будут наблюдаться эффекты, подобные эффектам в сетях.

Описание МВС-1000М

МВС-1000М многопроцессорная вычислительная система, предназначенная для решения сложных научно-практических задач; установлена в межведомственном суперкомпьютерном центре (МСЦ РАН). Пиковая производительность МВС-1000М составляет 10^{12} операций с плавающей точкой с двойной точностью в секунду. Общий объем оперативной памяти на всю систему целиком - 768 Гбайт. На вычислительной системе доступно 768 процессоров Alpha 21264, разбитое на 6 базовых блоков, состоящих из 64 двухпроцессорных модулей. Модули связаны между собой сетью Myrinet 2000. В комплект также входит файл-сервер NetApp F840. На рисунке 40 представлена общая схема вычислительной системы МВС-1000М.

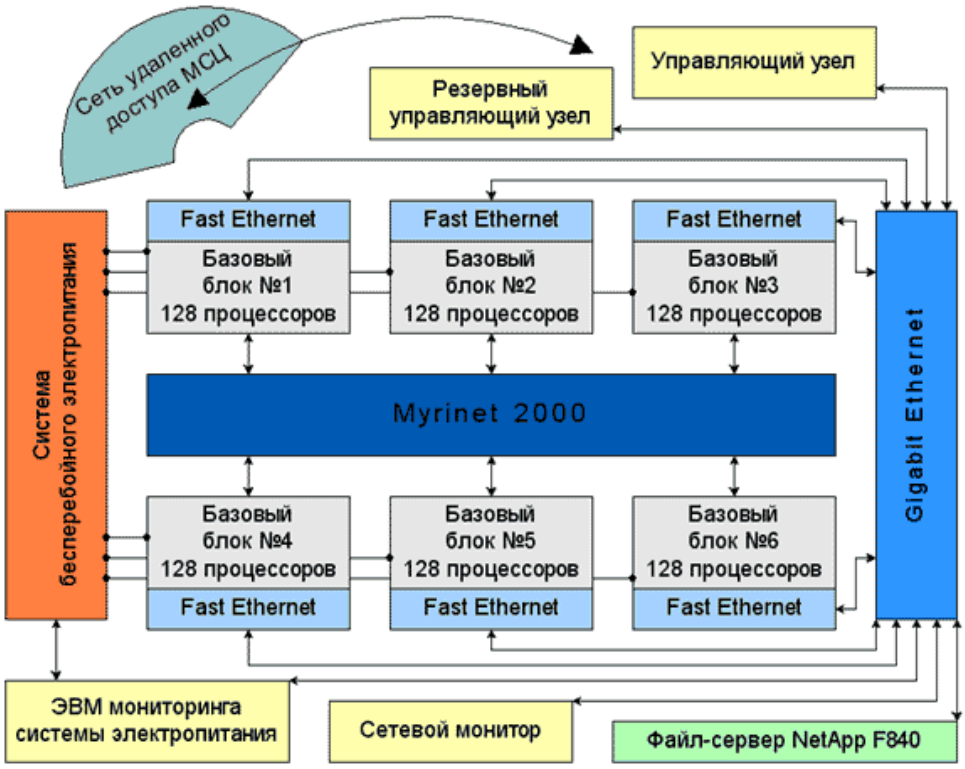

Рисунок 40. Структура МВС-1000М.

Каждый вычислительный модуль включает: 2 процессора Alpha 21264 667 Мгц с кэш-памятью 2-го уровня объемом 4 Мбайта, 2 Гбайта разделяемой оперативной памяти, жесткий диск объемом 20 Гбайт для локального хранения промежуточных результатов счёта. Вычислительные модули связаны между собой сетью Myrinet2000 с пропускной способностью канала 2000 Мбит/сек. Сеть Myrinet2000 в МВС-1000М реализована на базе 6-ти 128-входовых полносвязных коммутаторов. МВС-1000М работает под управлением операционной системы LINUX.

Таким образом, МВС-1000М имеет кластерную архитектуру, где взаимодействие отдельных процессоров в момент решения вычислительной задачи осуществляется через примитивы передачи сообщений. Для этих целей на МВС-1000М установлен пакет MPICH-GM, который реализует стандарт MPI.

Описание Sun Fujitsu PRIMEPOWER 850

Третья вычислительная система на которой проводились вычислительные эксперименты, – Fujitsu Sun PRIMEPOWER 850 сервер (в дальнейшем pp850). Эта машина установлена в Научно-исследовательском институте физико-химической биологии им.А.Н. Белозерского. Она используется в первую очередь для расчётов, связанных с биологией и химией. В pp850 установлено 16 процессоров SPARC64-V 1,89Ghz, кэш первого уровня (L1) 256kb на процессор, 3mb кэш второго уровня на процессор, 128Gb оперативной памяти в максимальной конфигурации.

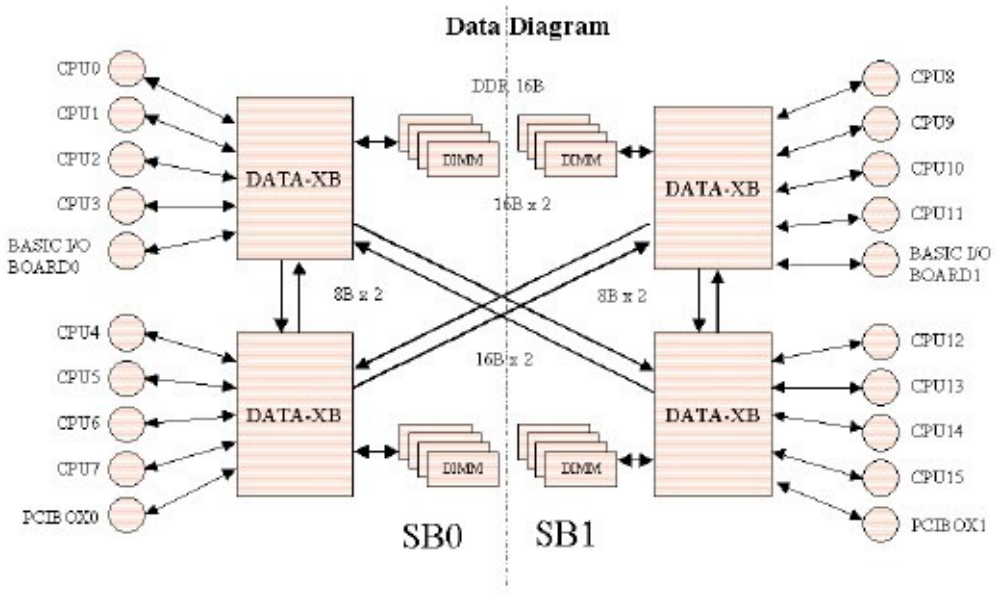

Рисунок 41. Подсистема достума к памяти на PRIMEPOWER 850.

Как показано на рисунке 41, подсистема доступа в оперативную память разделена на 4 блока. Каждый блок соединяет 4 процессора и один контролер периферийных устройств. Каждый блок ответственен за когерентность кэшей второго уровня. Адресное пространство оперативной памяти распределено между блоками таким образом, что сперва приложение обращается к паре блоков, а затем в случае обращения по адресу с большим номером, происходит обращение к остальным 2 блокам, и в этом случае адреса начинают распределяются уже по четырём блокам. Каждый блок управляет 4-мя модулями памяти, и адреса, по аналогии с блоками, также распределены. Блоки соединены между собой каналом связи с частотой работы 540Mhz, что в конечном случае для всей системы целиком позволяет передавать 41,8 Gb в секунду. Вся система в целом по доступу в память является SMP системой.