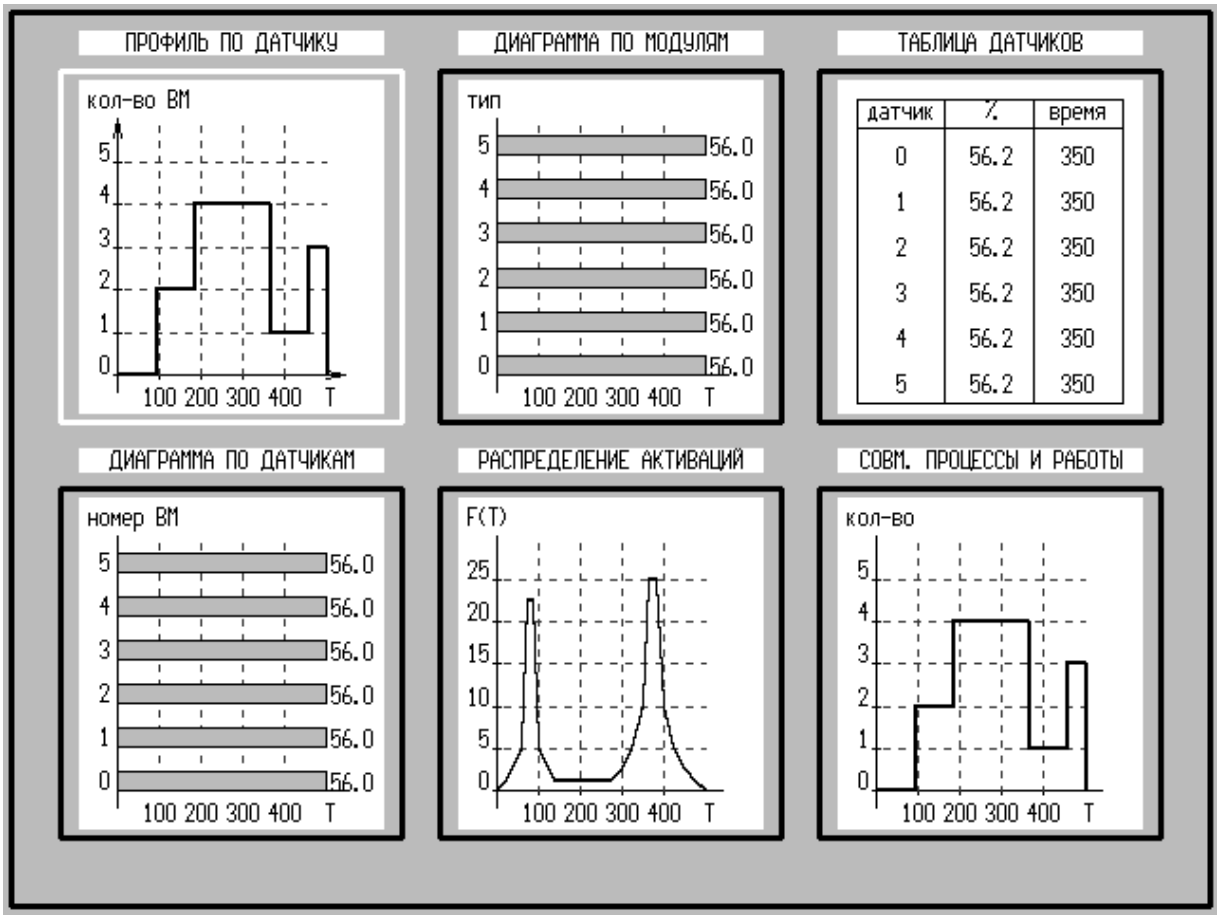
В.В. Васильчиков

Средства параллельного программирования для вычислительных систем с динамической балансировкой загрузки



Министерство образования Российской Федерации
Ярославский государственный университет им. П.Г. Демидова

В.В. Васильчиков

Средства параллельного программирования для вычислительных систем с динамической балансировкой загрузки

Ярославль 2001

УДК 681.3
ББК 3973.2 - 018
В 19

Рецензенты: лаборатория информационно-вычислительных систем и сетей ИМИ РАН; д-р техн. наук, проф. В.А. Курчидис

Печатается при финансовой поддержке федеральной целевой программы “Интеграция” (контракт № А-0068)

Васильчиков В.В. Средства параллельного программирования для вычислительных систем с динамической балансировкой загрузки / Яросл. гос. ун-т. Ярославль, 2001. 127 с.
ISBN 5-8397-0171-8

Рассмотрены основные моменты метода рекурсивно-параллельного программирования, архитектуры соответствующей вычислительной системы. Подробно описаны языковые и программные средства поддержки данного стиля программирования.

Особое внимание уделено методам и средствам повышения эффективности параллельного выполнения программ, а также средствам поиска ошибок. Подробно рассмотрены назначение, структура и средства работы в среде рекурсивно-параллельного программирования RPMSHELL. Приводится ряд примеров программ.

Может быть рекомендовано студентам, изучающим курс "Параллельное программирование", а также выполняющим курсовые и дипломные работы, связанные с разработкой или исследованием параллельных программ.

Ил. 19. Библиогр.: 8 назв.

ISBN 5-8397-0171-8

Введение

Данная работа посвящена описанию концепции рекурсивно-параллельного программирования, а также разработанных к настоящему моменту языковых и программных средств поддержки этого подхода к организации параллельных вычислений.

Подход этот заметно отличается от методов параллельного программирования, наиболее широко распространенных сейчас в мире, в частности, методов, основанных на MPI (message passing interface). Последний базируется на достаточно простых понятиях и предоставляет разработчику удобные средства межпроцессорного взаимодействия. Однако на программиста возлагается вся работа по распределению данных и работы по вычислительной системе, а также ответственность за корректную синхронизацию всех параллельных процессов. Разработка и отладка таких программ (по крайней мере нетривиальных) весьма сложна и трудоемка, и даже доказательство того, что вычислительный процесс не окажется в состоянии бесконечного ожидания, представляет собой серьезную математическую проблему.

Концепция рекурсивно-параллельного программирования основана на очень простой, даже примитивной, схеме синхронизации параллельных процессов, возможно несколько понижающей гибкость программирования. Но это позволяет не только существенно упростить процесс разработки и отладки параллельной программы, но и, что еще более важно, придать программе независимость от состава и быстродействия модулей, образующих вычислительную систему, а также возможность эффективного выявления и использования параллелизма, зависящего от обрабатываемых данных. Собственно, достижение этих качеств и было основной целью при разработке и реализации упомянутой концепции.

Данная работа может использоваться и в учебном процессе при изучении рекурсивно-параллельного программирования, а также оказаться полезной студентам, изучающим курс "Параллельное программирование" или выполняющим курсовые и дипломные работы, связанные с разработкой и исследованием параллельных программ.

Рекурсия как средство организации параллельных вычислений

Существующие подходы. Цели проекта

Целью проекта, положенного в основу данной публикации, являлась разработка стиля и языка программирования, а также методов и программных средств их поддержки, предназначенных для создания, отладки и оптимизации параллельных прикладных программ,

инвариантных к конфигурации вычислительной системы, на которой они выполняются, и в то же время достаточно эффективных. Такое свойство является весьма важным для создания переносимых библиотек и пакетов прикладных программ, предназначенных для использования на параллельных вычислительных системах, как в супер-ЭВМ [1, 2], так и в системах распределенных вычислений [3].

Для того, чтобы добиться эффективной параллельной работы программы, следует, во-первых, максимально сократить затраты времени на передачу данных между процессорными модулями (ПМ), во-вторых, распределить работу таким образом, чтобы по возможности более ровно загрузить вычислительные ресурсы системы. Последняя задача усложняется еще и тем обстоятельством, что ход вычислений и длительность выполнения тех или иных параллельных ветвей программы, вообще говоря, зависит от обрабатываемых данных. Зачастую это обстоятельство делает невозможным при написании программы или в процессе трансляции осуществить эффективное распределение работы.

В ряде случаев данный недостаток приводит к выполнению избыточных вычислений и, соответственно, увеличению времени решения задач. К примеру, в сеточных задачах (методы конечных разностей или конечных элементов) использование нерегулярных сеток для некоторых применений позволяет повысить точность решения и устранить избыточные вычисления. Использование нерегулярных сеток в этих случаях диктуется как пространственной, так и временной неоднородностью реальных процессов. В качестве другого примера можно привести задачу численного интегрирования методом адаптивной квадратуры, для которой статическое распараллеливание также не очень эффективно.

Динамическая балансировка, то есть оперативное перераспределение работы между процессорными модулями во время выполнения программы, позволяет добиться более равномерной загрузки вычислительных ресурсов системы и, следовательно, большей эффективности. Потребность в динамической балансировке возникает в тех случаях, когда невозможно сделать оптимальное статическое отображение процессов на процессорные модули из-за отсутствия достаточной информации об объеме вычислений на параллельных ветвях, либо из-за того, что каждому шагу решения задачи соответствует свое оптимальное отображение. За примером можно обратиться к тем же сеточным задачам, когда времена вычислений в узлах используемой сетки априори неизвестны и могут изменяться от одного временного шага к другому.

Другим аргументом в пользу динамической балансировки является то, что она делает возможной разработку прикладных программ, инвариантных к конфигурации параллельной вычислительной системы. Это существенно упрощает процесс программирования и позволяет создавать легко переносимые библиотеки и пакеты прикладных программ.

Задача прикладного программиста при этом фактически сводится к декомпозиции задачи на достаточное количество параллельных процессов. При этом, чем больше их количество, тем более равномерной загрузки мы можем добиться. Опыт показывает, что хорошие результаты достигаются, если их количество параллельных процессов превышает количество процессорных модулей на порядок, даже если трудоемкость параллельных ветвей программы существенно различается.

Однако при таких условиях организация в системе централизованного распределения работы и единой очереди готовых к выполнению процессов (типа "Пул задач" [3]) может привести к появлению "узкого места" и, следовательно, к существенному падению эффективности. Кроме того, в этом случае оценка времени заполнения всех процессорных модулей работой при условии, что в начальный момент работу имел только один, будет линейной.

В основу данной разработки был положен известный принцип рекурсивного порождения параллельных процессов, идея которого весьма проста и, можно сказать, лежит на поверхности. Однако в нашем случае этот метод используется не только как средство порождения параллельных ветвей задачи, но и как важный элемент механизма динамической балансировки загрузки, позволяя существенно уменьшить количество передач работы и результатов. Все это позволяет в значительной степени снизить влияние упомянутых факторов и разрабатывать достаточно эффективные параллельные прикладные программы, обладающие свойством инвариантности к конкретной конфигурации вычислительной системы.

Разработанный алгоритм динамической балансировки носит децентрализованный характер, то есть решение о запросе или передаче работы каждый процессорный модуль принимает самостоятельно на основе имеющейся в его распоряжении информации, может быть и устаревшей. Однако централизованный характер управления неизбежно накладывает серьезные ограничения на возможность наращивания системы. Оценка скорости заполнения всей системы работой становится логарифмической.

Отметим также, что предлагаемый язык программирования и средства поддержки отнюдь не исключают и не рекурсивный способ порождения параллельных ветвей программы, однако в этом случае временные затраты на распределение работы по системе будут, по-видимому, выше. Точно так же предусмотрена возможность явного назначения программистом порождаемых параллельных процессов на выполнение конкретным ПМ. В принципе, зная конфигурацию имеющейся вычислительной системы, в этом случае можно добиться даже некоторого повышения эффективности работы программы, однако в случае любого изменения конфигурации системы производительность ее может существенно снизиться.

Реализация вычислительного модуля на основе рекурсивно-параллельного метода программирования и предлагаемых механизмов порождения и распределения параллельных вычислительных процессов дает возможность создавать модульно-наращиваемые параллельные ВС различной конфигурации вплоть до супер-ЭВМ, обеспечивающие высокую эффективность функционирования на широком классе задач. Вместе с тем, те же самые методы и механизмы, реализованные программно, вполне пригодны для эффективной организации распределенных вычислений на локальной сети компьютеров. Такая возможность, в частности, реализована в описанной ниже пользовательской среде рекурсивно-параллельного программирования RPMSHELL. Образующее ее программное обеспечение позволяет произвести полный цикл создания такой рекурсивно-параллельной программы, включая ее последовательную и параллельную отладку, а также исследование ее поведения путем имитационного моделирования и оптимизацию.

Дальнейшее содержание посвящено описанию применяемой модели параллельных вычислений, основных механизмов порождения и активизации параллельных процессов, краткой характеристики используемого языка рекурсивно-параллельного (РП-) программирования, а также программных инструментальных средств поддержки данного стиля построения алгоритмов и программ.

Метод рекурсивно-параллельного программирования

Как следует из сказанного выше, рекурсивно-параллельный стиль программирования является одним из перспективных подходов к организации параллельного вычислительного процесса. Основным его достоинством является возможность использования потенциального параллелизма алгоритма, зависящего от исходных данных. Другое важнейшее достоинство – возможность обеспечения эффективной динамической балансировки загрузки процессорных модулей (ПМ) во время выполнения программы. При этом прикладной программист не должен ничего знать о количестве и быстродействии ПМ, входящих в состав вычислительной системы. Единственное требование, которому должна удовлетворять программа, заключается в том, что в кратчайшее время работа должна быть разбита на достаточное количество независимых фрагментов по возможности одинакового объема и соответствующим образом оформленных.

Для довольно широкого класса задач разбиение работы на два равных (или почти равных) фрагмента не составляет труда. Например, для нахождения суммы n чисел, являющихся результатом некоторой функции F(i), $1 \le i \le n$, можно независимо вычислить сумму первых n/2 слагаемых (здесь и далее деление целочисленное) и сумму оставшихся n/2 слагаемых. Решением будет сумма двух частичных результатов. Несложно видеть, что каждая из двух "половинок" работ может быть разбита на две

"четвертушки" и т.д. Таким образом, если в системе имеется N процессорных модулей, то теоретически уже за время порядка $\log_2 N$ мы можем их всех обеспечить работой, причем с высокой степенью равномерности.

Как оформить предложенный алгоритм в виде программы? Очень удобным и естественным оказывается оформление его как рекурсивной функции (на языке C):

```
float Sum(int i, int j)
{ if (i==j)
    return( F(i) );
  else
    return( Sum(i,(i+j)/2) + Sum((i+j)/2+1,j) );
}
```

Здесь параметры i, j – начальный и конечный индексы суммирования. Договоримся называть активацией функции (или процедуры) ее запуск с конкретными значениями параметров.

Результатом активации функции Sum(i, j) является, $\sum_{k=i}^{j} F(k)$, т.е. одна из частичных сумм. Чтобы получить сумму всех элементов, достаточно применить оператор вызова Sum(1, n).

Разумеется, для того, чтобы данная функция могла выполниться параллельным образом, необходимо изменить механизм вызова функций таким образом, чтобы в процессе выполнения программы соответствующая команда не приводила к передаче управления в тело функции, а просто порождала бы соответствующий параллельный (потенциально мигрирующий) процесс, а программа продолжала бы выполняться, не дожидаясь его выполнения. Аппаратура и системное программное обеспечение рекурсивно-параллельного мультипроцессора (RPM) должны обеспечивать передачу потенциально мигрирующих процессов между ПМ и возвращение результатов.

Кроме того, очевидно, что появляется необходимость принципиально новых операторов языка программирования, обеспечивающих потребности именно параллельной организации вычислительного процесса. Как минимум, это операторы описания и запуска параллельных функций, а также операторы синхронизации параллельных вычислительных процессов. Так, в нашем примере мы должны дождаться завершения двух рекурсивных вызовов функции Sum(), прежде чем произведем сложение полученных результатов.

С целью удовлетворения перечисленным требованиям на основе языка C был разработан язык программирования, ориентированный на применение рекурсивно-параллельного стиля организации вычислительного процесса. Его описание приводится в следующей главе, а мы просто приведем пример описания на этом языке рекурсивно-параллельной процедуры Sum().

```
struct BP_Sum       /*   ••••••••• •••••   */
       { int i, j;  /*   ••••••••••        */
         float s;   /*   ••••••••• Sum     */
       };
Parallel(Sum,bp)        /*    ••••••••• ••••••••••••   */
PARAM (BP_Sum,*bp);     /*        ••••••••• Sum        */
{  NEW_PARAM (BP_Sum,bp1);/* •••••••• •••••• ••••••••••• ••• */
   NEW_PARAM (BP_Sum,bp2);/*  •••••••••••• ••••••• Sum   */
   int m;
   if ((bp->i)==(bp->j)) /* ••••••• •••••••••• "•••••••••" */
      bp->s=F(bp->i);    /*    •••• •••••• •••••••••    */
   else
   {  /* •••••••••• "•••••••••" */
      bp1.i=bp->i; bp1.k=bp->k;
      bp1.j=(bp->i + bp->j)/2; /*  ••••••••• ••••••••• */
      bp2.i=bp1.j+1;           /*  •••••••••• •••••••    */
      bp2.j=bp->j; bp2.k=bp->k;
      P_Call(Sum,&bp1);        /*  ••••••••• 1 ••••••••  */
      P_Call(Sum,&bp2);        /*  ••••••••• 2 ••••••••  */
      Wait();                  /* •••••••••••• ••••••••• */
      bp->s=bp1.s+bp2.s;       /* •••••••• ••• ••••••••   */
   }
} /* Sum */
```

Здесь рекурсивная "раскрутка" происходит до тех пор, пока это вообще возможно. При этом порождается n "листьевых" активаций, после чего начинается процесс "обратного хода" рекурсии, в котором "дочерние" активации процедуры Sum() возвращают свои результаты "родительским" активациям. Это продолжается, пока не будет получен конечный результат. В дальнейшем мы увидим, что такое мелкое деление работы неразумно, но пока на этом не будем останавливаться подробнее.

Такой стиль программирования будем называть РП-программированием. РП-программа представляет собой множество процедур, допускающих рекурсивный вызов. Вызов РП-процедуры с конкретными значениями аргументов мы будем называть активацией. Возврат из "дочерней" активации осуществляется в специальную точку синхронизации "родительской" процедуры, а не в точку, непосредственно следующую за вызовом процедуры, как это имеет место в обычных последовательных программах.

Параллельный алгоритм, построенный таким образом, во-первых, инвариантен к количеству и быстродействию вычислительных модулей в системе, во-вторых, позволяет выявить параллелизм вычислительного процесса, зависящий от обрабатываемых данных, и, следовательно, проявляющийся только в ходе решения задачи. При этом скорость размножения параллельных процессов и распространения их по системе при грамотной организации может иметь экспоненциальный характер.

Описанные ниже механизмы позволяют произвести действия по распределению и перераспределению работы в системе на фоне основных вычислений, а также минимизировать количество передач, необходимых для равномерного распределения работы в системе. Последнее свойство заметно снижает накладные расходы на организацию динамического распараллеливания и позволяет добиться практически линейного характера ускорения с ростом числа ПМ.

Структура рекурсивно-параллельного вычислительного процесса и ее представление

Использованный выше в качестве примера рекурсивный алгоритм весьма типичен. Он описывает три этапа вычислений:

- рекурсивное деление интервала [1, N] пополам до получения интервала (рекурсивная раскрутка дерева активаций) заданной длины – в данном случае единичной;
- вычисление частичной суммы значений функции F(k) (вычисления на листе дерева активаций) – в данном случае это всего одно значение;
- вычисление частичных сумм на обратном ходе рекурсии вплоть до получения искомого результата (редукционные вычисления).

Структура порождаемого параллельного вычислительного процесса наглядно может быть представлена, например, деревом активаций параллельных процедур. На рис.1 изображено дерево активаций приведенной выше программы в случае, когда весь вычислительный процесс распадается на 8 независимых кусков. Здесь цифрой 1 обозначены действия по порождению дочерних активаций (прямой ход рекурсии), цифрой 2 действия после завершения дочерних активаций (обратный ход), цифрой 3 вычисления значений F(l) (листьевые активации).

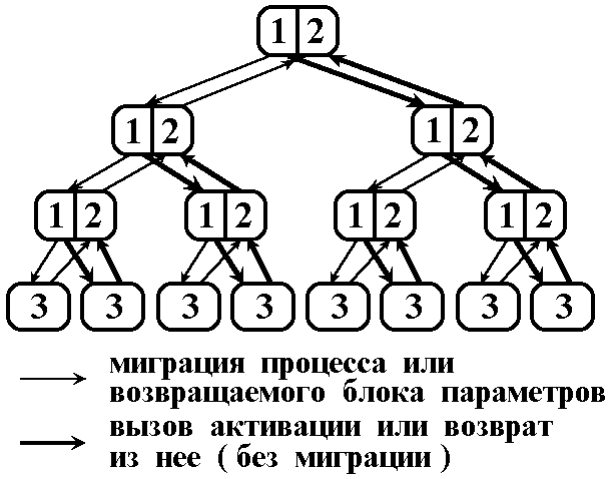


Рис. 1. Типичная структура рекурсивно-параллельного вычислительного процесса.

Развертку вычислительного процесса во времени, представленную как информационный граф, мы будем называть графом трассы вычислительного процесса. В графе трассы вершины соответствуют процессам, а дуги –информационным зависимостям между ними. Кроме того, каждая вершина содержит в себе необходимую информацию о соответствующем процессе.

Представление рекурсивно-параллельного вычислительного процесса в виде графа трассы не зависит от архитектуры вычислительной системы, но вполне представляет структуру параллельного процесса, порожденного программой при обработке конкретных входных данных. Это очень удобно для имитационного моделирования параллельного исполнения соответствующей программы на параллельной вычислительной системе заданной конфигурации.

Программные средства для проведения такого моделирования имеются в описанной ниже среде рекурсивно-параллельного программирования RPMSHELL и являются одним из основных инструментов исследования качества и свойств РП-программы. Итак, перечислим основные достоинства представления структуры параллельного вычислительного процесса в виде графа трассы:

- представление в виде графа трассы полностью отражает потенциальный параллелизм алгоритма решения задачи с конкретными исходными данными;
- представление инвариантно к конфигурации параллельной вычислительной системы, на которой решается задача, и, следовательно, можно сравнить эффективность параллельного исполнения программы на различных конфигурациях ВС, построив всего один граф трассы;
- граф трассы может быть построен в процессе последовательного выполнения пользовательской программы.

Иерархическая модель параллельных вычислений.

РП-программа представляет собой иерархическое множество процедур двух типов (параллельные и последовательные), допускающих рекурсивный вызов.

Вычислительный процесс, протекающий в вычислительной системе, поддерживающей РП-стиль программирования, является иерархическим параллельным процессом. Компоненты этого процесса есть активации параллельных процедур, а также некоторые системные функции. Любая компонента, соответствующая активации процедуры, в свою очередь может быть иерархическим параллельным процессом. В качестве основных типов компонент параллельного процесса можно выделить:

- параллельный вызов параллельной процедуры,
- последовательный вызов параллельной процедуры,

- параллельные операторы доступа в разделяемую общую, а также в статическую память,
- параллельные операторы ввода/вывода (в настоящей версии языка не предусмотрены),
- оператор синхронизации,
- некоторые другие параллельные операторы.

Активация процедуры является параллельной, если ее вызов осуществляется посредством специального оператора порождения параллельного процесса. В этом случае активация параллельной процедуры может быть выполнена на любом свободном ПМ системы. После вызова параллельной процедуры вычисления в родительской процедуре продолжаются без приостановки до точки синхронизации. Возврат из дочерней параллельной процедуры в родительскую процедуру осуществляется в точку синхронизации. Таким образом, вычисления в дочерних процедурах и родительской процедуре могут выполняться параллельно.

В точке синхронизации происходит ожидание завершения всех дочерних параллельных процессов, запущенных до момента попадания в данную точку родительской процедуры.

Иерархическая модель параллельных вычислений регламентирует взаимодействие процессов (активаций процедур) следующим образом:

- каждая активация имеет связь по управлению только с родительской и дочерними активациями, при этом активация может завершить свое выполнение только в том случае, если завершили выполнение все ее дочерние активации;
- родительская активация при порождении дочерней активации может передать ей в качестве параметров исходные данные, а после завершения дочерней активации получить от нее возвращаемые параметры, эта передача осуществляется на уровне локальной памяти процессоров без использования разделяемой памяти;
- любой другой обмен данными между активациями процедур может быть организован через специальные операторы доступа к разделяемой или статической памяти.

Подытожим основные достоинства рекурсивной формы представления параллелизма. Рекурсия описывает динамически порождаемые параллельные процессы в виде активаций процедур, что делает возможным:

- разрабатывать параллельные программы, инвариантные к конфигурации аппаратуры, что особенно важно при создании отказоустойчивых систем и написании широко используемых прикладных пакетов и библиотек стандартных параллельных программ;

- представлять параллелизм, зависящий от значений обрабатываемых данных, т.е. тот параллелизм, который можно выявить только в динамике вычислений;
- осуществлять эффективную динамическую балансировку загрузки ПМ системы путем миграции параллельных процессов.

Сам процесс динамической балансировки скрыт от программиста и осуществляется системными средствами. Основными отличиями ее в рекурсивно-параллельной вычислительной системе от обычного динамического назначения процессов на обработку являются следующие два.

Во-первых, отличительной особенностью применяемого алгоритма балансировки является то, что он – децентрализованный (то есть не существует общей для всей системы очереди работ). Это позволяет ликвидировать одно из потенциальных "узких мест" в системе. Во-вторых, ориентация на рекурсивный способ порождения активаций параллельных процедур (а именно активация процедуры является передаваемой единицей работы) в сочетании с описанным ниже механизмом передачи позволяет организовать более эффективное начальное распределение работ. Передается по коммутационной сети не каждая порождаемая активация параллельной процедуры, а лишь только порожденные первыми достаточно емкие активации.

При этом сам процесс порождения работы организован параллельным образом, а передача ее может осуществляться на фоне вычислений. Для заполнения системы с N ПМ требуется N-1 передача, и время порядка $O(\log_2 N)$. Процесс первоначального распределения завершается, как только у всех ПМ есть работа. Дальнейшее порождение активаций в ходе рекурсивной развертки алгоритма уже не приводит к передаче работы в другой ПМ и, следовательно, к существенному росту накладных расходов. Последующие миграции параллельных процессов возможны только в случае, когда какой-либо ПМ не имеет готовых к выполнению активаций. При этом благодаря описанному ниже способу хранения и передачи потенциально мигрирующих процессов, ПМ-работодатель отдает всегда один, самый емкий из имеющихся, что позволяет минимизировать количество необходимых передач работы и возврата результатов, который происходит в обратном порядке. Наконец, если процесс слияния результатов дочерних активаций достаточно трудоемкий, то он также легко может быть организован параллельным образом.

Как видно из сказанного, рекурсивная организация программы и средства динамической балансировки обеспечивают высокую скорость распространения параллельных процессов среди ПМ системы. Механизм распространения при этом подобен цепной реакции. Время миграции практически не зависит от объема входных данных, обрабатываемых процессом, т.к. основной их объем обычно размещен в разделяемой либо статической памяти. Мигрирует только дескриптор процесса и блок

параметров. Благодаря этому свойству вычислительная система может содержать очень большое число ПМ без ущерба динамике миграции параллельных процессов. Кроме того, в "развернутом" виде (с выделенным стеком и блоками локальных данных) одновременно существует сравнительно небольшое количество всех процессов (примерно логарифм двоичный от их общего числа), что существенно снижает требования к необходимой оперативной памяти.

Язык рекурсивно-параллельного программирования

Рекурсивно-параллельный язык С (сокращенно RPC) есть подмножество стандартного языка С, расширенное специальными системными вызовами (операторами). Выражение "рекурсивно-параллельный", используемое в названии языка, обусловлено соответствующим стилем программирования, на который ориентирован данный язык. Он удовлетворяет следующим требованиям:

- является языком параллельного программирования, ориентированным на архитектуру виртуальной мультипроцессорной ВС с динамическим распараллеливанием (RPM);
- параллельная программа на языке RPC может быть оттранслирована стандартным компилятором С как в параллельный исполнимый код, выполняемый на RPM, так и последовательный исполнимый код, выполняемый на обычной последовательной ЭВМ. Данное свойство дает дополнительные возможности при разработке, отладке и использованию параллельных программ на языке RPC;
- является средством для исследования параллелизма программ (или их моделей), а также эффективности функционирования RPM при выполнении данных программ (или их моделей).

Язык RPC, как подмножество стандартного языка С, получается в результате введения некоторых ограничений. В основном они обусловлены специальными соглашениями по передаче параметров между параллельными процедурами, а также необходимостью учета в программе особенностей организации системы памяти.

Переменные, которые в языке С являлись бы глобальными, здесь являются таковыми же, однако нужно иметь в виду, что область их действия ограничена пределами вычислительного модуля, на каждом из которых заводится своя копия таких переменных. Эти переменные, так же как и автоматические, располагаются в локальной памяти (ЛП) процессорных модулей. Кроме того, часть данных может быть размещена в разделяемой оперативной памяти (РОП) системы. Размещение и доступ к таким переменным осуществляется специальным образом.

Основные соглашения языка RPC.

Перечислим, во-первых, ограничения, накладываемые на язык C:

- Параллельными могут быть только процедуры, но не функции. Впрочем, данное ограничение никоим образом не ограничивает использование обычных последовательных функций, которые могут быть описаны и активизированы обычным для C образом. От них требуется лишь отсутствие в теле функции операторов, специфичных для RPC. Если без таковых не обойтись, то функцию следует оформить как параллельную процедуру со всеми вытекающими отсюда последствиями.
- Передача параметров в параллельную процедуру осуществляется специальным образом (описание см. ниже).
- В список зарезервированных имен, которые нельзя определять в пользовательских программах, дополнительно включены все встречающиеся ниже операторы языка RPC, а также все имена, начинающиеся с символов "v_".

Кроме того, разумеется, существуют некоторые правила, без которых невозможно корректное выполнение операторов RPC:

- В программу до первого использования упомянутых операторов должен быть подключен заголовочный файл Seq.h:

```
#include "seq.h"
```

Кавычки должны быть именно такими, поскольку при работе в среде RPMSHELL последняя при необходимости скопирует данный файл в рабочую директорию.

- Файлы, содержащие операторы RPC, должны иметь расширение .rpc, при этом не допускается включение таких файлов в программу посредством директивы #include. Последнее ограничение необходимо для корректной работы препроцессоров, обрабатывающих RPC-код во всех режимах, кроме последовательного, и впоследствии, возможно, будет снято.
- Все параллельные процессы, порожденные в активации параллельной процедуры, должны быть синхронизированы хотя бы одним оператором Wait() до момента завершения этой активации. Параллельными являются операторы вызова параллельной процедуры (кроме H_Call), операторы доступа в РОП, и некоторые другие, использующие общие ресурсы системы. Подробнее об этом сказано ниже.
- Значения, возвращаемые из дочерней активации в качестве результата через блок параметров, до синхронизирующего данный вызов оператора Wait() являются, вообще говоря, неопределенными.
- Передача дочерней активации параллельной процедуры в качестве параметров указателей чревата серьезными ошибками в параллельном

режиме работы, поскольку при выполнении ее на другом ПМ адрес становится бессмысленным. По этой причине следует избегать включения указателей в блок параметров. В данной версии среды RPMSHELL эта ситуация не отслеживается и при работе в последовательных режимах никак не проявляется.

Классы памяти языка RPC и их отображение на архитектуру RPM

При проработке вопросов организации памяти основное внимание было сконцентрировано на решении следующих задач:

- снижение нагрузки на коммутационную сеть;
- обеспечение равномерной нагрузки на коммутационную сеть с целью предотвращения эффекта "горячих точек", приводящего к деградации пропускной способности сети;
- возможность организации параллельного доступа к распределенной разделяемой памяти на фоне основных вычислений.

Каждый процессорный модуль имеет собственную локальную память (ЛП), доступ к которой могут иметь только активации, выполняющиеся на данном ПМ. Ее основное назначение – хранение информации, используемой при вычислениях в пределах конкретной активации, либо в пределах данного ПМ. В соответствии с этим ЛП разделяется на два класса: локальные данные активаций и статические данные, которые используются совместно всеми активациями параллельных процедур, выполняющимися на данном ПМ. При этом конфликты при использовании последних исключены, поскольку на одном ПМ одновременно не могут выполняться несколько активаций.

Автоматические (локальные) переменные параллельной процедуры размещаются в ЛП процессорного модуля, где активизирована данная процедура. Память под локальные данные процессов (активаций процедур) выделяется на стеке, который создается в момент активизации процесса и уничтожается по завершении активации.

В ЛП также размещаются локальные данные процедур и функций, описанные как статические. На эти данные распространяются обычные правила языка С, определяющие область их видимости, порядок размещения и инициализации. Кроме того, следует иметь в виду, что, поскольку при параллельной работе на каждом из ПМ запускается своя копия программы, на нем имеется и своя копия статических данных. То же самое ограничение действует на переменные, описанные как глобальные, – в каждом ПМ заводится своя копия, и эти копии не имеют непосредственной связи друг с другом. Мы будем в дальнейшем говорить о них как о статических данных, размещенных на данном ПМ.

Следует обратить особое внимание на следующий момент. Статические данные доступны любому процессу, выполняемому на ПМ, где они размещены, и, следовательно, существует возможность их модификации другими активациями параллельных процедур в случае приостановки на операторе Wait() выполняемого в данный момент процесса. Поэтому, если программа пишется так, что доступ к статическим данным по записи имеют несколько параллельных активаций, то нельзя быть уверенным, что записанная туда информация сохранилась после срабатывания оператора синхронизации Wait().

ЛП ПМ является "близкой" с точки зрения времени доступа к ней из центрального процессора модуля. При написании программы рекомендуется в максимальной степени использовать статические и автоматические переменные. Непосредственный доступ в ЛП другого ПМ невозможен. Наличие ЛП позволяет основную нагрузку по доступу к обрабатываемым данным локализовать внутри ПМ без обращения к коммутационной сети.

При необходимости программист может разместить данные, совместно используемые всеми активациями процедур, в разделяемой общей памяти (РОП) системы. РОП, вообще говоря, "размазана" по процессорным модулям с некоторым коэффициентом вертикального расслоения. Параметры ее размещения задаются при вызове оператора G_Alloc(). ОП системы является "удаленной", и непосредственный доступ к ней невозможен. Поэтому, для того чтобы доступ в ОП был эффективным, в RPC использованы блоковые операции доступа к ОП, реализованные через специальные системные вызовы (см. ниже). Они дают возможность программисту минимизировать число обращений к разделяемой памяти путем локализации разделяемых данных при их многократном использовании внутри активации процедуры. Блоковая операция доступа оформляется в виде независимого процесса и может выполняться параллельно с активацией процедуры, породившей ее. Это позволяет осуществлять доступ в разделяемую память на фоне основных вычислений, следовательно, время доступа становится менее критичным.

Заметим также, что разделяемые данные векторного типа могут обрабатываться параллельно, при этом конфликта по записи в каждый отдельный элемент вектора не произойдет, если вычисление его значения будет выполнять единственный процесс (активация процедуры). Если программист будет придерживаться данного правила и использовать разделяемую память по назначению, то при модификации ячеек разделяемой памяти отпадет необходимость в использовании специальных синхронизирующих примитивов, в частности, "замков".

Разделяемая память предназначается только для хранения общих данных и, может быть, реентерабельного программного кода. При этом не требуется физическое наличие в системе общей памяти – она может просто

эмулироваться, что, в частности, и реализовано в библиотеках, входящих в среду RPMSHELL[5].

Класс регистровых переменных можно использовать аналогично их обычному применению в языке С.

Системные вызовы параллельных процедур

Все процедуры, которые будут вызываться с помощью описанных ниже системных вызовов, должны быть описаны в начале того же файла как

```
CPP(pname);
```

где pname – имя параллельной процедуры. Вызовы P_Call() и P_Send() порождают параллельный процесс и требуют синхронизации посредством оператора Wait(). Вызов H_Call() приводит к обычному последовательному вызову и синхронизации не требует.

Вызов параллельной процедуры:

```
P_Call(pname, param);
```

где

pname – имя вызываемой процедуры,

param – указатель на структуру локальных данных, передаваемую в качестве блока параметров; если параллельная процедура не имеет блока параметров, то в качестве второго параметра следует использовать значение EMPTY, определенное как указатель на пустую структуру типа EMPTY.

В результате вызова параллельной процедуры с помощью P_Call() порождается потенциально мигрирующий процесс, который в дальнейшем может быть выполнен на любом ПМ.

Вызов параллельной процедуры с назначением ее активации на ПМ с заданным номером:

```
P_Send(pname, param, pm);
```

где pm – номер процессорного модуля, на котором запускается процедура (выражение типа int). Если pm > N_PM, где N_PM – количество процессорных модулей в системе, то берется значение pm % N_PM. Здесь и далее операция % означает остаток после выполнения целочисленного деления.

В результате вызова параллельной процедуры с помощью P_Send() порождается готовый к выполнению процесс, который будет выполнен на ПМ с номером pm. Данный процесс уже не может мигрировать на другой ПМ.

Примечание. Количество процессорных модулей в вычислительной системе, а также номер ПМ, на котором выполняется данная активация, можно узнать с помощью системного вызова N_PM() (см. пункт "Вспомогательные системные вызовы").

В параллельном режиме работы допускается употребление отрицательного значения аргумента pm. В этом случае вызов будет послан на все ПМ системы с одним и тем же блоком параметров (широковещательно). Однако в этом случае блок параметров является не возвращаемым и значения его полей после вызова, вообще говоря, нельзя считать определенными.

Вызов параллельной процедуры в последовательном режиме:

- `_Call(pname, param);`

Процедура будет вызвана как обычная подпрограмма без порождения параллельного процесса и, следовательно, не требует синхронизации с использованием оператора Wait(). При этом накладные расходы на вызов практически будут эквивалентны вызову обычной последовательной подпрограммы.

Примечание. Не порождая параллельного процесса, данный вызов создает, тем не менее, активацию, которая выполняется так же, как и активации, порожденные вызовами P_Call() и P_Send(), и имеет свой счетчик синхронизации.

Описание заголовка параллельной процедуры

Заголовок параллельной процедуры является унифицированным и имеет следующий вид:

```
Parallel(pname,param)
PARAM(namestruct,*param);
```

где namectruct – имя структуры блока параметров.

Примечание. Назначение формальных параметров аналогично рассмотренным выше фактическим параметрам системных вызовов. Если процедура не использует блок параметров, то описание формальных параметров должно иметь вид:

```
Parallel(pname,param)
PARAM(EMPTY,*param);
```

Оператор синхронизации параллельных процессов.

```
Wait(); /* ••• •••••••••• */
```

Приостанавливает выполнение процедуры с целью ожидания завершения всех параллельных процессов, запущенных до начала выполнения Wait() в пределах данной активации. Под параллельным процессом понимается:

- процесс выполнения параллельной процедуры, активизированной посредством вызова P_Call() или P_Send();
- процесс выполнения операции доступа в общую или статическую память.

В родительской процедуре между оператором, порождающим параллельный процесс, и оператором Wait(), синхронизирующим дочерние параллельные процессы, могут использоваться любые операторы языка. При этом родительская процедура и запущенные в ней дочерние процессы могут выполняться параллельно.

Блок параметров.

Блок параметров предназначен для передачи локальных данных вызывающей процедуры в вызываемую процедуру и обратно из вызываемой процедуры в вызывающую.

Блок параметров – это структура данных языка С, которая формируется в области локальных данных вызывающей процедуры. Объявление блока параметров в вызывающей процедуре происходит следующим образом:

```
NEW_PARAM(namestruct, pb);
```

где namesruct – имя структуры блока параметров;

pb – имя блока параметров.

В RPC существует единственный способ передачи блока параметров в вызываемую параллельную процедуру – через указатель на соответствующую структуру.

ЗАМЕЧАНИЯ

1. Передача дочерней активации (за исключением вызова через H_Call()) указателей на локальные данные родительской активации либо статические данные недопустима, так как при параллельном выполнении программы нельзя быть уверенным в том, что дочерний процесс будет иметь доступ к адресному пространству, в котором они расположены.
2. В вызывающей процедуре элементы блока параметров, которые предназначены для возврата результатов, вообще говоря, нельзя использовать ни по чтению, ни по записи с момента вызова дочерней процедуры и до момента ее завершения, т.е. до момента срабатывания соответствующего оператора Wait(). Это вызвано тем, что ни момент отправки, ни момент возврата блока не являются определенными.
3. Две параллельные дочерние активации могут использовать один и тот же блок параметров только в том случае, если он не используется ими для возврата результатов в родительскую активацию.
4. Допускается передача блока параметров транзитом из вызывающей процедуры в вызываемую (т.е. формальный параметр param в

заголовке вызывающей процедуры можно использовать в качестве фактического параметра для вызова дочерней процедуры).

Поиск и диагностика ошибок неправильного использования блока параметров производится с помощью специальной утилиты – анализатора корректности RPC-программ, а также могут быть выявлены при выполнении программы в режиме поиска ошибок.

Динамическое выделение памяти.

Память в области ЛП данного ПМ может быть выделена и освобождена посредством использования стандартных функций языка C: malloc(), calloc(), realloc(), free() и т.д. Использование выделенных таким образом областей памяти производится путем непосредственной адресации. Доступ к ним имеют, естественно, только активации, выполняющиеся на ПМ, где выделена память.

Размещение в памяти, а также процесс доступа для данных, относящихся к двум другим классам памяти, допускающим динамическое выделение, а именно, разделяемой ОП и т.н. "статических" данных, описано в двух следующих параграфах.

Работа с разделяемой ОП.

Выделение памяти.

Выделение и освобождение областей разделяемой ОП происходит с помощью специальных системных вызовов, описываемых ниже.

Прежде всего, для доступа к разделяемой ОП следует завести дескриптор – переменную типа handleDM, например:

```
handleDM hA, hB;
```

Переменная этого типа обязательно должна быть глобальной или статической, попытка использования других классов памяти в данной версии библиотек, входящих в состав RPMSHELL, приведет к ошибке. Допускаются массивы переменных типа handleDM. С каждой областью РОП связывается своя переменная данного типа, ее использование для работы с другой областью, пока не освобождена предыдущая, приведет к сообщению об ошибке во время выполнения программы.

Выделение памяти в РОП и соответственно инициализация связанного с ней дескриптора производится посредством оператора G_Alloc(), а освобождение – оператора G_Free(). Данные операторы порождают параллельный процесс, поэтому требуют синхронизации (Wait()).

Формат оператора G_Alloc():

```
G_Alloc (SizeEl, NumEl, VCoeff, FirstPM, pH);
```

Здесь:

unsigned SizeEl – размер элемента в байтах;

unsigned long NumEl – количество размещаемых элементов;

unsigned VCoeff – коэффициент вертикального расслоения в элементах;

unsigned FirstPM – ПМ, где размещается нулевой элемент;

handleDM far *pH – указатель на дескриптор.

Если VCoeff равен 0, то берется VCoeff=1. Если FirstPM > N_PM, где N_PM – количество процессорных модулей в системе, то берется значение FirstPM % N_PM.

Формат оператора G_Free():

```
G_Free (pH);
```

Здесь:

handleDM far *pH – указатель на дескриптор.

Операторы доступа в разделяемую ОП.

Доступ в РОП осуществляется с помощью специальных векторных операций, причем элемент вектора может быть структурой произвольного вида. Операция доступа в РОП является параллельным процессом, поэтому она должна быть синхронизирована с помощью системного вызова Wait().

Ниже приводится описание синтаксиса операторов доступа к разделяемой ОП. В качестве иллюстрации к ним предлагаются эквивалентные кусочки программ на С, которые демонстрируют действие этих операторов, так, как оно происходило бы, если бы адресация РОП была сквозной и интересующий нас массив располагался в памяти сплошным куском.

В данных примерах следует считать, что указатель типа void *ga до начала выполнения приведенных фрагментов программы настроен на начало массива в РОП, а целая переменная size содержит размер элемента массива в байтах.

Чтение вектора с единичным шагом из ОП в ЛП

```
Load_1(la,pH,gOff,vl);
```

Запись вектора с единичным шагом из ЛП в ОП

```
Store_1(la,pH,gOff,vl);
```

Здесь:

void *la – указатель на локальную переменную,

handleDM far *pH – указатель на дескриптор области в РОП,

unsigned long gOff – смещение в глобальном массиве (в элементах),

unsigned vl – количество читаемых элементов вектора.

Эквивалентная программа на языке С:

```
{
  int i; char *p,*p1;
  p=la; p1=ga+gOff*size;
            /* p=ga+gOff*size; p1=la; – ••• •••••• */
  for(i=0;i<vl*size;i++) *p++=*p1++;
}
```

Чтение вектора с произвольным шагом из ОП в ЛП

```
Load_V(la,pH,gOff,step,vl);
```
Запись вектора с произвольным шагом из ЛП в ОП
```
Store_V(la,pH,gOff,step,vl);
```
Здесь:

long step – шаг между элементами читаемого (записываемого) из (в) ОП вектора (в количестве элементов);

Эквивалентная программа на языке С:
```
{
  int i,j; char *p,*p1;
  p=la; p1=ga+gOff*size;
            /* p=ga+gOff*size; p1=la; – ••• •••••• */
  for(i=0;i<vl;i++)
  {
    for(j=0;j<size;j++)
      *p++=*p1++;
    p1+=(step-1)*size;
            /* p+=(step-1)*size; – ••• •••••• */
  }
}
```
Чтение вектора данных по индексному вектору из ОП в ЛП
```
Load_I(la,pH,gOff,iv,vl);
```
Запись вектора данных по индексному вектору из ЛП в ОП
```
 Store_I(la,pH,gOff,iv,vl);
```
Здесь:

int *iv – указатель на индексный вектор, расположенный в локальной памяти.

Эквивалентная программа на языке С:
```
{ int i,j; char *p,*p1;
  p=la;           /* p1=la; – ••• •••••• */
  ga+=gOff*size;
  for(i=0;i<vl;i++)
  { p1=ga+iv[i]*size;
                  /* p=ga+iv[i]*size; – ••• •••••• */
    for(j=0;j<size;j++)
      *p++=*p1++;
  }
}
```
Чтение вектора по маске из ОП в ЛП
```
 Load_M(la,pH,gOff,mv,vl);
```
Запись вектора по маске из ЛП в ОП
```
 Store_M(la,pH,gOff,mv,vl);
```
Здесь:

int *mv – указатель на вектор маски, расположенный в локальной памяти.

Эквивалентная программа на языке С:

```
{
  int i,j; char *p,*p1;
  p=la;
  p1=ga+gOff*size;
              /* p=ga+gOff*size; p1=la; – ••• •••••• */
  for(i=0;i<vl;i++)
    if(mv[i]==1)
      for(j=0;j<size;j++)
        *p++=*p1++;
    else p1+=size;          /* p+=size; – ••• •••••• */
}
```

Обобщенная операция чтения фрагмента матрицы из ОП в ЛП

```
Load_G(la,pH,gOff,m,n,rinc,cinc);
```

Обобщенная операция записи фрагмента матрицы из ЛП в ОП

```
Store_G(la,pH,gOff,m,n,rinc,cinc);
```

Здесь:

unsigned m – число строк подматрицы;

unsigned n – число столбцов подматрицы;

long rinc – шаг для выхода на новую строку матрицы;

long cinc – шаг для выхода на новый столбец матрицы;

Эквивалентная программа на языке С:

```
{
  int l, k, i, j;
  k=0; ga+=gOff*size;
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      for(l=0;l<size;l++)
        la[k++]=ga[l+i*rinc*size+j*cinc*size];
/* ga[l+i*rinc*size+j*cinc*size]=la[k++]; ••• •••••• */
}
```

Отметим, что в Store_G() левые части операторов присваивания могут совпадать при различных парах значений i, j. Аналогичная ситуация может наблюдаться для правых частей операторов присваивания для Load_G. В этих случаях результат выполнения оператора является, вообще говоря, неопределенным. Отметим лишь, что таких ситуаций можно избежать, наложив на используемые значения параметров rinc и cinc некоторые естественные ограничения.

ЗАМЕЧАНИЯ. Зачастую на время выполнения операций доступа в ОП в ПМ, где данные операции запущены, разумно наложить запрет на активизацию потенциально мигрирующих процессов из дека, даже если ПМ простаивает (оператор Set_M()). Это нужно для того, чтобы данный ПМ не активизировал у себя излишнее число процессов во время кратковременных простоев ПМ. Иначе может возникнуть ситуация, приводящая к несбалансированной загрузке ПМ, в результате которой общие простои системы могут значительно превысить частные простои

отдельных ПМ из-за ожидания завершения операций доступа в ОП. Данное решение ограничивает "жадную" стратегию распределения работы между ПМ системы.

Для того чтобы минимизировать простои, связанные с ожиданием завершения доступа в ОП, существует специальный прием программирования, позволяющий осуществлять доступ в ОП на фоне вычислений. Более подробно этот вопрос рассмотрен в разделе, посвященном методам повышения эффективности рекурсивно-параллельного программирования.

Примеры использования блочных операций доступа к РОП.

В виду того, что формальное описание двух последних операций доступа к РОП не слишком наглядно, автору кажется уместным привести несколько простых примеров их вызова в качестве иллюстрации.

Пусть в РОП хранится матрица размера M∗N. Нам требуется считать в локальный массив, начиная с адреса la, подматрицу размером m∗n. Для этого можно использовать вызов

```
Load_G (la, pH, Off, m, n, N, 1);
```

Здесь Off задает смещение начального элемента считываемой подматрицы.

Вызов

```
Load_G (la, pH, 0, min(M,N), 1, N, *);
```

или

```
Load_G (la, pH, 0, 1, min(M,N), *, N+1);
```

позволяет считать главную диагональ (здесь вместо звездочки можно употребить любое значение).

Пример транспонирования матрицы при чтении:

```
Load_G (la, pH, 0, N, M, 1, N);
```

Считываемая подматрица может располагаться в исходном массиве не сплошным образом, а быть разреженной. Примеры таких вызовов, по-видимому, читателю не составит труда написать.

Одним из важных достоинств такого подхода к записи и чтению информации является возможность компактного хранения некоторых специальных видов разреженных матриц, которые часто возникают в задачах прикладной электродинамики, акустики, оптики, обработки изображений и т.д. При этом в РОП матрица хранится в сжатом виде, а считывается как развернутая.

Операторы размещения и копирования данных в локальной памяти процессорных модулей (статическая память).

Работа с данными в разделяемой оперативной памяти сопряжена со значительными затратами времени на доступ к ним. И хотя существуют приемы совмещения во времени операций доступа к РОП и вычислений (например, опережающая подкачка данных), полностью избежать потерь производительности не всегда удается.

Альтернативным решением является хранение в ЛП каждого процессорного модуля своей копии общих данных, особенно в случае, если они используются только по чтению. В этом случае операций чтения из РОП и связанных с ними потерь производительности можно избежать совсем. Ценой такого решения является существенно больший расход оперативной памяти, поскольку приходится хранить много копий одних и тех же данных.

Кроме того, если данные используются также и по записи, то программист должен сам позаботиться о том, чтобы параллельные процессы не пользовались устаревшими значениями и не обращались к одним и тем же данным одновременно (за исключением доступа по чтению). Последняя ситуация может быть выявлена, например, при выполнении программы в режиме поиска ошибок параллельного выполнения.

Ниже описаны предусмотренные в языке RPC средства размещения такого типа данных и средства доступа к ним. Для краткости мы будем называть их статическими (по историческим причинам), хотя, как будет видно из дальнейшего, имеется возможность и их динамического выделения.

Размещение статических данных.

Все данные, которые в языке C относятся к классу памяти static, являются статическими и в языке RPC. Поскольку при работе в параллельном режиме на каждом ПМ имеется своя копия кода и статических данных, к ним относится в полной мере все сказанное выше. То есть, если вы описали и инициализировали в программе некий статический массив, то вы можете в любой активации пользоваться этими данными путем прямой адресации, имея при этом в виду, что вы работаете именно с копией данных, расположенной на данном ПМ.

Под инициализацией здесь понимается присвоение значений именно при описании, поскольку в ходе выполнения программы присвоение приведет к изменению значений переменных только на данном процессорном модуле. Это относится в том числе и к действиям, производимым в функции main(), поскольку она запускается только на одном процессорном модуле (главном). Если вы хотите изменить значения

переменных либо элементов массива на всех ПМ, вам следует воспользоваться описанными ниже операторами. Они позволяют программисту считывать либо модифицировать данные, расположенные в локальной памяти других ПМ, как статические, так и выделенные динамически.

Чтобы сделать такой обмен данными возможным, для каждой области памяти следует завести дескриптор – статическую переменную типа handleSM. Дескриптор должен быть именно статической переменной (допускаются статические массивы дескрипторов), что позволяет иметь на каждом ПМ копии этих переменных с одним и тем же смещением в сегменте данных.

В языке предусмотрены следующие два варианта описания таких переменных. Во-первых, для работы с данными уже описанного статического массива, следует воспользоваться определением дескриптора с одновременной инициализацией:

```
handleSM_Init (hA, Array, SizeEl);
```

Здесь:

hA – имя описываемой переменной типа handleSM;

Array – имя описанного ранее статического массива;

SizeEl – размер элемента массива в байтах (unsigned).

Второй вариант описания предназначен для работы с данными, память под которые отводится динамически. Прежде всего, производится описание одной или нескольких переменных типа handleSM обычным образом, например,

```
handleSM hA, hB;
```

После этого, например, в функции main(), посредством вызова оператора S_Alloc() производится размещение одинаковых массивов данных на всех ПМ системы и связывание их с соответствующим дескриптором. Формат оператора S_Alloc():

```
S_Alloc (SizeEl, NumEl, pH, pPtr);
```

Здесь:

unsigned SizeEl – размер элемента массива в байтах;

unsigned NumEl – количество элементов массива;

handleSM *pH – указатель на статическую переменную типа handleSM;

void **pPtr – указатель на статическую переменную типа указатель.

В качестве последнего параметра передается указатель именно на статическую переменную (называемую ниже Ptr) типа указатель. В противном случае функция завершится аварийно. Результатом работы является выделение областей памяти требуемого размера на всех ПМ системы, связывание их с дескриптором и запись в местные копии переменной Ptr адреса выделенной области памяти (разумеется, на каждом модуле своего). К ней впоследствии можно обращаться напрямую с использованием адреса Ptr так, как это обычно делается с динамически

запрашиваемыми областями памяти. При работе в среде MS DOS объем памяти, запрашиваемой посредством одного оператора S_Alloc(), не должен превосходить размера сегмента.

Оператор S_Alloc() так же, как и остальные операторы для работы со статическими переменными, порождает параллельный процесс и, следовательно, требует синхронизации (оператор Wait()).

Освободить выделенную оператором S_Alloc() память можно с помощью вызова

```
S_Free(pH);
```

Здесь:

handleSM *pH – указатель на статическую переменную типа handleSM.

Операторы копирования статических данных.

Как отмечалось выше, работать со статическими переменными и областями памяти, в том числе выделенными через вызов S_Alloc(), можно обычным образом. При этом реально работа происходит с копией данных, расположенной на данном процессорном модуле.

Для копирования этих данных в соответствующие области памяти на других ПМ или получения доступа к ним в языке RPC предусмотрены следующие операторы.

Копирование своей копии данных на другой процессорный модуль либо на все остальные ПМ с единичным шагом:

```
S_CopyTo_1(pH,Off,vl,PM);
```

Здесь:

handleSM *pH – указатель на статическую переменную типа handleSM;

unsigned Off – смещение (в элементах);

unsigned vl – длина копируемого вектора (в элементах);

int PM – номер ПМ – получателя данных.

Если параметр PM имеет отрицательное значение, это означает, что копирование производится на все ПМ системы.

Копирование статических переменных, размещенных на другом ПМ, в соответствующие области памяти своего процессорного модуля производится посредством оператора

```
S_CopyFrom_1(pH, Off, vl, PM);
```

Здесь параметр PM>=0 задает номер ПМ – источника данных. В обоих случаях, если PM больше или равен количеству ПМ в системе N_PM в качестве адресата будет взято значение PM по модулю N_PM.

Операторы для доступа в разделяемую общую память (группа операторов с именами Load_*() и Store_*()) копируют группы, вообще говоря, разрозненных элементов в РОП в сплошную область локальной памяти и обратно (смотри соответствующее описание). Поэтому для

работы со статическими данными могут оказаться полезными операторы, реализующие аналогичный же доступ к копии статических данных, размещенной на текущем ПМ. Это операторы с именами S_Load_*() и S_Store_*(), которые являются чисто последовательными и не требуют синхронизации.

Загрузка из статической области данных в сплошной массив ЛП с единичным шагом

```
S_Load_1(la, pH, Off, vl);
```

Здесь:

void *la – адрес области в ЛП;

handleSM *pH – указатель на статическую переменную типа handleSM;

unsigned Off – смещение (в элементах);

unsigned vl – длина копируемого вектора (в элементах).

Передача данных из сплошного массива в ЛП в статическую область памяти с единичным шагом

```
S_Store_1(la, pH, Off, vl);
```

Здесь:

void *la – адрес области в ЛП;

handleSM *pH – указатель на статическую переменную типа handleSM;

unsigned Off – смещение (в элементах);

unsigned vl – длина копируемого вектора (в элементах).

Алгоритм копирования данных аналогичен алгоритму работы вызовов Load_1() и Store_1(), поэтому для уточнения деталей следует обратиться к описанию этих двух операторов.

Для доступа в РОП нами было рассмотрено пять различных вариантов копирования данных. Соответственно столько же аналогичных вариантов доступа предусмотрено и для работы со статическими данными. Мы ограничимся ниже перечислением форматов соответствующих операторов. Назначение их должно быть понятно из их имен. Формат и смысл параметров pH, Off, PM, la такой же, как у описанных выше операторов S_CopyTo_1(), S_CopyFrom_1(), S_Load_1() и S_Store_1(). Остальные параметры имеют такой же формат и смысл, что и соответствующие параметры операторов доступа в РОП Load_*() и Store_*(), если не оговорено противное. Процесс копирования также аналогичен описанному процессу для операторов Load_*() и Store_*().

Запись своей копии данных на другой либо на все остальные ПМ с произвольным шагом, а также чтение чужой копии статических данных

```
S_CopyTo_V(pH,Off,step,vl,PM);
S_CopyFrom_V(pH,Off,step,vl,PM);
```

Загрузка из статической области данных в сплошной массив ЛП с произвольным шагом

```
S_Load_V(la,pH,Off,step,vl);
```

Передача данных из сплошного массива в ЛП в статическую область памяти с произвольным шагом

```
S_Store_V(la,pH,Off,step,vl);
```

Запись своей копии данных на другой либо на все остальные ПМ по индексному вектору, а также чтение чужой копии статических данных

```
S_CopyTo_I(pH,Off,iv,vl,PM);
S_CopyFrom_I(pH,Off,iv,vl,PM);
```

Загрузка из статической области данных в сплошной массив ЛП по индексному вектору

```
S_Load_I(la,pH,Off,iv,vl);
```

Передача данных из сплошного массива в ЛП в статическую область памяти по индексному вектору

```
S_Store_I(la,pH,Off,iv,vl);
```

Запись своей копии данных на другой либо на все остальные ПМ по маске, а также чтение чужой копии статических данных

```
S_CopyTo_M(pH,Off,mv,vl,PM);
S_CopyFrom_M(pH,Off,mv,vl,PM);
```

Загрузка из статической области данных в сплошной массив ЛП по маске

```
S_Load_M(la,pH,Off,mv,vl);
```

Передача данных из сплошного массива в ЛП в статическую область памяти по маске

```
S_Store_M(la,pH,Off,mv,vl);
```

Обобщенная операция копирования своей копии данных на другой либо на все остальные ПМ, а также чтение чужой копии статических данных

```
S_CopyTo_G(pH,Off,m,n,rinc,cinc,PM);
S_CopyFrom_G(pH,Off,m,n,rinc,cinc,PM);
```

ЗАМЕЧАНИЕ. В этом, а также двух следующих операторах в отличие от операторов доступа в РОП (Load_G() и Store_G()) параметры rinc и cinc имеют тип int:

```
int rinc, cinc;
```

Обобщенная операция загрузки из статической области данных в сплошной массив ЛП

```
S_Load_G(la,pH,Off,m,n,rinc,cinc);
```

Обобщенная операция передачи данных из сплошного массива в ЛП в статическую область памяти

```
S_Store_G(la,pH,Off,m,n,rinc,cinc);
```

Заданный набор статических переменных (в том числе массивов) можно скопировать за одну операцию (что предпочтительнее) с использованием операций копирования

```
S_CopyTo_Set(PM,ptr1,size1,...,ptrN,sizeN,NULL);
```

или

```
S_CopyFrom_Set(PM,ptr1,size1,...,ptrN,sizeN,NULL);
```

Здесь аргумент целого типа PM задает номер процессорного модуля, откуда (или куда) надо произвести копирование, и его использование подчиняется тем же правилам, что и для ранее описанных операций копирования статических данных. Указатели ptr1, ptr2,..., ptrN задают адреса областей статической памяти подлежащих копированию, а size1, size2,..., sizeN (тип int) – соответственно задают их размеры в байтах. В качестве последнего аргумента следует задать нулевой указатель как признак завершения списка.

Дополнительные средства синхронизации процессов.

Для организации работы с "критическими секциями" программы, осуществляющими параллельный доступ к разделяемому ресурсу, каждый процессорный модуль RPM имеет собственный набор "замков" для синхронизации процессов внутри модуля. Т.е. разделяемому ресурсу должен соответствовать номер "замка" и номер модуля, на котором будут обрабатываться соответствующие "критические секции". Работа с "замками" осуществляется посредством следующих системных вызовов:

```
Lock(l); /* ••••••• ••••• */
```

или

```
Unlock(l); /* ••••••• ••••• */
```

Здесь l – номер замка (целое число от 0 до L-1, где L – число используемых замков, задаваемое в опции при запуске пользовательской программы, по умолчанию L=1).

Если в момент вызова Lock(l) соответствующий замок закрыт, выполнение текущей активации будет приостановлено на данном операторе Lock(l) до открытия замка l.

Использование системных функций Lock(l) и Unlock(l) может привести к дедлоку. Например, если один процесс, захватив замок A, ждет, пока откроется замок B, а процесс, захвативший замок B, в свою очередь ждет освобождения A. Контроль над возникновением таких ситуаций целиком возлагается на пользователя. Однако можно предложить несколько правил, соблюдение которых гарантирует корректность программы в этом смысле.

Во-первых, если вам требуется захватить несколько замков в рамках одной активации, располагайте вызовы функций Lock(l) всегда в одном и том же порядке, например, в порядке возрастания номеров замков. Во-вторых, обязательно освобождайте все захваченные замки: перед выходом из активации, и перед вызовом оператора Wait(), который синхронизирует хотя бы один дочерний процесс.

Вспомогательные системные вызовы.

Для повышения эффективности параллельных вычислений при решении некоторого класса задач может потребоваться информация о имеющемся количестве процессорных модулей в системе и номере

модуля, на котором выполняется данная активация. Для этого программист может воспользоваться следующим системным вызовом

```
N_PM(&N,&mn);
```

который возвращает:

int N – общее количество ПМ в вычислительной системе,

int mn – номер ПМ, на котором выполнен данный системный вызов.

Благодаря данному системному вызову пользователь может динамически настроить отдельные процессы и задачу в целом на конкретную конфигурацию вычислительной системы. Данное средство будет особенно полезным, если для порождения активаций параллельных процедур пользователь применяет системный вызов P_Send(). В этом случае он может разбить задачу так, что количество процессов окажется кратным числу ПМ, и самостоятельно распределить их между ПМ.

Отметим, что данный системный вызов есть смысл использовать только в параллельном режиме исполнения, поскольку во всех последовательных режимах будут возвращены значения N=1, mn=0;

С целью повышения эффективности параллельного выполнения программы может использоваться оператор

```
Set_M();
```

который устанавливает запрет на выборку потенциально мигрирующих процессов из дека (режим монопольного захвата ПМ) вплоть до завершения данной активации параллельной процедуры (обычно листьевой). Более подробно этот вопрос рассмотрен в разделе, посвященном описанию причин снижения эффективности рекурсивно-параллельных программ и методов борьбы с ними.

Дополнительные операторы управления работой программы в параллельном режиме (для MS DOS).

ВНИМАНИЕ: Данный параграф настоятельно рекомендуется прочитать перед тем, как пытаться запустить RPC-программу на выполнение в параллельном режиме на локальной сети компьютеров под управлением MS DOS. Несоблюдение сформулированных ниже требований с большой вероятностью приведет к ошибкам во время выполнения вплоть до зависания.

Для повышения эффективности работы распределительного механизма в параллельном режиме предусмотрены переключения с выполнения одного процесса на другой, которые могут происходить при обработке сетевых прерываний. В то же время некоторые прерывания DOS не являются реентерабельными. То же самое относится к библиотечным функциям, если они вызывают в процессе своей работы соответствующие прерывания DOS. Если во время выполнения такой функции происходит сетевое прерывание, вызывающее переключение программы на

выполнение другого параллельного процесса, то последствия непредсказуемы.

Для того, чтобы избежать подобных ситуаций в настоящей версии библиотеки поддержки параллельного режима выполнения RPC-программы предусмотрены два варианта действий.

Первый заключается в том, что все подозрительные библиотечные функции (а может быть и просто все) предварительно объявляются запрещенными для переключений процессов посредством одного из трех ниже перечисленных операторов, а именно того, который соответствует типу возвращаемого этой функцией значения.

Для функций, возвращающих значения, относящиеся к одному из целочисленных типов (как знаковых, так и беззнаковых), а также значения типа указатель следует воспользоваться оператором Subst(), для функций, возвращающих значения типов float или double – оператором SubstF(), и, наконец, для функций, не возвращающих значений – оператором SubstV(). Формат перечисленных операторов:

```
Subst (type, Name(arg1,...,argN));
SubstF (type, Name(arg1,...,argN));
SubstV (Name(arg1,...,argN));
```

Здесь:

type – тип значения, возвращаемого функцией,

Name – имя функции.

Примеры использования:

```
Subst (int, getchar());
Subst (void *, malloc(a));
SubstF (double, sin(a));
SubstV (putimage(l, t, b, o));
```

Данные операторы должны встретиться в тексте программы до первого использования соответствующей функции, однако после описания ее прототипа, если таковое имеется (обычно в стандартном заголовочном файле).

Вторым способом, иногда более эффективном (особенно, если такие функции встречаются очень часто, например, в цикле) является явный вызов функций запрещения или разрешения переключений текущего процесса (v_DS() и v_ES() соответственно).

Пример использования:

```
v_DS();
for (i=0; i<n; ++i)
  putpixel(i,j,a[i]);
v_ES();
```

Виртуальная архитектура RPM

Структура вычислительной системы

В данном параграфе структура рекурсивно-параллельного мультипроцессора представлена в самом общем виде, без какой бы то ни было детализации. С одной стороны, такого представления прикладному программисту вполне достаточно, чтобы понять, как будет организовано выполнение его программы, а с другой – позволяет не привязываться к какой-то конкретной реализации вычислительной системы. RPM может быть реализована и как супер-ЭВМ, и как сеть из персональных компьютеров. Кстати, именно такой реализацией и является среда RPMSHELL.

RPM состоит из универсальных процессорных модулей, разделяемой (или общей – возможна и такая реализация) оперативной памяти и, возможно, управляющей ЭВМ, которые объединены между собой некоторой коммуникационной средой. Наличие управляющей ЭВМ, если таковая имеется, отнюдь не означает, что система асимметрична: в таком качестве может выступать просто ПМ, выделенный для связи с внешним миром, реализации пользовательского интерфейса и/или ввода и вывода. В самом общем виде такая структура представлена на рис. 2.

Рис. 2. Структура RPM.

Еще раз подчеркнем, что представленная структура является виртуальной, т.е. такой, какой ее должен воспринимать пользователь, разрабатывающий программы для RPM. Реальных конфигураций RPM может быть множество. Например, общая память может состоять из отдельных модулей, физически распределенных по всем ПМ системы. Именно так она организована в программном обеспечении, образующем среду RPMSHELL.

Основное назначение ПМ:

- выполнение активаций процедур параллельного вычислительного процесса;
- организация динамического планирования и распределения работы по ПМ системы;
- организация взаимодействия с другими ПМ, а также с ОП и управляющей ЭВМ, если таковая имеется.

Внутренняя структура ПМ может быть параллельной, обеспечивая тем самым выполнение некоторых системных функций (в том числе операций доступа в РОП) на фоне вычислений.

ОП системы является разделяемой, т.е. она имеет единое адресное пространство для всех ПМ. ОП является "удаленной" для процессорных модулей, поэтому для повышения эффективности работы доступ к ней организуется с помощью блоковых операций. С этой целью в ОП рекомендуется размещать только основные структурированные массивы обрабатываемых данных, а все промежуточные результаты хранить в локальной памяти ПМ. Для того чтобы доступ в ОП не останавливал работу модуля, блоковые операции организуются как параллельные процессы и выполняются на фоне основных вычислений.

Кроме динамических (стековых) данных и данных, размещаемых в разделяемой ОП, в RPM предусмотрен еще один класс данных, называемых статическими. Это данные, к которым можно обращаться непосредственно из любой процедуры программы, причем на каждом ПМ имеется своя их копия. Использование этого класса памяти может позволить в ряде случаев повысить эффективность программирования.

Коммуникационная среда обеспечивает связь между отдельными ПМ, ОП и управляющей ЭВМ. Ключевой характеристикой коммуникационной среды, которую необходимо учитывать при программировании, является ее пропускная способность. Даже в рамках решения одной задачи почти всегда можно предложить несколько алгоритмов, предъявляющих различные требования к пропускной способности коммуникационной среды. В параграфе, посвященном рассмотрению приемов повышения эффективности РП-программ, вопрос о разумной с этой точки зрения организации процесса доступа в память рассмотрен более подробно.

Механизм порождения и активизации параллельных процессов

Основные соглашения по порождению и распределению активаций параллельных процедур

Очевидно, что метод рекурсивно-параллельного программирования и программа, написанная в соответствующем стиле, могут быть реализованы параллельным образом только в том случае, если мы откажемся от

традиционного способа вызова функции. Необходимо реализовать его таким образом, чтобы оператор вызова не приводил к передаче управления, а лишь только помещал порожденную активацию параллельной процедуры в некий накопитель, из которого впоследствии этот вызов можно было бы извлечь и активизировать на свободном процессорном модуле.

Основным требованием к алгоритму обработки операторов вызова параллельных функций является возможность обеспечения эффективной работы распределенного алгоритма динамической балансировки загрузки вычислительной системы. Под "эффективной" работой мы подразумеваем такую, которая осуществляет по возможности равномерное распределение вычислений по системе, затрачивая на это минимальное количество передач. Последнее очень важно, поскольку в любой параллельной вычислительной системе коммуникационная сеть является критическим ресурсом, во многом определяющим эффективность работы всей системы.

С этой точки зрения очень важен выбор подходящей структуры данных для хранения готовых к выполнению активаций параллельных процедур и алгоритмов их распределения. Основные структуры данных, используемые для организации параллельного выполнения программы с динамической балансировкой загрузки, представлены на рис. 3.

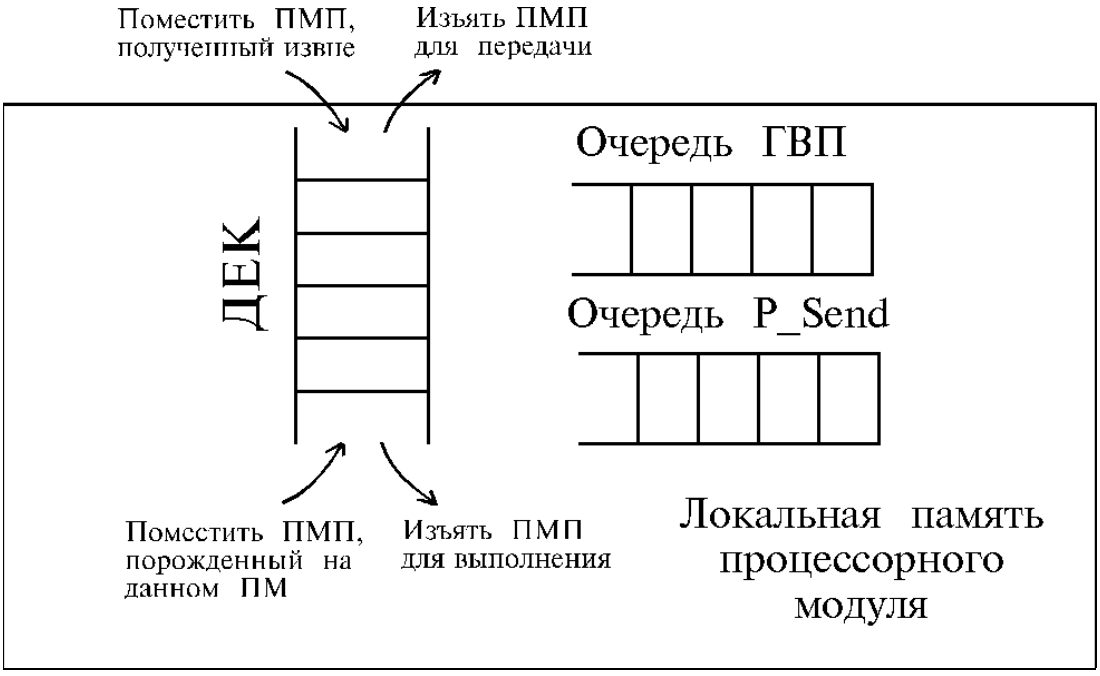


Рис. 3. Основные структуры данных для управления распределением работы.

Собственно алгоритм распределения работы описан ниже, а здесь мы остановимся на вопросах организации хранения и очередности

активизации параллельных процедур, которые могут быть переданы на исполнение любому ПМ. Мы будем называть их потенциально мигрирующими процессами (ПМП).

Потенциально мигрирующими являются только активации параллельных процедур, порожденные посредством оператора P_Call(). В языке РП-программирования, описанном выше, для вызова параллельных процедур предусмотрены также оператор последовательного выполнения параллельной активации (H_Call()) и оператор порождения параллельной активации с направлением ее для выполнения на конкретный ПМ (P_Send()).

Для хранения ПМП, порожденных на данном процессорном модуле, в локальной памяти каждого ПМ отводится область, организованная как дек (т.е. очередь с двумя входами), и называемая в дальнейшем ДЕКом. Оказывается, что такая организация хранения ПМП идеально подходит для эффективной работы распределительного механизма.

ДЕК имеет два конца: внутренний, используемый для помещения вновь порожденных ПМП и изъятия ПМП для активизации на ПМ, на котором он был порожден, и внешний, используемый для выборки ПМП, передаваемого другому ПМ.

Активации процедур, однажды активизированные, не могут быть переданы никакому другому ПМ. В случае, если выполнение некоторой активации было приостановлено на операторе синхронизации Wait(), управление может быть передано другой временно приостановленной активации, если завершились все ожидаемые ею параллельные процессы. Такую активацию мы будем называть готовым к выполнению процессом (ГВП). Готовые к выполнению процессы организованы в очередь, подчиняющуюся обычной дисциплине FIFO, и имеют высший приоритет при выборе процесса для активизации.

Если ГВП отсутствует, то просматривается очередь параллельных процессов, порожденных посредством оператора P_Send(), и ожидающих выполнения на данном ПМ. На каждом ПМ имеется своя очередь для таких процессов.

Наконец, если и таких процессов нет, то из своего дека ПМ может быть извлечен (с внутреннего конца) и активизирован очередной ПМП. Это делается в том случае, если в процессорном модуле не установлен запрет на активизацию ПМП (посредством оператора Set_M()).

Каждая активация параллельной процедуры имеет свой цикл жизни, который можно описать диаграммой перехода состояний, изображенной на следующем рисунке (рис. 4).

Рис. 4. Диаграмма состояний параллельных процессов в RPM.

Активации процедур, порожденные посредством оператора P_Call(), становятся потенциально мигрирующими процессами, и являются таковыми вплоть до извлечения из ДЕКа и активизации на одном из ПМ. Активации процедур, созданные с использованием оператора P_Send(), немедленно передаются на соответствующий ПМ и помещаются в очередь P_Send(). После этого они могут быть извлечены из очереди и активизированы в соответствии со сформулированным выше порядком.

Активации параллельных процедур, порожденные путем использования оператора H_Call(), не являются параллельными процессами. Они немедленно получают управление, так, как это происходит при обычном вызове последовательных процедур и функций. Родительская активация в этом случае становится пассивным процессом и получает управление (становится активным процессом) немедленно по завершении этой дочерней активации.

С каждой активацией параллельной процедуры независимо от способа ее запуска связана специальная переменная в локальной памяти ПМ, которая называется счетчиком синхронизации. При первом входе в активацию ее значение равно нулю. При каждом запуске параллельного процесса из данной активации ее значение увеличивается на единицу, а

при завершении – уменьшается (независимо от того, в каком состоянии находится активация, запустившая процесс). Параллельными процессами являются вызовы параллельных процедур посредством операторов P_Call() и P_Send(), операции доступа в разделяемую общую память и ряд других операторов, перечисленных в описании языка РП-программирования. При запуске параллельного процесса выполнение текущей активации продолжается.

Активный процесс может быть отложен, как видно из приведенной диаграммы, лишь том в случае, если встретился оператор синхронизации Wait(), а текущее значение счетчика синхронизации отлично от нуля. Процесс становится пассивным вплоть до момента, когда будут завершены все запущенные им параллельные процессы, и, следовательно, его счетчик синхронизации становится равным нулю. В этом случае он помещается в очередь ГВП, откуда может быть выбран и активизирован, как это описано выше.

Каждый ПМ имеет ДЕК, в котором содержатся ПМП, очередь процессов, порожденных оператором P_Send(), и очередь готовых к выполнению процессов. Считается, что ПМ имеет работу, если не пуст ДЕК, не пуста очередь P_Send(), не пуста очередь готовых к выполнению процессов или есть отложенные процессы. В противном случае ПМ будет безработным. ПМ может поделиться работой, если в его ДЕКе количество потенциально мигрирующих процессов превышает некоторое значение (специально устанавливаемый параметр – обычно 0 или 1).

Описание потенциально мигрирующего процесса вместе со значениями входных параметров, а также некоторой дополнительной информацией, необходимой для корректной организации вычислительного процесса, является порцией работы, которой можно поделиться с другими ПМ. Она содержит все необходимые данные для выполнения активации процедуры на любом ПМ и возврата выходных параметров в родительскую активацию.

На рис. 5 изображена диаграмма изменения состояний процессорного модуля.

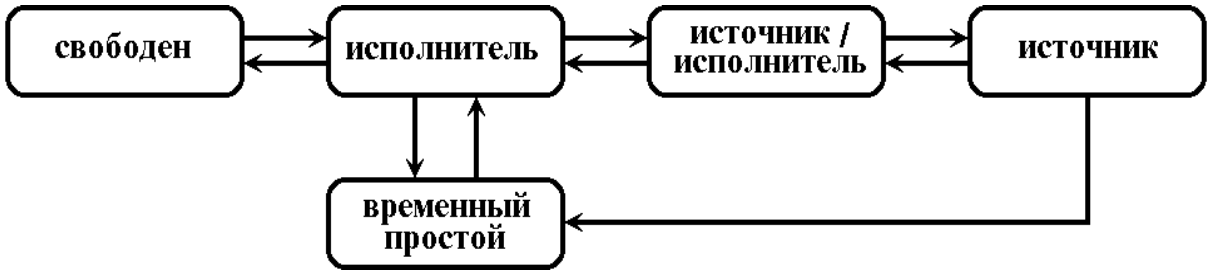


Рис. 5. Диаграмма состояний процессорного модуля.

ПМ находится в состоянии "свободен", если он не имеет активизированных процессов, очередь P_Send() и ДЕК пусты.

ПМ находится в состоянии "исполнитель работы", если он имеет работу, но не может поделиться ею с другими ПМ.

ПМ находится в состоянии "источник/исполнитель работы", если он имеет работу и может ею поделиться.

ПМ находится в состоянии "источник работы", если он функционирует в монопольном режиме, при этом имеет работу только в виде отложенных процессов и потенциально мигрирующих процессов.

ПМ находится в состоянии временного простоя, если он имеет работу только в виде отложенных активизированных процессов.

Режим монопольного захвата процессорного модуля

Описывая механизм порождения и активизации параллельных процессов в RPM, следует сказать и о том, как реализован режим монопольного захвата модуля (оператор Set_M()). Применяется он как средство борьбы с так называемым "эффектом надкусывания" активаций, который заключается в том, что модуль, простаивающий в ожидании завершения операций доступа в общую память, может извлечь из дека и активизировать слишком много ПМП. Результатом таких действий может стать существенная неравномерность загрузки системы и, следовательно, падение эффективности. Установление временного запрета на извлечение потенциально мигрирующих процессов из дека позволяет легко устранить эту проблему.

Поскольку прикладному программисту требуется иметь представление о том, как происходит установка и отмена режима монопольного захвата ПМ, ниже приводятся необходимые разъяснения.

На каждом процессорном модуле системы заводится статическая переменная-счетчик (назовем ее M_PM). При запуске программы она получает нулевое значение. Перед выборкой из дека (для себя) модуль проверяет текущее значение этой переменной. Ее ненулевое значение означает запрет на выборку. На передачу активаций из дека другим ПМ значение M_PM никак не влияет. Точно так же оно не влияет и на процесс выборки активаций из очереди P_Send(), поскольку они все равно должны быть исполнены на текущем процессорном модуле.

Кроме того, с каждой активацией параллельной процедуры, независимо от способа ее запуска – последовательно или параллельно, связывается переменная, которую мы назовем здесь M_Act. Она принимает значение 1 или 0, в зависимости от того, устанавливала ли данная активация режим монополии. Приведенная ниже таблица описывает, в каких случаях происходит изменение этих управляющих переменных.

Событие	Изменение значений M_PM и M_Act
Запуск программы	M_PM=0
Начало исполнения активации параллельной процедуры, независимо от способа вызова	M_Act=0
Вызов оператора Set_M()	Если M_Act=0, то M_Act=1, M_PM увеличить на 1
Постановка в дек, выход из активации	Если M_Act=1, то M_Act=0, M_PM уменьшить на 1
Запуск из ГВП, остановка на Wait()	Переменные не изменяются

Как можно видеть, сброс режима монопольного захвата ПМ происходит автоматически при завершении активации, что естественно, а также при помещении ею вновь порожденной активации в дек. Последнее нужно для того, чтобы избежать невозможности извлечения из дека только что помещенной туда активации – в ряде случаев это могло бы привести к невозможности дальнейшего выполнения программы. Впрочем, и так очевидно, что использование оператора Set_M() в нелистьевых активациях неразумно. Несложно убедиться в том, что отменить режим монопольного захвата модуля могут только те активации, которые его установили.

Алгоритм динамической балансировки загрузки

Описание алгоритма

Основной целью разработанного распределительного механизма являлось обеспечение по возможности более равномерной загрузки процессорных модулей при минимальном количестве передач работы как на этапе первоначального распределения работы, так и при ее перераспределении в ходе вычислений (динамическая балансировка загрузки ПМ).

В значительной степени это достигается организацией хранения ПМП в ДЕКе, что обеспечивает естественную сортировку активаций процедур по уровню вложенности и, следовательно, по размеру. Здесь под размером активации мы понимаем количество порождаемых листьевых активаций, поскольку исходим из предположения, что об их вычислительной емкости априори ничего не известно. В ДЕКе наиболее крупный ПМП всегда располагается ближе к внешнему концу, а его размер (при условии деления работы при рекурсивном вызове пополам) не меньше суммарного размера остальных ПМП, находящихся в ДЕКе. Поскольку для передачи работы на

другой модуль из ДЕКа извлекается ПМП с внешнего конца, это позволяет свести количество передач работы к минимуму.

Второй составляющей распределительного механизма является алгоритм принятия решения о передаче или запросе работы и выбор модуля приемника. Этот алгоритм и описывается ниже.

Такое решение принимается процессорным модулем самостоятельно на основании имеющейся в его распоряжении информации, может быть частично устаревшей, о распределении работы по системе. Эта информация представлена в виде вектора состояния Vs (на каждом ПМ имеется своя копия вектора), i-я компонента которого Vs[i] содержит информацию о состоянии i-го ПМ. Она может принимать одно из трех значений:

0 ("свободен") – модуль простаивает из-за отсутствия работы;

1 ("исполнитель") – у модуля есть работа, но не столько, чтобы он мог ею поделиться;

2 ("источник") – у модуля достаточно работы, чтобы часть ее он мог передать другому ПМ.

Мы считаем, что модуль может поделиться работой, если количество ПМП в его ДЕКе превышает некоторое пороговое значение MinWork, которое является одним из настроечных параметров распределительного механизма и, как видно из дальнейшего, может изменяться по ходу работы.

Если у i-го ПМ в ходе вычислений изменяется состояние, то он, вообще говоря, должен разослать новое значение компоненты Vs[i] всем другим ПМ. Однако в целях снижения нагрузки на коммутационную сеть предлагаемый распределительный механизм делает это не всегда. Более детально этот механизм описан ниже.

Мы будем выделять два основных этапа выполнения программы. Этап 1 соответствует процессу первоначального распределения работы по системе в ходе рекурсивной развертки программы. Он начинается с запуском программы и заканчивается в тот момент, когда в системе не остается ни одного безработного ПМ. Этап 2 соответствует собственно вычислениям и процессу обратного хода рекурсии, то есть свертыванию вычислительного процесса. Если вычислительный процесс имеет "гамачную" структуру, то в тот момент, когда в системе остается всего одна активация (процесс свернулся в точку), происходит переключение в режим этапа 1. Если i-й ПМ по имеющейся у него информации установил, что произошло изменение этапа вычислений, он оповещает об этом всех остальных.

На упомянутых двух этапах могут быть установлены различные значения порогового значения MinWork, которое задается в файле конфигурации перед запуском программы отдельно для этапов 1 и 2. На этапе 1, как правило, есть смысл устанавливать это значение равным 0, если для запуска двух порожденных дочерних процессов используется

пара операторов P_Call() и H_Call(), или 1, если используется пара операторов P_Call() и P_Call(). На этапе 2 в принципе может оказаться полезным использовать большее значение MinWork как средство для борьбы с так называемым "эффектом дребезга", когда на завершающей стадии вычислений начинают передаваться все менее емкие ПМП. Впрочем, этого можно избежать, просто установив разумные ограничения на глубину рекурсии, что с нашей точки зрения является более предпочтительным.

Кроме того, на различных этапах выполнения программы применяется различная стратегия распределения работы.

На этапе 1 инициатива передачи работы принадлежит ПМ-источнику работы. Как только у него в ДЕКе появляется ПМП, который можно отдать, он выясняет, используя свою копию вектора Vs, есть ли ПМ, которому требуется работа. Если таковой находится, то ПМ-источник посылает ему работу и модифицирует соответствующую компоненту своей копии вектора Vs. Собственно для поиска ПМ-приемника предусмотрено два варианта, выбор между которыми производится при запуске программы путем соответствующей установки в файле конфигурации.

Основным (и действующим по умолчанию) является вариант, предназначенный для случая, когда порождаемый программой параллельный вычислительный процесс имеет структуру достаточно сбалансированного бинарного дерева. Под словами "достаточно сбалансированного" подразумевается, что дерево не имеет листьевых активаций уровня вложенности меньше, чем $\log_2 N$, где N – количество ПМ в системе. В этом случае поиск ПМ-приемника происходит по следующему алгоритму:

1. Если сосед справа имеет работу, заканчиваем поиск (считаем, что не нашли), в противном случае переходим к п.2. Соседом справа мы считаем ПМ с номером на единицу большим, для (N-1)-го ПМ соседом справа считаем 0-й ПМ.
2. Определяем длину цепочки безработных ПМ, стоящих подряд вслед за соседом справа, и выбираем в качестве приемника работы ПМ, стоящий посередине этой цепочки. Если таких ПМ два (цепочка имеет четную длину), то берем ПМ, стоящий правее.

Несложно убедиться в том, что если на каждом шаге рекурсии работа разбивается поровну, то данный алгоритм начального распределения разместит ее максимально ровным образом (а если N – степень двойки, то просто поровну), использовав при этом N-1 передачу. При этом, если в ходе рекурсивного разбиения работы она делится на неравные куски, то рекомендуется больший кусок оставить себе, породив соответствующую активацию посредством оператора H_Call().

Альтернативный вариант поиска ПМ-приемника работы на этапе 1 отличается тем, что поиск происходит и в том случае, если правый сосед

ПМ-источника имеет работу. В этом случае, двигаясь вправо, мы находим ближайшую цепочку безработных ПМ и выбираем в качестве приемника работы ПМ, стоящий в середине этой цепочки.

Недостатком этого варианта является то, что существует вероятность того, что два или более процессорных модуля одновременно направят работу одному ПМ, что приведет к некоторой неравномерности первоначального распределения работы и, следовательно, необходимости дополнительных передач работы на этапе 2. Однако, с другой стороны, данный вариант позволяет избежать преждевременного прекращения первоначального распределения работы, которое могло бы случиться, если бы в дереве активаций встретилась листьевая вершина с уровнем вложенности меньше $\log_2 N$, а для размещения работы использовался бы основной вариант алгоритма.

Как только один из ПМ, основываясь на значениях компонент своей копии вектора Vs, обнаруживает, что в системе не осталось безработных ПМ, он принимает решение о наступлении этапа 2 и рассылает всем соответствующее сообщение.

На этапе 2 инициатива передачи работы принадлежит ПМ-приемнику работы. Если на некотором модуле нет готовых к выполнению процессов, пуста очередь P_Send() и ДЕК, а также не установлен режим запрета выборки из ДЕКа (монопольный захват ПМ процессом), он считает себя безработным. Основываясь на значениях компонент своей копии вектора Vs, он находит, двигаясь влево, ПМ-источник и посылает ему сообщение о запросе работы. Тот, в свою очередь, присылает либо дескриптор активации параллельной процедуры с блоком параметров, либо отказ. В случае получения отказа безработный ПМ корректирует свою копию вектора Vs и производит повторный поиск работы.

Такой алгоритм динамической балансировки действует до тех пор, пока процесс не свернется до ширины 1. Произойти это может только на ПМ, на котором вычислительный процесс начинался (у нас это всегда 0-й ПМ), и соответственно только он диагностирует эту ситуацию и рассылает всем сообщение об окончании этапа 2 и установке режима этапа 1. Это делается на тот случай, если вычислительный процесс несколько раз сворачивается и разворачивается, то есть имеет так называемый "гамачный" характер. В этом случае алгоритм первоначального распределения работы повторяется в начале каждого "гамака".

Отметим здесь также тот факт, что если максимальная ширина вычислительного процесса меньше N, то режим этапа 2 не будет установлен никогда, однако в этом случае именно алгоритм распределения, действующий на этапе 1, обеспечит распределение работы по системе с минимальным количеством передач.

Для обеспечения эффективной работы алгоритма динамической балансировки при принятии решения о передаче или запросе работы ПМ необходимо иметь некоторую (но не всю) информацию о текущем

состоянии других ПМ. Поэтому ПМ, состояние которого изменилось (имеются в виду три состояния, находящие отражение в векторе Vs), должен, вообще говоря, сообщить об этом другим, прислав новое значение компоненты вектора, соответствующей данному ПМ. Однако такие пересылки информации также создают определенную нагрузку на коммутационную систему и их количество желательно по возможности сократить. В нашем случае мы достигли этого путем исключения пересылок значений вектора состояния в тех случаях, когда эти значения не влияют на работу действующего в данный момент алгоритма динамической балансировки. С учетом особенностей описанного выше алгоритма и в зависимости от этапа вычислений достаточными являются пересылки информации, перечисленные в следующей таблице.

Установленное состояние	Действия процессорного модуля
Этап 1	
"Свободен"	Компонента вектора Vs посылается всем
"Исполнитель"	Если предыдущее состояние было "свободен", то всем рассылается компонента вектора Vs
"Источник"	Никаких сообщений не посылается
"Этап 2"	Рассылается сообщение об установке этапа, а также компонента вектора Vs, если состояние модуля "источник"
Этап 2	
"Свободен"	Компонента вектора Vs посылается всем в случае, если не найден источник работы
"Исполнитель"	Всем рассылается компонента вектора Vs
"Источник"	Всем рассылается компонента вектора Vs
"Этап 1"	Главный (0-й) ПМ диагностирует ситуацию и рассылает всем сообщение об установке режима "Этап 1"

Основные моменты организации передачи данных и служебной информации.

Очевидно, что для эффективной работы РП-программы очень важно обеспечить передачу информации по сети на фоне основных вычислений везде, где только возможно. Мы будем выделять три основных типа такого обмена информацией:

1. Передача активации, передача и возврат блока параметров. Сюда же относятся запросы на передачу работы и соответствующие ответы.
2. Операции доступа к общей памяти и копирования статических данных. Таковыми являются не только пересылки собственно данных, но также и запросов по их посылке.
3. Передача служебной информации: векторов состояний, а также сообщений об установлении того или иного этапа вычислений.

Сразу отметим, что для последнего типа информации предполагается в целях экономии использование широковещательных сообщений и соответственно "ненадежных" протоколов (IPX, UDP). Как показывает имитационное моделирование, потеря некоторой части таких сообщений практически не сказывается на эффективности вычислений.

Что же касается первых двух типов обмена информацией, то они осуществляются "из точки в точку". Потеря сообщений для них недопустима, а кроме того, довольно часто необходимо еще и сохранить порядок, в котором поступают и обрабатываются сообщения. Поэтому при работе в сети здесь предполагается использование протоколов с гарантированной доставкой (SPX, TCP).

Для обеспечения возможно большего совмещения операций пересылки информации и собственно вычислений требуются как усилия прикладного программиста (см. главу, посвященную приемам повышения эффективности программирования), так и соответствующая поддержка со стороны разработчиков архитектуры RPM и программных средств поддержки рекурсивно-параллельных вычислений.

При аппаратной реализации предусматривалось наличие в составе каждого процессорного модуля специального коммутационного процессора, который должен был выполнять обслуживание всех действий, направленных на взаимодействие ПМ с окружающим миром: формирование, нарезку и сборку пакетов, их посылку, прием и передачу транзитом. В программе имитационного моделирования, входящей в состав среды RPMSHELL, реализован именно такой подход.

При программной реализации средств поддержки РП вычислений работу коммутационного процессора необходимо эмулировать. Естественным решением представляется реализация соответствующих функций в отдельном потоке (thread) вычислений – при работе под Windows. Собственно, имеющаяся в настоящее время DOS-версия использует такой же подход, только организацию потоков и переключений между ними (в той мере, в какой это необходимо для решения данной задачи) пришлось реализовать вручную.

Более точно, необходимо организовать даже два потока: один для приема сообщений и постановки их в очереди на обработку (с высоким приоритетом и минимумом вычислений), и второй – для собственно обработки. Последний поток имеет приоритет ниже, чем поток приема, но

выше, чем поток собственно вычислений. Предусмотрено три очереди сообщений для коммутационного процессора с дисциплиной относительного приоритета между ними:

1. Очередь ответов на любые запросы. Сюда относятся и все служебные сообщения.
2. Очередь запросов извне.
3. Очередь запросов, сформированных на данном ПМ.

Здесь очереди перечислены в порядке уменьшения приоритета на обработку. Следует отметить также, что некоторые события в ПМ, например, постановка в ДЕК, могут вызвать необходимость каких-то действий, связанных с пересылкой информации: отправка компоненты вектора состояний или активации. Поэтому в очередь коммутационного процессора (в последнюю) помещаются также уведомления о таких событиях, а он в процессе обработки выполняет, если надо, соответствующие действия.

Нагрузка на коммутационную сеть

Описанный выше алгоритм динамической балансировки загрузки в сочетании со способом хранения и передачи активаций параллельных процедур (см. описание работы ДЕКа), очевидно, позволяет минимизировать количество передач работы и результатов. Однако для того, чтобы он работал, необходимо иметь на каждом ПМ копию вектора состояний и регулярно пересылать по коммутационной сети информацию об изменении состояния того или иного процессорного модуля. Это, естественно, создает дополнительную нагрузку на сеть, и возникает вопрос, как она велика. Ведь именно нагрузка на коммутационную сеть зачастую является основным фактором недостаточно эффективной параллельной работы программы. Кроме того, поскольку информация о состоянии отдельных ПМ может оказаться устаревшей, время от времени происходят излишние передачи работы, запросы работы у ПМ, которые не могут ее предоставить и т.п. Это также несколько увеличивает загрузку сети.

Оценка влияния этих факторов аналитическим путем достаточно сложна, поэтому для этой цели была разработана имитационная модель и проведен численный эксперимент. Основной целью исследования была оценка числа передач работы, а также количества пересылок компонент вектора состояний Vs в процессе вычислений.

На рис. 6 изображен полученный в результате имитационного моделирования график зависимости количества передач работы и компонент вектора Vs от числа процессорных модулей в системе. Он получен при следующих допущениях: количество листьевых активаций равно 2000, длительность их исполнения случайна и имеет показательное

распределение с математическим ожиданием, в 50 раз больше времени передачи (считается фиксированным). Впрочем, как показало моделирование, изменение этого соотношения даже в 100 раз не приводит к очень значительному изменению количества передач (разница около 20 процентов). Представленные на графике доверительные интервалы соответствуют вероятности 0.99.

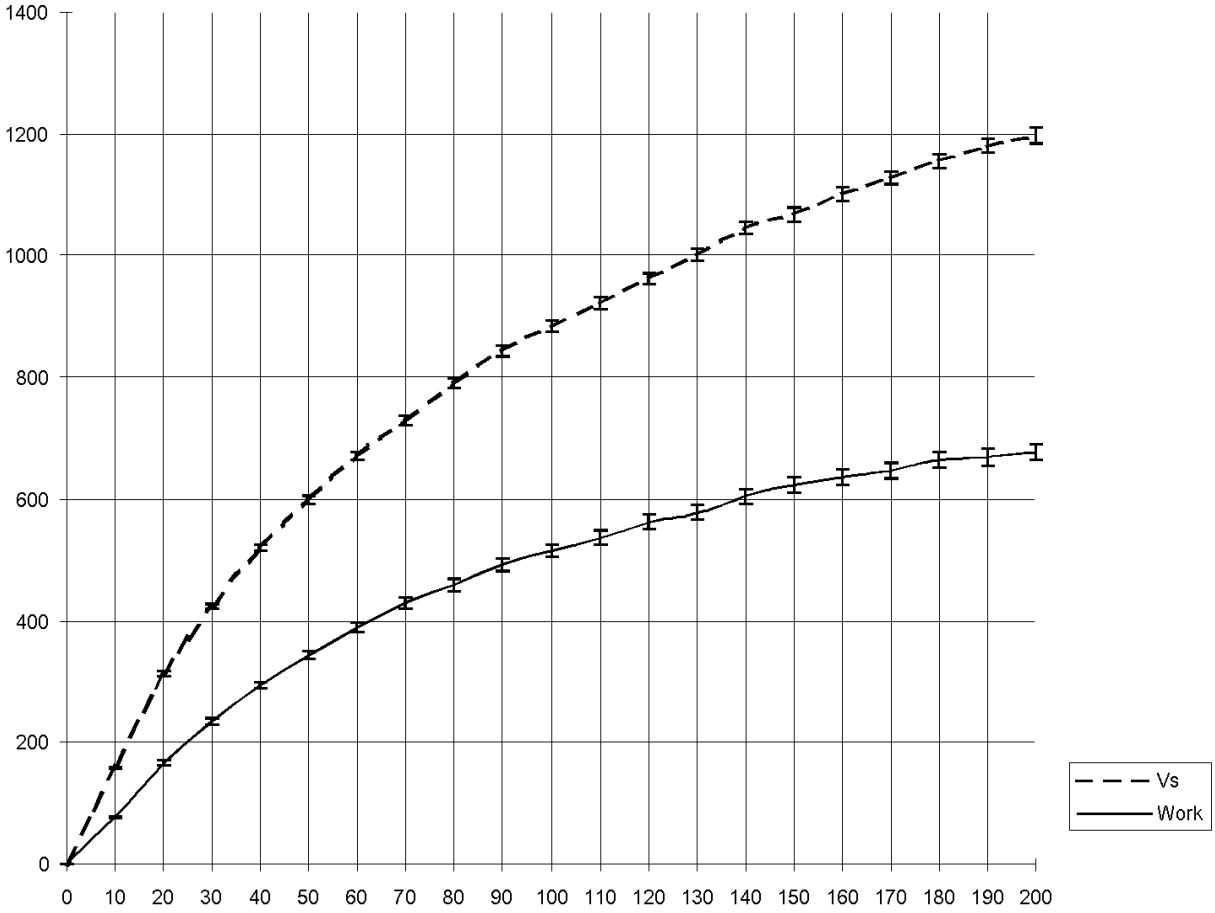


Рис. 6. Количество передач работы (Work) и векторов состояний (Vs) в зависимости от числа процессорных модулей в системе.

Хочется отметить, что показательное распределение было выбрано намеренно, поскольку оно имеет довольно большой коэффициент вариации. Поэтому трудоемкость листьевых активаций различалась весьма существенно. Можно отметить также, что количество активаций (как и процессорных модулей) не было степенью двойки. При этих условиях только качественная динамическая балансировка способна обеспечить достаточно равномерную загрузку системы. Значения параметров MinWork, управляющих "жадностью" ПМ, были взяты минимальными (0 для этапа 1 и 1 для этапа 2). Такой выбор приводит к максимальному количеству передач и наиболее равномерному распределению работы.

Несложно видеть, что необходимое количество передач работ и, следовательно, нагрузка на коммутационную сеть у предлагаемого алгоритма существенно меньше, нежели у алгоритма, использующего

единую очередь заданий (2000), не говоря уже о том, что в системе отсутствует "узкое место". Моделирование показало также, что путем не слишком значительного (менее 10%) замедления можно уменьшить количество передач работы в 1.6 – 1.7 раз, а количество пересылок векторов в 1.4 – 1.5 раз. Для этого достаточно установить значение параметра MinWork на втором этапе, равным 2. Дальнейшее его увеличение, разумеется, еще более снижает нагрузку на коммутационную сеть, однако и потери производительности при этом заметно больше.

Программные средства поддержки рекурсивно-параллельного стиля программирования

Назначение и структура среды RPMSHELL

Одним из основных результатов работы по развитию программных средств поддержки рекурсивно-параллельного программирования явилась разработка интегрированной пользовательской среды RPMSHELL. Она позволяет произвести полный цикл разработки РП-программы.

Структура и основные функции настоящей версии среды рекурсивно-параллельного программирования RPMSHELL представлены на рис. 7.

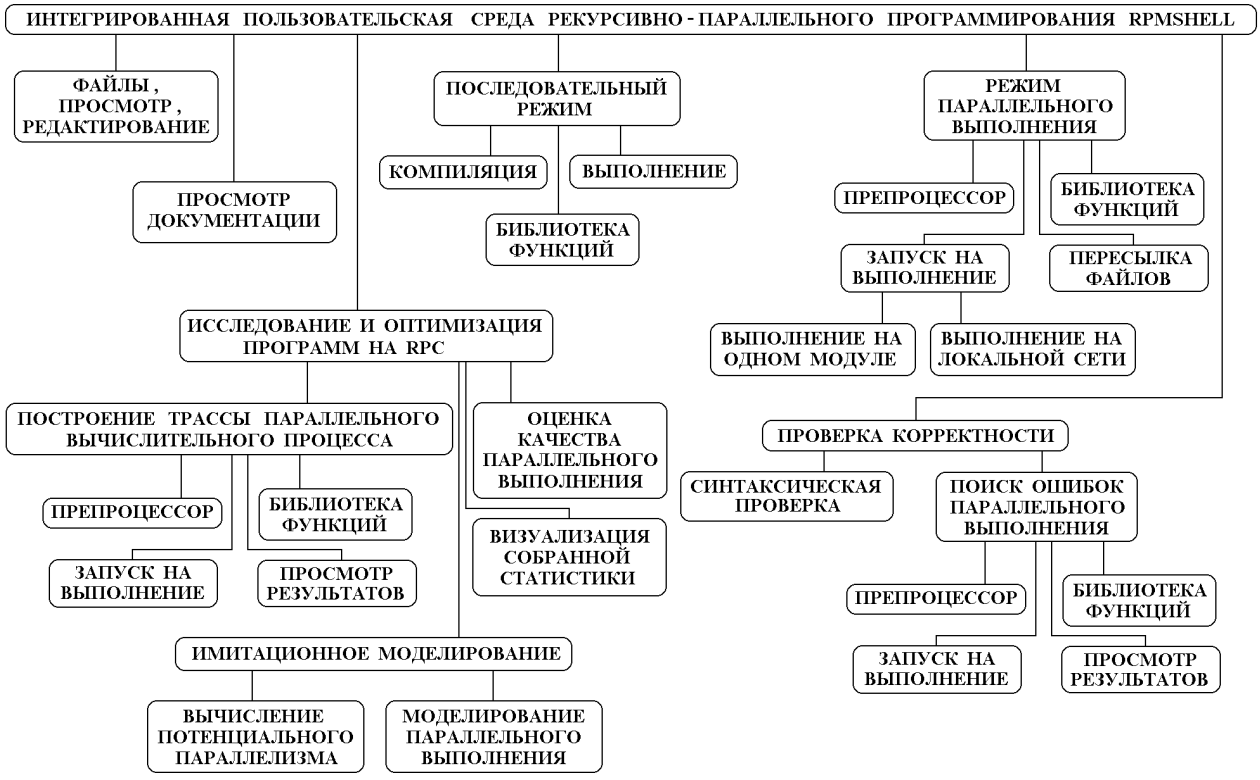


Рис. 7. Структура и основные функции среды рекурсивно-параллельного программирования RPMSHELL.

При разработке программных средств поддержки РП-стиля программирования мы исходили из того, что основными режимами выполнения программы и соответственно основными этапами разработки такой программы являются следующие:

- отладка в режиме последовательного выполнения;
- отладка в режиме последовательного выполнения с выявлением ряда ошибок, которые могут возникнуть при параллельном выполнении;
- последовательное выполнение с одновременным выявлением параллельной структуры программы для исследования параллельных свойств и имитационного моделирования;
- исследование параллельных свойств программы, оптимизация путем подбора параметров, управляющих ходом вычислительного процесса, имитационное моделирование параллельной работы программы на заданной конфигурации RPM;
- параллельное выполнение.

Настоящая версия программной среды RPMSHELL обеспечивает проведение всех этих этапов.

Работа в различных режимах обеспечивается своим набором функций, реализующих операторы языка RPC и образующих библиотеку соответствующего режима выполнения пользовательской программы. Кроме того, для всех режимов работы, за исключением первого, требуется предварительная обработка исходного текста программы с помощью специального препроцессора, который его подготавливает для обработки одним из стандартных компиляторов, а также формирует соответствующий файл-проект и файл для запуска компиляции "в строку". Дальнейшая обработка программы осуществляется с использованием программной среды turboC или BorlandC со всеми их возможностями для отладки программ. Исключение составляет параллельный режим, в котором интерактивная отладка невозможна.

Основные модули оболочки

Среда RPMSHELL представляет собой программную оболочку, предназначенную для создания, отладки, исследования и оптимизации программ рекурсивно-параллельного типа, написанных на языке RPC. Предлагаемое ниже описание ее структуры и работы в ней относится к версии, работающей под управлением MS DOS.

Программно оболочка реализована в виде двух модулей: программ RpmShell.com и Shell.exe, которые предназначены для:

- обеспечения удобного пользовательского интерфейса с программами и библиотеками поддержки рекурсивно-параллельного стиля программирования. Комплекс включает в себя более 20 программ и библиотек, а также ряд конфигурационных и информационных файлов. Оболочка позволяет пользователю не изучать форматы

вызова отдельных программ и принимает на себя все функции по корректной подготовке входной информации для программ комплекса и преобразованию их результатов в вид, удобный для пользовательского восприятия;

- учебно-демонстрационных целей. В демонстрационном режиме оболочка по заданной программе показывает структуру и возможности исследовательского комплекса, а также представляет сопутствующую документацию.

Программа RpmShell представляет собой резидентную часть оболочки. В ее функции входит корректный вызов загружаемой части (программа Shell) и обеспечение ее повторного вызова, если возникает такая необходимость.

Программа Shell – загружаемая (основная) часть оболочки, она обеспечивает выполнение всех перечисленных выше функций. Ее не следует вызывать непосредственно.

Чтобы начать работу,следует вызвать программу RpmShell.com. В качестве аргумента при вызове можно указать имя файла. Этот файл становится "текущим", т.е. при вызове программы просмотра или редактора обрабатываться будет именно он. Разумеется, по ходу работы "текущим" можно сделать любой другой файл (пункт меню Edit). Вызывать программу RpmShell не следует из ее собственной директории, так как текущая директория автоматически становится рабочей и все файлы, создающиеся в процессе работы, сохраняются именно в ней. Впрочем, при попытке сделать это пользователь получит отказ.

Требования к оформлению программ

Перечислим требования к оформлению рекурсивно-параллельных программ, предъявляемые средой:

- программа, написанная на языке RPC, должна быть оформлена в виде одного или нескольких текстовых файлов, имеющих расширение .rpc. В последнем случае предполагается, что файлы подлежат раздельной компиляции и собираются в одну программу при компоновке. Не допускается текст, написанный на RPC, подключать посредством директивы #include, однако это допускается для кусков программы, написанных на стандартном языке C. Последние при этом должны быть оформлены в виде файлов с расширениями .c или .h;
- все .rpc-файлы, относящиеся к данной программе, должна быть собраны в одной директории. Других .rpc-файлов здесь быть не должно, то есть действует правило: одна директория – одна задача. Имя директории при этом считается именем задачи и используется оболочкой для создания имен рабочих файлов.

При формировании файла-проекта для компиляции средой автоматически включаются туда имена необходимых файлов с исходным

текстом программы (полученными из имеющихся в текущей директории файлов с расширением .rpc), а также необходимые для работы библиотеки. Если для работы программы пользователю требуется подключить что-то еще, например, свои библиотеки или объектные коды, следует создать в текущей директории текстовый файл с именем Add.prj и включить туда всю необходимую информацию так, как это делается для файлов-проектов turboC 2.0. Это требование остается в силе и в том случае, если для работы используется другой компилятор (turboC++ или BorlandC).

Пользовательский интерфейс среды RPMSHELL

Оболочка оформлена в виде иерархической системы меню. Выбор тех или иных пунктов осуществляется стандартным образом посредством клавиш Left, Right, Up, Down, Home, Enter, Esc. Если загружен драйвер "мыши", то переместиться в любой из видимых в данный момент на экране пунктов меню можно одинарным щелчком , а вызвать его исполнение – двойным. Щелчок правой кнопки "мыши" эквивалентен нажатию клавиши Esc.

В оболочке за рядом функциональных клавиш закреплены определенные действия. Полный их список приведен в Приложении 1. Здесь мы перечислим только некоторые часто используемые сочетания. F1 вызывает подсказку по действию пункта, на котором в данный момент находится курсор меню, Alt-F1 – подсказку по назначению функциональных клавиш. F10 при работе с меню эквивалентно Esc. Alt-F10 приводит к выходу из оболочки с сохранением ее состояния, Alt-X – к выходу без сохранения, а Alt-O – к временному выходу в DOS путем запуска копии command.com (возвращение, как обычно, через exit).

Внешний вид оболочки показан на рис. 8.

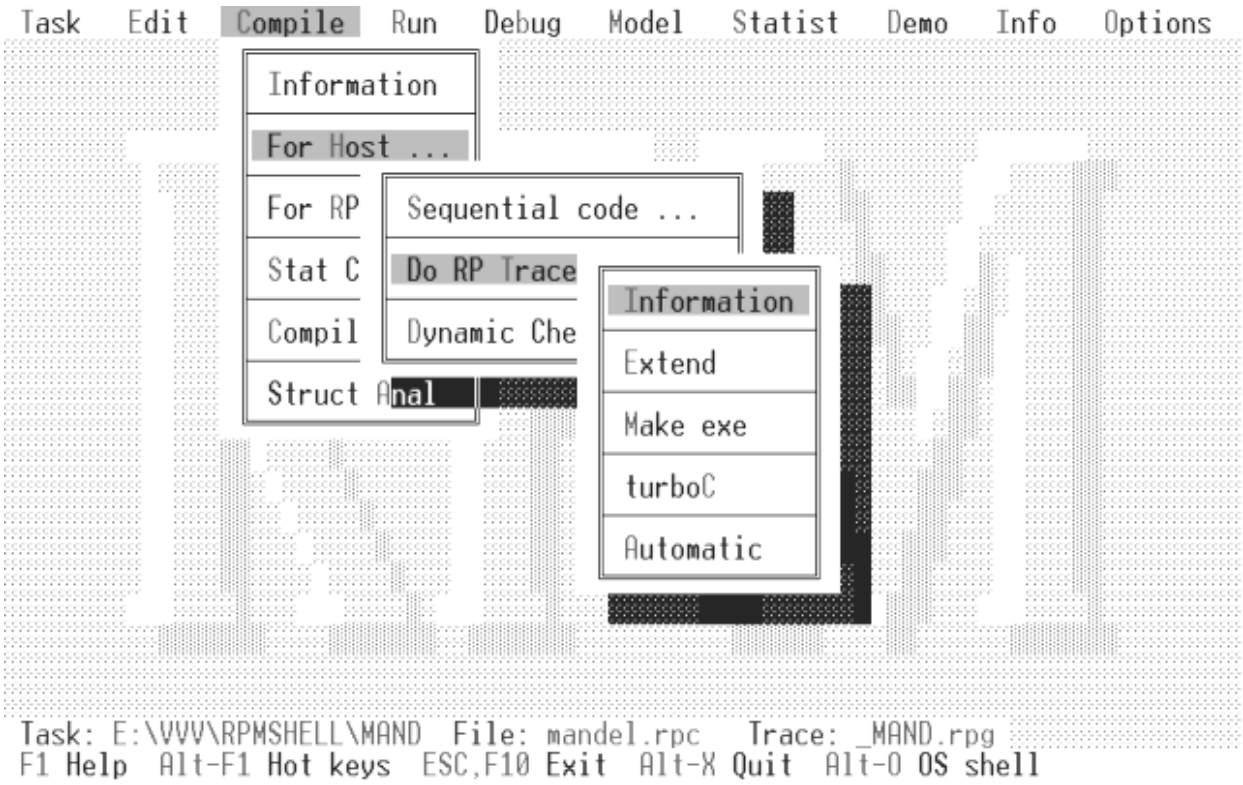


Рис. 8. Оболочка среды RPMSHELL.

Верхняя строка экрана отведена под главное меню, в нижней находится подсказка по действию некоторых функциональных клавиш, а во второй строке снизу помещается информация о имени текущей задачи (директории), имени текущего файла (для просмотра и редактирования) и текущем имени файла трассы.

Вход в среду RPMSHELL происходит путем запуска на выполнение файла rpmshell.com. Можно указать в качестве параметра имя файла, в этом случае оно отображается на второй снизу строке экрана, а при запуске редактора или функции просмотра они берут для работы именно этот файл. При этом, если запуск на выполнение происходит из инструментальной директории RPMSHELL, то последует отказ. Это связано с тем, что при работе появляется много рабочих файлов, присутствие которых в инструментальной директории нежелательно.

Ниже приводится краткое описание назначения пунктов главного меню.

Пункт Task предназначен для перехода в другую директорию или, что то же самое, переключения на другую задачу. Выбор осуществляется посредством обычных манипуляций с меню файлера.

Пункт Edit позволяет просмотреть или отредактировать файл, имя которого выведено на вторую снизу строку экрана или назначить для этой цели другой файл. Последнее назначение осуществляется либо манипуляциями в файлере (пункт Directory), либо явным заданием имени (Select File), либо выбором одного из ранее назначавшихся файлов (Pick).

Программа-редактор в комплект RPMSHELL не входит. Пользователь может выбрать для работы любой из установленных у него текстовых редакторов, воспользовавшись для этого пунктом главного меню Options.

Пункты Compile и Run предназначены соответственно для компиляции и запуска на выполнение программ на языке RPC и подробно рассмотрены ниже.

Пункт Debug в версии, работающей под управлением MS DOS, не используется. В версии, предназначенной для работы под UNIX, он служит для запуска программы-отладчика.

Пункт Model служит для запуска программ имитационного моделирования рекурсивно-параллельного вычислительного процесса, а пункт Statist для обработки результатов моделирования. Подробнее они рассмотрены ниже.

Пункт Demo служит для демонстрации структуры и возможностей оболочки. Он предназначен в основном для демонстрационной версии оболочки, в которой все рабочие программы заменены мультфильмами, демонстрирующими их работу. В рабочей версии запуск этого пункта осуществляет просто экскурсию по оболочке.

Выбрав пункт Info главного меню, пользователь получает доступ к документации по ряду вопросов рекурсивно-параллельного программирования:

- вводная информация (пункт меню Introduction),
- архитектура рекурсивно-параллельной машины (RPM Arcitecture),
- описание языка рекурсивно-параллельного программирования RPC (RPC Language),
- приемы повышения эффективности программирования на RPC (Progamming Methods),
- описание имитационного комплекса (Imitation Complex).

Пункт главного меню Options предназначен для настройки конфигурации оболочки и более подробно описан ниже.

Общие сведения о компиляции и запуске программ в различных режимах

Компиляция и компоновка программ

При работе в среде RPMSHELL в операционной системе MS DOS пользователь может откомпилировать программу, написанную на языке RPC, в одном из четырех режимов:

- для последовательного исполнения;
- для последовательного исполнения с последующей имитацией параллельной работы;
- для последовательного исполнения с поиском ошибок, которые могут возникнуть при параллельном исполнении;

- для параллельного исполнения на локальной сети с использованием протоколов IPX/SPX/

Для выбора режима компиляции из этого списка следует войти в пункт меню соответственно

- Compile / For Host / Sequential code;
- Compile / For Host / Do RP Trace;
- Compile / For Host / Dynamic Checking;
- Compile / For RPM.

Здесь и далее перечень пунктов меню, разделенных косой чертой, означает их последовательный выбор.

При работе под MS DOS для компиляции программ на языке RPC используются обычные в таких случаях компиляторы с языка C фирмы Borland turboC версий 2.0 или 3.0, а также BorlandC версии 3.1. При этом в соответствующем пункте меню можно выбрать либо вызов интерактивной среды выбранной системы программирования, либо компиляцию и сборку программы "в строку". Для выбора того или иного компилятора используется пункт меню Compile / Compiler.

Операторы RPC реализованы либо как макрорасширения, либо как библиотечные функции, написанные на C, необходимо только правильно их использовать при компиляции и компоновке. Для правильной сборки программы нужно, в частности, чтобы в рабочую директорию был скопирован файл Seq.h (при работе под управлением описываемой оболочки это делается автоматически), а при компоновке подключена библиотека поддержки соответствующего режима. Если для создания файла с исполнимым кодом Вы воспользовались соответствующим пунктом меню (а не вызвали turboC либо BorlandC как редактор), то все необходимые действия оболочка сделает сама.

Как было сказано выше, для работы в различных режимах не требуется вносить какие-либо изменения в исходный текст программы. Чтобы обеспечить такую универсальность, перед тем как вызвать собственно компилятор языка C, приходится подвергать текст предварительной обработке. Эту функцию выполняют специальные программы-препроцессоры (разные для различных режимов работы). Исключение составляет режим последовательного исполнения и отладки, для которого не требуется никакой специальной обработки текста. В задачу препроцессора входит также формирование файла-проекта для компоновки с соответствующей библиотекой функций. В случае последовательного режима формирование файла-проекта осуществляет оболочка.

Для компиляции программы в последовательный код (пункт меню Sequential code) можно воспользоваться вызовом компилятора (tcc.exe или bcc.exe) "в строку" (пункт меню Make exe) либо вызвать соответствующую интегрированную среду (пункт turboC либо BorlandC). В последнем случае пользователь имеет возможность выполнить свою программу под

управлением встроенного отладчика. От него потребуется назначить созданный оболочкой файл-проект и установить модель памяти, соответствующую подключаемой библиотеке поддержки (в данной версии это модель памяти Large). Оболочка выдаст соответствующую подсказку на экран. Несоблюдение этих требований приведет в лучшем случае к ошибкам при компоновке, в худшем – к зависанию во время выполнения программы. Для правильного выполнения компиляции "в строку" необходимо также, чтобы был указан правильный путь к INCLUDE- и LIB-директориям (конфигурационный файл в инструментальной директории).

При выборе одного из оставшихся трех режимов работы последовательность обработки должна быть такой. Сначала текст обрабатывается препроцессором (более подробная информация о параметрах вызова содержится в параграфах, посвященных описанию конкретного режима работы). Препроцессор вносит необходимые изменения в исходный текст, а также формирует файл-проект. Затем осуществляется компиляция и сборка "в строку" либо с использованием интегрированной среды компилятора.

Соответствующее меню среды RPMSHELL содержит следующие пункты:

- Extend – для запуска препроцессора;
- Make exe – для запуска созданного на предыдущем этапе bat-файла для компиляции "в строку";
- TurboC или BorlandC – для вызова соответствующей интегрированной среды;
- Automatic – для запуска последовательности Extend, а затем Make exe.

Здесь следует отметить, что при необходимости подключения дополнительных файлов (файл Add.prj), вариант компиляции "в строку" невозможен, поскольку препроцессор не осуществляет обработку этого файла. При попытке пользователя выбрать этот вариант среда сформирует соответствующее предупреждение.

Следует отметить также следующую особенность работы с РП-программами: при рекурсивном вызове процедур увеличиваются требования к размеру стека. Особенно это относится ко второму режиму работы, когда стек активно используется и функциями наработки трассы, а диагностика о переполнении по ряду причин отключена. В результате при недостаточном размере стека программа просто зависает во время выполнения. Поэтому рекомендуется увеличить размер стека до максимально возможного, поместив в начало программы оператор

```
extern unsigned _stklen=65535;
```

Запуск программы на выполнение

Четырем режимам компиляции программы на RPC, перечисленным в предыдущем параграфе, соответствуют четыре возможных режима выполнения:

- последовательное выполнение;
- последовательное выполнение с одновременной наработкой трассы параллельного вычислительного процесса;
- последовательное выполнение с поиском ошибок, которые могут возникнуть при параллельном выполнении;
- для параллельного исполнения на локальной сети с использованием протоколов IPX/SPX.

Перечисленным режимам соответствуют пункты меню

- Run / On Host / Sequentially;
- Run / On Host / Do RP Trace;
- Run / On Host / Dynamic Checking;
- Run / On RPM.

Собственно, выбор первого из перечисленных пунктов меню просто приводит к запуску на выполнение файла с именем <имя задачи>.exe. При запуске в других режимах появляется вложенное меню, позволяющее установить опции запуска и, возможно, выполнить дополнительные действия. Более подробно об этом рассказано в последующих параграфах.

Задать аргументы командной строки (общие для всех режимов) можно в пункте Run / Arguments.

Разработанное программное обеспечение включает в себя некоторые средства отладки, однако они предназначены только для поиска ошибок, которые не могут быть обнаружены средствами отладчиков, включенных в среды turboC и BorlandC. Поскольку любая РП-программа может быть выполнена под управлением упомянутых отладчиков, разработчики посчитали излишним дублировать их работу. Более подробно о типах специфичных для RPC ошибок и средствах для их поиска рассказано в параграфах, посвященных соответствующим режимам работы.

Отладка программ в последовательном режиме

Описанные выше механизмы порождения и распределения параллельных процессов, разумеется, есть смысл реализовывать только при работе в режиме параллельного исполнения. При работе во всех последовательных режимах вызовы параллельных процедур осуществляются самым обычным образом: через запись на стек адреса возврата и передачу управления вызванной процедуре.

Первое, что следует сделать разработчику, – заставить свою программу работать должным образом в последовательном режиме. Процесс компиляции и запуска в этом случае полностью описан в двух предыдущих параграфах. Операторы RPC реализованы либо в виде

макроподстановок (файл seq.h), либо в виде библиотечных функций (seq.lib). Поскольку для компиляции используются среды turboC или BorlandC, специальных средств отладки не требуется – ее можно выполнить с помощью встроенных средств используемой системы программирования.

Имена исполнимых файлов формируются оболочкой по следующему правилу: для режима последовательного выполнения файл получает имя <имя задачи>.exe, а для режима с наработкой трассы _<имя задачи>.exe. Напомним, что именем задачи считается имя текущей директории.

Средства исследования программ путем имитационного моделирования

Компиляция и запуск на выполнение

Для того, чтобы исследовать параллельные свойства программы на RPC, имея в распоряжении последовательный компьютер, и в частности смоделировать ее параллельное выполнение применяется следующий подход. Программа компилируется в режиме, при котором в нее добавляются функции, позволяющие в процессе ее последовательного исполнения выявить структуру соответствующего параллельного вычислительного процесса, которая в виде графа специального вида запоминается в файле (файл графа трассы). Эта информация в дальнейшем служит в качестве исходной для программы имитационного моделирования и других программ для изучения параллельных свойств Вашей рекурсивно-параллельной программы.

Последовательность действий, реализующая упомянутый подход, следующая. Исходный текст программы на RPC обрабатывается программой Extend.exe (пункт меню Do RP Trace / Extend). В результате ее работы образуются файлы, содержащие расширенный исходный текст (расширение .rpe). Кроме того, для последующего вызова tcc.exe или tc.exe создаются .bat- и .prj-файлы. Вызов упомянутых компиляторов осуществляется выбором пунктов меню Do RP Trace /Make exe или Do RP Trace / turboC соответственно. При этом следует иметь в виду все моменты, на которые мы обращали Ваше внимание при описании режима компиляции в последовательный код. Кроме этого, при работе с tc.exe следует установить для операций с плавающей точкой режим эмуляции (это нужно для повышения достоверности результатов моделирования).

Если Вы запускаете программу на выполнение из-под turboC, не следует прерывать ее выполнение через Ctrl-F2, Alt-X или тому подобным образом – это может привести к ошибкам вплоть до зависания системы. Дело в том, что функциям наработки трассы для своей работы приходится изменять работу таймера, перехватывать некоторые прерывания и т.п. Естественно, перед завершением программы необходимо дать им возможность восстановить исходное состояние системы. Поэтому, если

появилась необходимость прервать выполнение программы, нажмите Ctrl-Q, и завершение работы произойдет корректно.

При запуске программы в режиме с наработкой трассы сначала происходит вычисление параметров, необходимых для построения трассы при работе на данном компьютере (при этом на экран выводятся некоторые служебные сообщения), и лишь затем происходит выполнение программы пользователя. Если выполнение завершилось успешно, то информация о настройке сохраняется в файле Default.cnf, и при последующих запусках повторная настройка не производится.

Перед запуском на выполнение программы в режиме с наработкой трассы пользователь имеет возможность задать/изменить следующие опции (пункт меню Do RP Trace / Options):

- имя файла трассы. Впрочем, такая необходимость возникает только в том случае, если Вы хотите использовать в дальнейшем для моделирования несколько графов одновременно;
- опция Control. Установка в состояние On этой опции позволяет отловить во время выполнения программы некоторые ошибки, например, выход из активации параллельной процедуры при незавершенных дочерних параллельных процессах. Кроме того, во время работы сохраняется ретроактивный след из 20 последних пройденных контрольных точек, о которых будет сказано чуть ниже. В случае обнаружения ошибки этот след выводится на экран, что позволяет локализовать ошибку;
- опция Tracing. Установка этой опции (при включенной опции Control) позволяет в процессе работы программы постоянно иметь на экране след контрольных точек, что позволяет локализовать ошибку даже в том случае, если в процессе работы произошло зависание, так как номера последних пройденных точек сохраняются на экране. Кроме того, в верхнем правом углу экрана выводится информация о размере построенного графа трассы (в байтах).

Контрольные точки в текст программы вставляет программа Extend.exe во всех местах, которые предшествуют каким-либо вычислениям. При этом считается, что вычисления прерываются операторами вызова параллельных процедур, доступа в общую память, оператором ожидания Wait() и т.п. Обнаружить их можно в расширенном тексте программы (файлы с расширением .rpe) в виде v_NLK(a, b). Здесь b – номер контрольной точки.

Граф трассы

Граф трассы параллельного вычислительного процесса, порождаемого программой, при обработке конкретных данных содержит всю информацию, необходимую для исследования параллельных свойств программы и имитационного моделирования ее выполнения на RPM заданной конфигурации. Он отражает свойства только вычислительного

процесса и никак не привязан к деталям архитектуры RPM. Это свойство позволяет, однажды построив граф трассы, исследовать на имитационной модели поведение программы при ее исполнении на различных конфигурациях вычислительной системы.

Вершины графа трассы соответствуют запуску тех или иных процессов при выполнении РП-программы, а дуги – информационным зависимостям. Процесс может начаться только после того, как завершились все процессы, ему предшествовавшие. Граф содержит всю необходимую информацию для последующего моделирования. Так, вершины, соответствующие процессам доступа в память, содержат сведения о типе операции и ее параметрах, вершины, соответствующие вычислениям – информацию об их длительности, и т.д.

О последних следует сказать особо. Разумеется, для более качественного моделирования полезно было бы знать, как долго те или иные участки вычислений будут выполняться именно на интересующей нас системе. Однако при сборе такой информации в процессе вычислений на другом компьютере неизбежны какие-то упрощающие допущения и, как следствие, погрешности. В описываемом комплексе сбор информации о длительностях тех или иных участков вычислений осуществляется путем считывания информации из портов системного таймера и пересчет ее с некоторым коэффициентом (вычисляется в процессе настройки) в специальные временные единицы. Эти единицы соответствуют тактам вычислительной системы, разрабатывавшейся в свое время в ИПВТ РАН, поскольку первоначально система моделирования создавалась именно для потребностей этой разработки. Однако, очевидно, что выбор единицы измерения времени в данном случае принципиального значения не имеет. Кроме того, он может быть изменен как при моделировании путем задания соответствующего параметра в файле конфигурации Default.cnf, так и в процессе моделирования.

Визуальное представление графа трассы обеспечивается программой Vis_tr.exe, которую можно вызвать, выбрав пункт меню Model / Ideal / View Trace Graph или Model / Model1 / View Trace Graph. Внешний вид программы представлен на рис. 9.

Здесь, разумеется, граф трассы изображен не полностью, поскольку обычно этого не позволяют размеры экрана. Программа позволяет двигаться по графу, выводя на экран те или иные его куски, изменять масштаб изображения, а также выводить на экран подробную информацию о текущей вершине. Исследование структуры графа трассы помогает обнаружить недостатки организации вычислительного процесса, например, наличие длинных участков последовательных вычислений или те участки алгоритма, где процесс развивался не так, как это было задумано разработчиком программы.

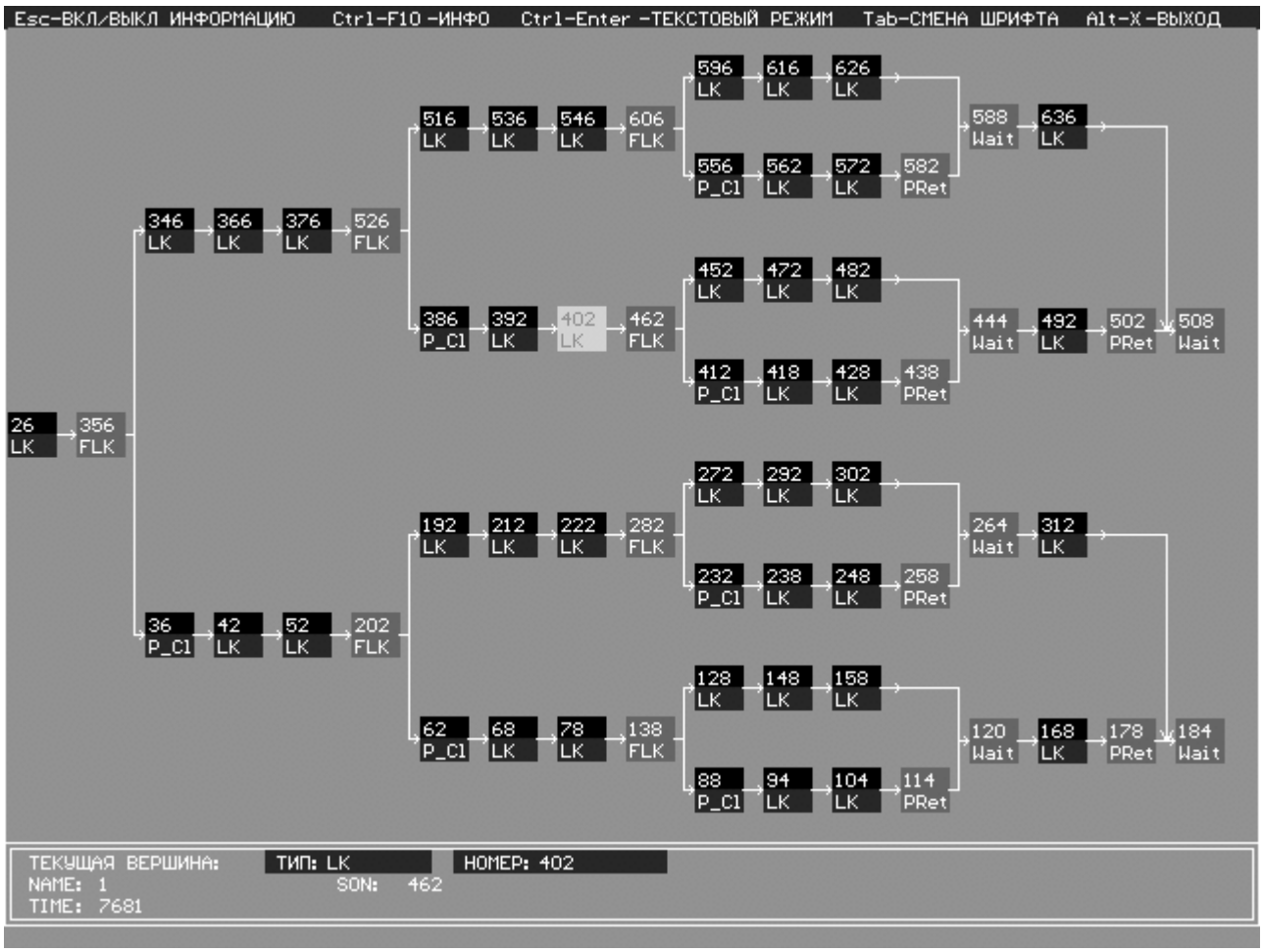


Рис. 9. Программа визуализации графа трассы.

Но основное назначение графа заключается в том, что он является источником входной информации для программ имитационного моделирования, которые описываются ниже. В настоящее время комплекс RPMSHELL включает в себя следующие две программы имитационного моделирования:

- Программа для исследования потенциального параллелизма программы и характеристик РП-трассы Testgr.exe. Из среды ее проще всего вызвать, войдя в пункт меню Model / Ideal. Там же в диалоге можно задать или изменить предлагаемые по умолчанию значения параметров запуска: имя файла, содержащего граф трассы, собираемую статистику и т.п.
- Программа Mod64.exe. Она предназначена для имитационного моделирования выполнения РП-программы на упомянутой выше параллельной вычислительной системе, разрабатывавшейся в ИПВТ РАН. Для вызова и настройки параметров в среде RPMSHELL удобнее всего воспользоваться пунктом меню Model / Model1. Более подробно работа с программой описана ниже.

При необходимости произвести моделирование процесса параллельных вычислений на системе, архитектура которой радикально

отличается от той, которая заложена в программу Mod64, придется разрабатывать другую программу моделирования, однако, как представляется автору, это вполне разрешимая задача, тем более, что построенный граф трассы содержит в себе всю необходимую информацию о структуре вычислительного процесса.

Исследование потенциального параллелизма РП-программы

Выбор пункта меню Model / Ideal / Run приводит к запуску программы Testgr.exe, которая предназначена для вычисления некоторых собственных характеристик РП-процесса (потенциальный параллелизм, максимальный параллелизм и т.п.), проверки корректности построения графа РП-трассы, а также сбора некоторой статистической информации. Последняя в основном предназначена для того, чтобы пользователь с помощью программы-визуализатора мог получить представление о динамике параллелизма своей программы.

Собственно программа Testgr является имитационной моделью идеализированной параллельной вычислительной системы, построенной в соответствии с концепцией неограниченного параллелизма, то есть основана на следующих допущениях:

- количество процессорных модулей в системе не ограничено;
- при передаче любых данных отсутствуют конфликты, а сама передача происходит мгновенно;
- длительность всех процессов, кроме собственно вычислений, равна нулю.

Результатом работы программы Testgr.exe является протокол работы и, если требуется, файл статистики. Структура этих файлов описана ниже. Файл протокола содержит следующие характеристики РП-процесса:

- длина критического пути в графе трассы. Эту величину можно интерпретировать как время выполнения программы на гипотетической рекурсивно-параллельной системе с неограниченным числом ПМ и нулевыми накладными расходами на организацию вычислительного процесса (сюда входит время на передачу вызовов параллельных процедур, их параметров, результатов, время доступа в РОП и т.п.). Длина критического пути дает нижнюю оценку времени работы программы: более быстрого выполнения добиться в принципе невозможно;
- суммарный объем вычислений в программе. Эту величину можно также интерпретировать как время последовательного выполнения программы на однопроцессорной ВС с нулевыми накладными расходами на организацию вычислительного процесса;
- средний потенциальный параллелизм. Эта величина определяется как отношение суммарного объема вычислений к длине критического пути.

Она служит верхней оценкой ускорения, которое может быть достигнуто на параллельной вычислительной системе при выполнении данной программы;
- максимальная ширина РП-процесса;
- количество операций доступа в РОП и их удельный вес, т.е. средний объем вычислений, приходящийся на одну операцию доступа в РОП.

Параметром программы Testgr является имя файла конфигурации. При отсутствии параметра считается, что файл конфигурации имеет имя Default.cnt. Если файл конфигурации отсутствует или содержит недостаточно информации, значения недостающих параметров выясняются в диалоге с пользователем. Файл конфигурации представляет собой текстовый файл, каждая строка которого задает значение одного параметра и имеет вид:

<Имя параметра> = <Значение параметра>

Строки, содержащие ошибочное имя параметра, игнорируются. Программе необходимо задать следующие параметры:
- Graph - имя файла трассы. Допускается указать несколько имен через запятую. В этом случае будет считаться, что соответствующие задачи запущены в режиме мультипрограммирования (параллельно);
- Stat - имя файла статистики;
- Prot - имя файла протокола.

Необязательными параметрами являются следующие:
- KeyStat шестнадцатеричное целое. i-й бит равен 1, если требуется сбор статистики по i-му датчику (см. описание файла статистики). При отсутствии параметра статистика не собирается;
- _vert, _send - внутренние переменные программы, задающие максимально возможное количество одновременно обрабатывающихся вершин графа, и максимальный номер процессорного модуля, к которому параллельные активации процедур направляются оператором P_Send. Эти параметры есть смысл задавать только в том случае, если сама программа Testgr.exe диагностирует такую необходимость.

Отметим здесь, что оболочка RPMSHELL позволяет пользователю задавать в диалоге (пункт меню Ideal / Configuration) только параметры Graph, Stat, Prot, KeyStat. При необходимости задания или изменения других параметров следует с помощью текстового редактора внести соответствующие добавления или изменения в файл конфигурации (обычно Default.cnt).

Входной информацией для программы Testgr является граф трассы РП-процесса, построенный в соответствии с соглашениями, описанными в документации по построению графа трассы.

Программа предоставляет пользователю следующие возможности:
- Наблюдение за текущим состоянием вычисляемых характеристик графа РП-трассы, перечисленных в пункте "Результат";

- Частичное или полное отключение вывода информации на экран с целью ускорить работу программы;
- Пошаговое выполнение процесса анализа трассы;
- Остановку анализа трассы в заданный момент;
- Установку задержки для замедления процесса анализа трассы;
- Вывод в любой момент более подробной информации о состоянии (отладочный вывод).

Перечисленные возможности реализуются путем нажатия соответствующих клавиш, подсказка всегда находится на экране.

Ниже приводится описание файла статистики, формируемого программой Testgr. Он используется в качестве входной информации для программы Ideal, которая должна представить собранную информацию в графическом виде. В результате пользователь может получить представление о динамике параллелизма своей программы, емкости активаций параллельных процедур и т.п. При работе в среде RPMSHELL вызов программы визуализации статистики следует непосредственно за вызовом Testgr.

В начале файла статистики находится поле длиной 4 бт., которое содержит информацию о времени выполнения задачи на идеальной модели RPM (длина критического пути).

Далее последовательно идут записи длиной 5 бт., содержащие информацию о значении встроенных в модель датчиков. Запись имеет следующую структуру:

- номер датчика (1 бт.),
- время (4 бт.).

Если датчик может принимать значение 0 и 1, то происходит установка в соответствующее значение старшего бита поля 'датчик'.

Датчик	Описание датчика
0	В поле "время" содержится момент появления очередной операции Load/Store
1	Принимает значение 1 при увеличении текущей ширины вычислительного процесса, 0 – при уменьшении
2	Собственный объем активаций параллельных процедур. В поле "время" содержится объем очередной активации. Если в нем содержится 0, то это указание на начало очередного гамака

Результаты работы программы Testgr записываются в файл протокола. Последний представляет собой текстовый файл, в котором содержится информация о характеристиках, вычисленных в результате анализа РП-трассы. При повторном запуске программы Testgr файл протокола не

создается заново, а дописывается. В этом случае результаты различных экспериментов разделяются строкой имеющей вид
########## Testgr.exe ##########

Запись имеет вид
<Обозначение характеристики> = <Значение характеристики>

В начале файла записывается информация о применяемых обозначениях характеристик:
Ts - суммарное время вычислений
Tk - время критического пути
Pmax - максимальный параллелизм
Sa - средний параллелизм
Pg - количество порожденных параллельных процессов
Vp - объем вычислений, приведенный к Pg
Nls - количество операций доступа к разделяемой памяти
Vls - объем вычислений, приведенный к Nls

Программа визуализации собранной статистики Ideal.exe позволяет вывести на экран в виде графиков и таблиц следующую информацию о характере параллельного вычислительного процесса, представленного РП-трассой:
- параметры РП-трассы;
- профиль параллелизма программы;
- распределение активаций параллельных программ по длительности;
- распределение во времени операций доступа в РОП.

При входе в программу Ideal пользователь попадает в главное меню программы. Он может либо завершить сеанс работы, нажав клавишу Esc, либо обычным образом выбрать одну из перечисленных выше функций программы.

При выборе первого из перечисленных пунктов на экран будет выведена сводная таблица параметров РП-трассы, вычисленных программой Testgr. При этом следует иметь в виду, что эта информация берется из файла протокола, а не из файла статистики, и следовательно, последние записи в файле протокола и статистика в файле статистики должны относиться к одному и тому же графу трассы. Это всегда так, если пользователь не изменял файл протокола самостоятельно.

Пункт главного меню "профиль параллелизма" (рис. 10) позволяет вывести график, показывающий динамику изменения ширины параллельного вычислительного процесса во времени. Исходя из его вида, пользователь может сделать выводы о том, на каком участке работы программы параллелизм достаточен, а на каком недостаточен для загрузки всех процессорных модулей системы. При этом следует помнить, что эта статистика была получена на идеализированной модели с нулевыми накладными расходами, и высокий средний параллелизм сам по себе еще не гарантирует эффективную работу РП-программы.

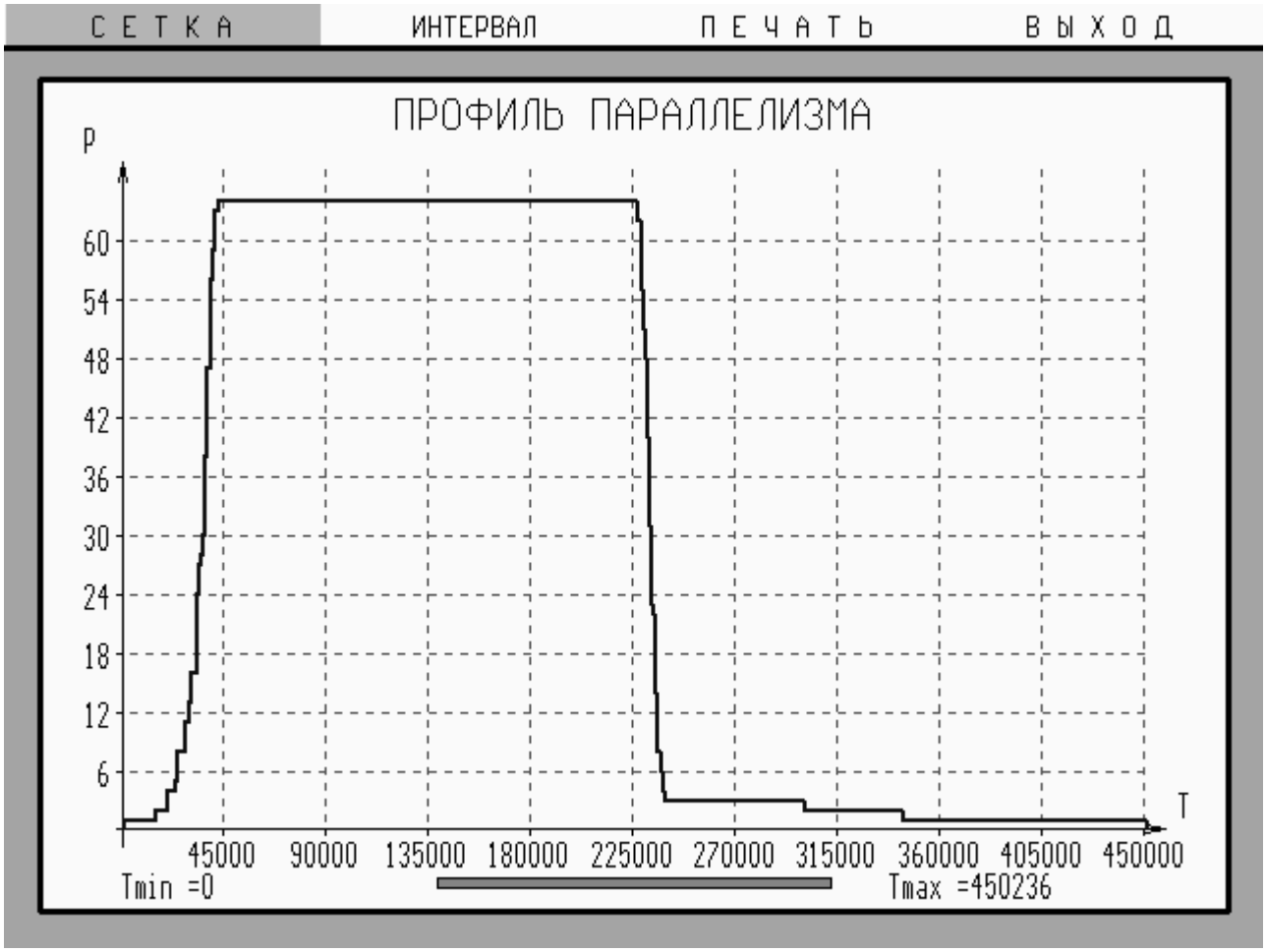


Рис. 10. Профиль параллелизма простейшей рекурсивно-параллельной программы.

Пункт "распределение активаций" позволяет представить гистограмму емкости активаций параллельных процедур. Это позволяет сделать выводы о степени равномерности распределения работы по листьевым активациям и, следовательно, о принципиальной возможности ее равномерного распределения по процессорным модулям.

Пункт "операции доступа в общую память" позволяет увидеть, в какие моменты времени и сколько раз происходили операции обращения в ОП. Слишком большое количество одновременных обращений в общую память приводит к перегрузке коммутационной сети и как следствие – к недостаточно эффективной работе РП-программы.

При работе в том или ином пункте меню пользователю предоставляется возможность регулировать сетку графика путем задания ее шага, отмены или включения, задания конкретного временного интервала, в котором требуется вывести график, вывода графиков и таблиц с экрана на печатающее устройство. В двух последних пунктах главного меню имеется возможность изменения шага гистограммы путем задания другого количества точек (пункт меню "точки"), на которое разбивается

весь анализируемый временной отрезок (по умолчанию количество точек разбиения равно 10)

Имитационная модель RPM

Программа Mod64.exe предназначена для имитационного моделирования процесса выполнения РП-программы на RPM заданной конфигурации и сбора статистики, позволяющей оценить качество параллельной программы и дать рекомендации по ее улучшению (это делает специальная программа-анализатор). Для имитационного моделирования РП-программа должна быть представлена своим графом трассы, который и служит исходным материалом для моделирования.

Относительно архитектуры коммутационной среды и отдельного процессорного модуля в описываемой модели делаются следующие основные допущения. Коммутационная сеть представляет собой квадратную решетку, свернутую в тор, в узлах решетки располагаются процессорные модули. Передаваемая информация нарезается на пакеты фиксированной длины, которые передаются по сети независимо, конфликты в промежуточных узлах сети отсутствуют, время передачи зависит только от расстояния и размера пакета.

В процессорном модуле имеется три параллельно работающих устройства:

- Арифметический процессор (АП). Его назначение – выполнение собственно вычислений.
- Управляющий процессор (УП). В его обязанности входит обработка запросов от АП о необходимости выполнения тех или иных системных действий, сообщений об остановке, реализация механизмов порождения и исполнения активаций параллельных процедур, принятие решений о передаче или запросе работы, активизации процедур – в общем всех действий, управляющих ходом вычислительного процесса. Все запросы к УП со стороны других делятся на две группы в зависимости от требуемой срочности исполнения и имеются две очереди запросов, обрабатываемых в соответствии с дисциплиной относительного приоритета.
- Коммутационный процессор (КП). Это устройство обеспечивает нарезку на пакеты, отправку и прием всех данных и служебной информации, предназначенных для передачи по коммутационной сети. Все запросы к КП делятся на три группы приоритета. В первую очередь обрабатываются ответы на сформированные данным ПМ запросы, затем запросы на информацию или данные со стороны других ПМ, в последнюю очередь обрабатываются собственные запросы данного процессорного модуля.

Результатом работы программы Mod64 является протокол работы модели, а также файл статистики, собранной в ходе работы модели. Эта

информация используется как входная для программы визуализации статистики, а также программы-анализатора эффективности РП-программ. Их описание приведено ниже.

Параметром программы Mod64 является имя файла конфигурации. При отсутствии параметра считается, что файл конфигурации имеет имя Default.cnm. Если файл конфигурации отсутствует или содержит недостаточно информации, значения недостающих параметров выясняются в диалоге с пользователем. Файл конфигурации представляет собой текстовый файл, каждая строка которого задает значение одного параметра и имеет вид:

<Имя параметра> = <Значение параметра>

Строки, содержащие ошибочное имя параметра, игнорируются. Программе необходимо задать следующие параметры:

- Graph – имя файла трассы. Допускается указать несколько имен через запятую. В этом случае будет считаться, что соответствующие задачи запущены в режиме мультипрограммирования (параллельно);
- Kolpm – количество процессорных модулей в системе;
- Zapas – минимальное количество вызовов в деке, которое модуль оставляет себе;
- Mnseg – количество готовых для выполнения на АП сегментов программы, при котором выключается механизм опережающей подкачки из дека и активизации процедур;
- Stat – имя файла статистики;
- Prot – имя файла протокола.

Необязательными параметрами являются следующие:

- KeyStat – шестнадцатеричное целое. i-й бит равен 1, если требуется сбор статистики по i-му датчику (см. описание файла статистики). При отсутствии параметра статистика не собирается;
- Monopoly – разрешение/запрещение (On/Off) монопольного захвата процессорного модуля;
- _up0, _up1, _kp0, _kp1, _kp2, _deq, _pak – параметры, задающие отличную от стандартной длину очереди управляющего или коммутационного процессора, размер дека, длину очереди пакетов. Последняя не соответствует никакому физическому устройству – это внутренний параметр модели.

Эти параметры имеет смысл задавать только в том случае, если в ходе моделирования выясняется, что выбранный по умолчанию размер недостаточен, и происходит переполнение.

Программа Mod64 предоставляет пользователю следующие возможности:

- Выбор одного из двух режимов вывода на экран информации о текущем состоянии модели, а именно: в виде гистограммы загрузки арифметических процессоров (рис. 11) или в виде таблицы состояния отдельных устройств: АП, УП, КП, деки и пр.

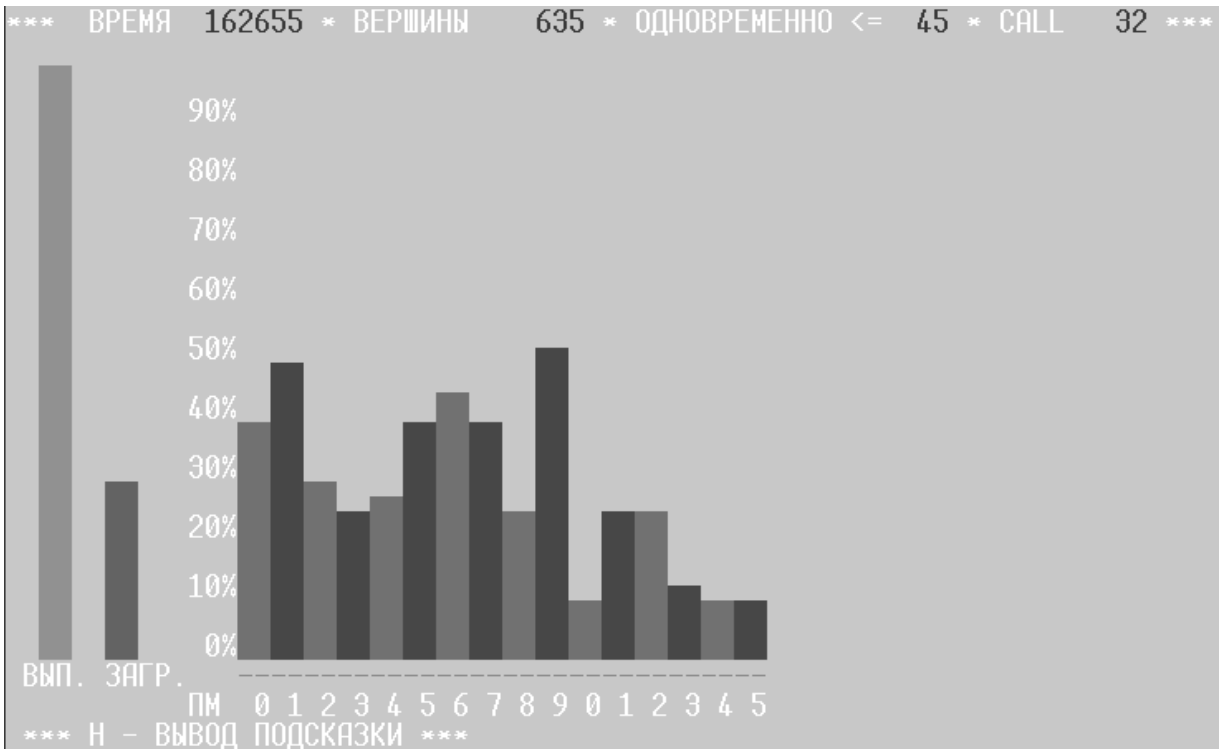


Рис. 11. Программа моделирования Mod64.

- Пошаговое выполнение процесса моделирования;
- Остановку моделирования в заданный момент;
- Установку задержки для замедления процесса моделирования;
- Вывод в любой момент более подробной информации о состоянии модели (отладочный вывод);

Возможность интерпретации больших графов трассы по частям (режим Auto). Отдельные части при этом должны быть законченными гамаками.

Выбор той или иной возможности осуществляется нажатием соответствующих клавиш. Подсказка по их назначению вызывается нажатием клавиши 'h'.

Файл статистики, формируемый программой Mod64, предназначен для обработки программой-анализатором, описанной в пункте "Анализ причин снижения эффективности", и содержит информацию о моментах начала и завершения основных процессов в имитационной модели RPM. Он имеет следующую структуру.

В начале файла находятся 2 поля, содержащих общую информацию:

- количество процессорных модулей (1 бт.),
- время выполнения задачи (время моделирования) (4 бт.)

Далее последовательно идут записи длиной 6 бт., содержащие информацию о значении встроенных в модель датчиков. Запись имеет следующую структуру:

- номер датчика (1 бт.),
- номер ПМ (1 бт.) (используется не для всех датчиков),

- время (4 бт.). Если датчик может принимать значение 0 и 1, то происходит установка в соответствующее значение старшего бита поля 'датчик'. Содержание датчиков ясно из приводимой ниже таблицы.

Номер	Описание датчика
0	Принимает значение 1 при начале рабочего цикла АП, 0 – при завершении
1	Принимает значение 1 при начале рабочего цикла ЦП, (вся работа за исключением обменных операций), 0 – при завершении
2	1 – при выполнении в АП арифметических операций (LK), 0 – при их завершении
3	0 – при нахождении ПМ в состоянии безработицы, 1 – в противном случае
4	1 – при увеличении в модуле числа совмещенных процессов (под ними понимаются готовые и отложенные по Load/Store процессы), 0 – при завершении
5	1 – в начале простоя АП по причине обменных операций с общей памятью, 0 – при завершении
6	1 – при увеличении в системе количества работ (т.е. активаций, находящихся в деках, активном состоянии или отложенных по Load/Store), 0 – при уменьшении Поле "Номер ПМ" не используется
7	Фиктивный датчик для сокращения объема файла статистики. Установка его в 1 или 0 означает таковую установку одновременно для датчиков 4 и 6. Поле "Номер ПМ" имеет смысл при этом только для датчика 4
8	Статистика по собственному объему активаций процедур. Собирается перед началом работы модели и записывается в начале файла статистики отдельным куском. В поле "Время" содержится объем очередной активации. 0 в поле "Время" означает начало следующего гамака. В поле "Номер ПМ" записывается 0. Если этот датчик используется для указания на начало очередного гамака в процессе сбора прочей статистики, в поле "Номер ПМ" записывается 1, а в поле "Время" – соответствующий момент времени
9	1 – при увеличении количества не активизированных работ в модуле (CALL + SEND) 0 – при уменьшении

Статистическая информация, собираемая в ходе имитационного моделирования.

Файл протокола, формируемый программой имитационного моделирования Mod64, – это текстовый файл, в котором содержится информация о результатах моделирования. При повторном запуске программы Mod64 файл протокола не создается заново, а дописывается. В этом случае результаты различных экспериментов разделяются строкой имеющей вид

```
########## Mod64.exe ##########
```

Запись имеет вид

<Обозначение характеристики> = <Значение характеристики>

В начале файла записывается информация о применяемых обозначениях характеристик:

TF - имя файла трассы
N - количество процессорных модулей
T - реальное время вычислений
Pg - количество порожденных параллельных процессов
Upm - средняя загрузка процессорных модулей (в %)
Ucp - средняя полезная загрузка центральных процессоров (в %)
M - количество миграций параллельных процессов

Утилиты обработки результатов имитационного моделирования

Визуализатор статистики

Собранную в ходе имитационного моделирования статистическую информацию можно представить в удобном для исследования виде с помощью программы Visual.exe. Формат ее вызова:

```
Visual.exe <••• ••••• ••••••••••>
```

При работе в среде RPMSHELL для этой цели удобнее воспользоваться пунктом меню Statist / Visualisation.

Программа Visual позволяет представить в качестве графиков и гистограмм по отдельным модулям, а также по системе в целом статистику:

- по работе арифметических устройств системы;
- по работе центральных процессоров системы;
- по выполнению чисто счетной работы (арифметические операции);
- по наличию работы в модулях в каждый момент времени;
- по простоям арифметических устройств из-за ожидания завершения операций доступа в общую память;
- по длительности отдельных активаций параллельных процедур;
- по ширине параллельного процесса в каждый момент времени.

Анализ этой информации может помочь пользователю понять причины непроизводительных простоев отдельных устройств (главным образом арифметических процессоров), а следовательно, недостаточно эффективной работы программы, а также выработать способ их устранения, если, конечно, это возможно.

При входе в программу Visual (рис. 12) пользователь попадает в главное меню программы. Он может либо завершить сеанс работы, нажав клавишу Esc, либо обычным образом выбрать одну из следующих функций программы:

- профиль по датчику;
- диаграмма по модулям;
- таблица датчиков;
- диаграмма по датчикам;
- распределение активаций;
- совмещенные процессы и работы.

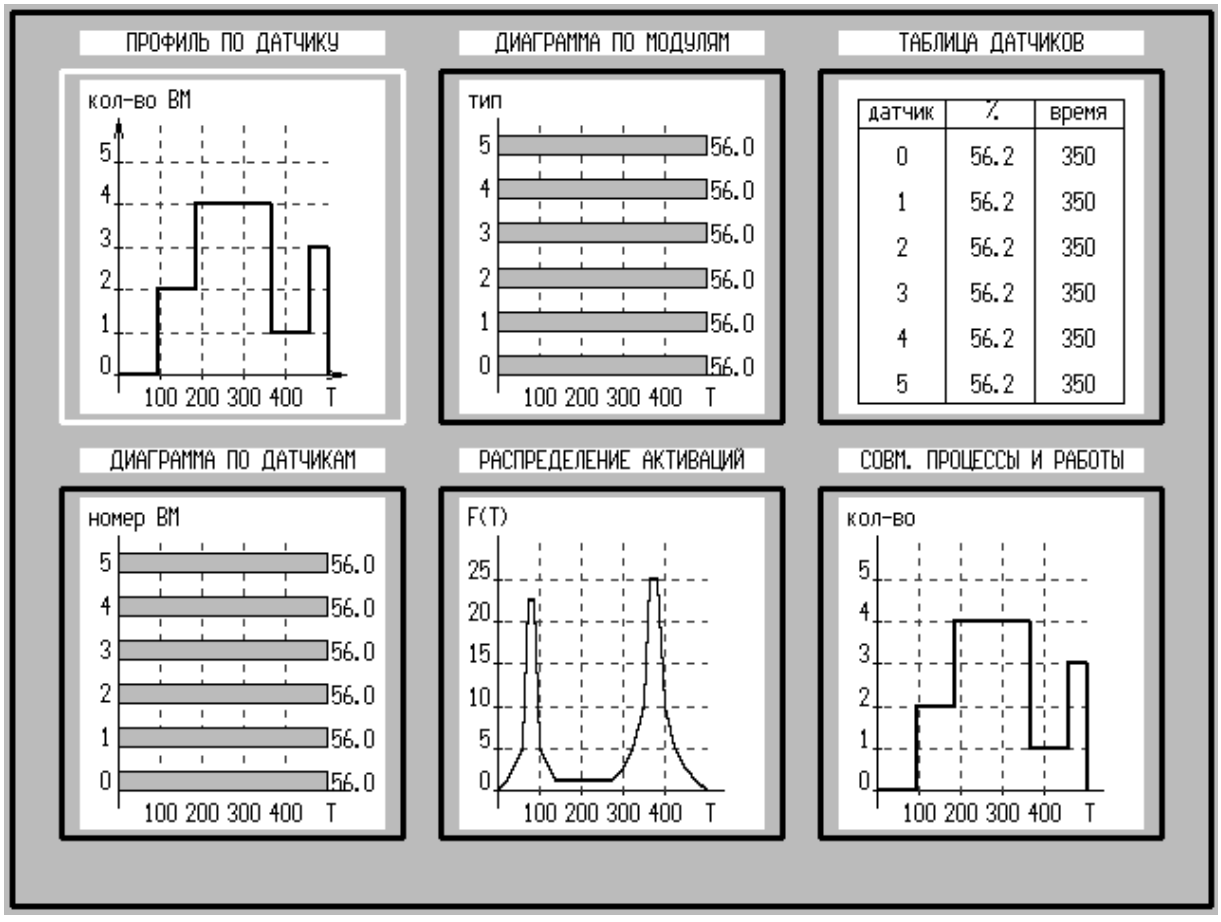


Рис. 12. Программа визуализации результатов моделирования.

Первые три пункта главного меню предназначены для вывода в различном виде информации о рабочем цикле арифметических процессоров, центральных процессоров, выполнении чисто счетной

работы, наличии работы в процессорных модулях, простоях из-за ожидания завершения операций доступа в общую память.

При этом в первом пункте ("профиль по датчику") соответствующая информация представлена в виде графика, демонстрирующего динамику изменения значения соответствующего датчика с течением времени (в целом по системе). Пользователь может видеть, как изменялось с течением времени количество процессорных модулей, имевших работу, или, например, простаивающих из-за ожидания завершения операций доступа в общую память.

Во втором пункте ("диаграмма по модулям") эта же информация представлена для каждого отдельного модуля в виде диаграммы. Третий пункт ("таблица датчиков") в виде таблицы и гистограммы представляет значения перечисленных датчиков, усредненные по времени (в целом по системе). В этом пункте можно, таким образом, получить информацию о доле времени, в течение которого система находилась в том или ином состоянии.

Пункт "диаграмма по датчикам" дублирует пункт "диаграмма по модулям" с той лишь разницей, что здесь диаграммы по конкретному датчику выводятся одновременно по всем (насколько позволяет экран) процессорным модулям (рис. 13).

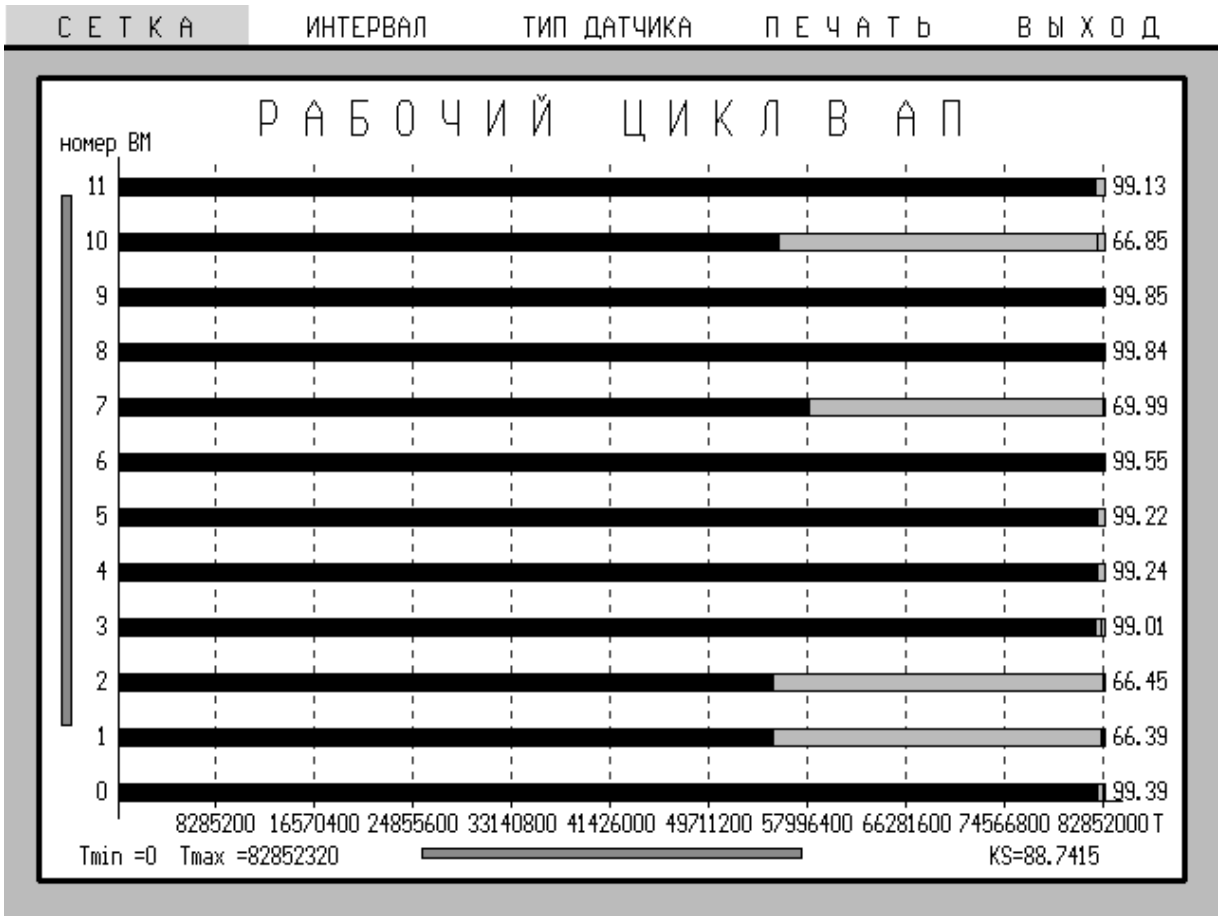


Рис. 13. Диаграмма загрузки АП.

Пункт "распределение активаций" идентичен описанному ранее такому же пункту для программы Ideal.

Пункт "совмещенные процессы и работы" позволяет вывести графики, представляющие во времени количество потенциально мигрирующих процессов (работ) в каждом процессорном модуле и в системе в целом, а также количество совмещенных процессов в каждом модуле. Под совмещенными здесь понимаются все готовые к выполнению, а также отложенные из-за обращения в ОП процессы.

Так же, как и в программе Ideal, пользователь может регулировать шаг сетки при выводе графиков и диаграмм, выводить график в заданном временном интервале, изменять количество точек разбиения при выводе распределения емкости активаций, выводить графики, диаграммы и таблицы на печать.

Анализ причин снижения эффективности

Разработка рекурсивно-параллельных алгоритмов и программ, как и параллельное программирование любого другого рода, требует учета особенностей выполнения программы на вычислительной системе, для которой она предназначена. При этом, как правило, возникают проблемы, специфичные именно для данного подхода к параллельному программированию, которые в силу этой их особенности было трудно предусмотреть заранее. Не учитывать их при программировании – значит заведомо сделать свою программу недостаточно эффективной. Выявить же возможные причины снижения эффективности параллельного выполнения программы можно, как правило, только экспериментальным путем.

Выше был описан подход к проведению эксперимента по оценке эффективности рекурсивно-параллельной программы, написанной на языке RPC, а также программное обеспечение, позволяющее такой эксперимент произвести. С целью выявления причин, которые могут понизить эффективность исполнения RPC-программы, с использованием этого программного комплекса было произведено имитационное моделирование параллельного выполнения большого числа рекурсивно-параллельных программ. Эти программы были посвящены решению задач из области линейной алгебры, дискретной оптимизации, газодинамики и других возможных областей применения.

В результате проведенного исследования был выявлен ряд причин, которые в процессе работы приводят к непроизводительным простоям вычислительных ресурсов и соответственно снижают эффект от параллельного исполнения RPC-программы. Так, например, излишне мелкое дробление задачи на независимые параллельные ветви приводит к неоправданному росту накладных расходов за счет излишних передач работы между процессорными модулями, а нерационально организованное совмещение вычислений и процесса доступа в общую память может

привести к неравномерному распределению работы между модулями параллельной вычислительной системы.

В результате анализа и классификации выявленных причин они были разбиты на следующие группы:

1. недостаточный параллелизм задачи;
2. невозможность обеспечить равномерное распределение работы по процессорным модулям;
3. большой объем накладных расходов на организацию рекурсивно-параллельного процесса;
4. недостаточность удельного веса параллельных операций доступа в память;
5. "надкусывание" активаций;
6. плохая динамика обратного хода рекурсии (процесс "свертки");
7. плохая динамика прямого хода рекурсии (разворачивание до листьевых активаций);
8. плохая динамика распределения работы по системе;
9. большой объем вычислений на последовательных участках алгоритма.

На основе разработанного способа диагностики и оценки степени влияния этих причин, который описывается ниже, был определен состав специальных датчиков для сбора имитационной моделью RPM статистической информации о ходе параллельного вычислительного процесса. Собранная статистика сохраняется в файле и используется в дальнейшем в качестве входной информации для программы-анализатора (Anal.exe), которая входит в состав программного комплекса для разработки и исследования РП-программ.

В приложениях приведено описание датчиков, информация о которых содержится в файле статистики, собираемой программой имитационного моделирования Mod64. Из него можно сделать вывод, что в любой момент времени t известны значения датчиков $d_{ij}(t)$ (i – номер процессорного модуля, j – номер типа датчика), характеризующих состояние вычислительной системы. Если датчик общий для всей вычислительной системы, то для него используется обозначение $d_j(t)$. Ниже приводится реализованный в программе-анализаторе алгоритм определения с помощью этих датчиков причин неэффективной работы РП-программы.

- ***Недостаточный параллелизм задачи***

Для выявления этой причины используется датчик 6. Значение $d_6(t)$ равно количеству работ (активаций параллельных процедур), накопленных в системе к моменту времени t. Зададим функцию $F_0(t)$ следующим образом:

$$F_0(t) = \begin{cases} 0, \text{ если } d_6(t) \geq N_{pm}, & N_{pm} \text{ - число процессорных модулей в системе} \\ 1, \text{ в противном случае} \end{cases}$$

Параллелизм задачи мы считаем недостаточным, если доля времени, когда F0(t)=1, превышает некоторое заданное пороговое значение α:

$$\int_0^T F_0(t)\,dt \leq \alpha,$$

По умолчанию принято α=0.5. Изменить это значение можно, если войти в пункт меню программы-анализатора "Параметры".

В качестве оценки степени влияния этой причины на качество работы РП-программы используется значение

$$Res_1 = \int_0^T F_0(t) * \sum_{i=1}^{N_{pm}} (1 - d_{i1}(t))\,dt / N_{pm},$$

где T – модельное время.

Здесь использован датчик $d_{i1}(t)$, который принимает значение 1 в течение рабочего цикла i-го процессорного модуля и 0 – во все остальное время.

- ***Невозможность обеспечить равномерное распределение работы по процессорным модулям***

Данная причина, как правило, вызвана либо сильной неравномерностью распределения вычислений по листьевым активациям при недостаточном их количестве (а мы считаем, что основной объем вычислений сосредоточен именно в листьях), либо при равновесных листьевых активациях некратностью их количества числу процессорных модулей.

Для диагностики этой причины используется датчик d_8 – емкость активаций. Оценка степени ее влияния на качество выполнения РП-программы вычисляется следующим образом.

Пусть N_a – число активаций, $V_{сум}$ – их суммарный объем, а $V_{ср}=V_{сум}/N_a$ – средний. Заведем массив V размерности N_{pm}. Произведем укладку активаций в V по следующему алгоритму. Будем последовательно выбирать активации из файла статистики; если объем выбранной активации меньше некоторой критической доли α (значение параметра α выбирается пользователем, по умолчанию равно 1/2) от среднего объема активации $V_{ср}$, то ее отбрасываем. В противном случае ищем в массиве V элемент с минимальным значением и увеличиваем его на объем очередной активации. По окончании укладки ищем максимальный элемент в массиве V: $T_{max}= \max \{ V_i, 1 \leq i \leq N_{pm} \}$.

В качестве оценки принимается значение

$Res_2 = (T_{max} * N_{pm} - V_{сум}) / V_{сум}$.

- ***Большой объем накладных расходов на организацию рекурсивно-параллельного процесса***

Обычно эта причина диагностируется в случае, если объем вычислений в листьевых активациях недостаточно велик (мелкий гамак) и не компенсирует временных затрат на организацию рекурсивно-параллельного вычислительного процесса. Для диагностики используется датчик $d_{i1}(t)$, описанный выше. Пусть

$$T_{real} = \int_0^T \sum_{i=1}^{N_{pm}} d_{i1}(t)\,dt -$$

суммарное время, затраченное всеми процессорными модулями на решение задачи.

Тогда оценка накладных расходов есть

$Res_3 = (T_{real} - V_{сум}) / T_{real}$.

- ***Недостаточность удельного веса параллельных операций доступа в память***

О возможности возникновения данного недостатка следует подумать не только при написании программы, но еще при выборе алгоритма решения задачи. Следует отдавать предпочтение тем алгоритмам, при реализации которых удельный вес параллельных операций доступа в память (т.е. объем вычислений, приходящийся на одну операцию Load/Store) был как можно больше. Если алгоритмически повысить удельный вес обменных операций уже не удается, стоит писать программу так, чтобы Load/Store производились на фоне полезных вычислений, например, осуществлять опережающую "подкачку" исходных данных и/или запаздывающую запись результатов вычислений. Примеры таких программ приведены в приложении.

Для диагностики используется датчик 5. $d_{i5}(t)$ принимает значение 1, если в момент t имеет место простой i-го ПМ по причине ожидания завершения операций обмена с общей памятью. Оценка степени влияния данной причины имеет вид

$$Res_4 = \int_0^T \sum_{i=1}^{N_{pm}} d_{i5}(t)\,dt / (N_{pm} * T).$$

Величина Res_4 имеет смысл средней доли процессорных модулей, простаивающих из-за обменных операций.

- ***"Надкусывание" активаций***

"Надкусывание" листьев – это захват одним процессорным модулем нескольких листьевых активаций. Механизм этого захвата примерно таков: на ПМ начинается выполнение листьевой активации, в начале которой, как правило, стоит обменная операция. Модуль запускает параллельный процесс доступа, но вычисления останавливаются на операторе синхронизации Wait(), и центральный процессор захватывает следующую активацию из дека. Та перестает быть потенциально мигрирующим процессом и ее исполнение возможно теперь только на данном ПМ. В начале этой активации также стоит операция доступа в память, и процесс повторяется. Это будет продолжаться до тех пор, пока хотя бы одна из запущенных операций доступа в память не завершится. В результате складывается ситуация, когда некоторые модули захватили несколько листьевых (и, очевидно, наиболее трудоемких) активаций, а другие простаивают из-за отсутствия работы.

Чтобы избежать этого недостатка, достаточно использовать установку режима монопольного использования ПМ в листьевых активациях (оператор Set_M()), который запрещает захватывать дополнительную работу во время простоя из-за операций доступа в общую память. Многочисленные эксперименты показывают, что это весьма действенная мера, и потери, вызванные таким запретом, намного меньше, нежели потери из-за неравномерной загрузки вычислительных ресурсов.

Здесь, однако, следует сказать об еще одной ситуации, которая может привести к такому же результату, то есть захвату некоторыми модулями излишней работы, в то время как другие ПМ простаивают. Пусть вычислительный процесс организован так, что на обратном ходе рекурсии он сворачивается не до конца, а затем разворачивается вновь, образуя "вложенные гамаки". Это довольно типичная организация вычислений. В качестве примера можно привести РП-алгоритм решения рекуррентных соотношений (см. приложения). Пусть, например, такой вычислительный процесс разворачивается до ширины 4, затем сворачивается до ширины 2 и вновь разворачивается. Пусть для вычислений используется система с 4 ПМ. Тогда распределение по модулям может произойти так, как это показано на рис. 14.

Здесь двойной рамкой обозначены листьевые активации, цифры обозначают номер ПМ, на котором активация выполняется. Теоретически ситуация может сложиться следующим образом. Первыми завершают работу модули 0, 1, 2. Модуль 0 порождает две новых листьевых активации, одну оставляет себе, а другую отдает на исполнение 2 ПМ. После этого завершает работу 3 ПМ, однако активация, обозначенная звездочкой, содержащая точку синхронизации, находится на 2 ПМ и не может получить управление, пока модуль занят выполнением полученной от 0 ПМ работы. В результате 1 и 3 модули простаивают.

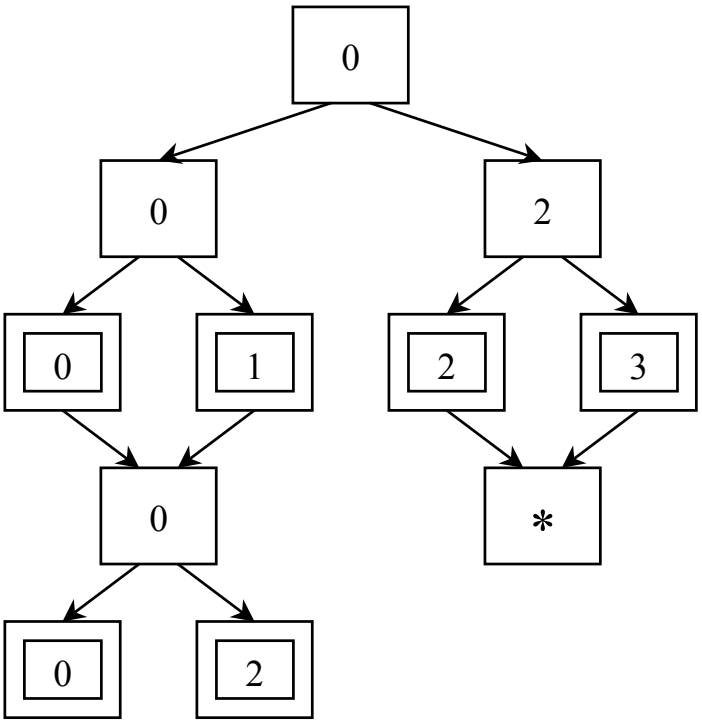


Рис. 14. Структура вычислительного процесса
с "надкусыванием" второго типа.

Справедливости ради отметим, что такого сорта ситуация построена искусственно и на практике автору не встречалась. Однако ему не известен способ борьбы с подобной проблемой, хотя бы и возможной только теоретически. По крайней мере, применяемый алгоритм динамической балансировки в такой ситуации может распределить работу не оптимальным образом.

Для диагностики "надкусывания" используются датчики 3 и 4. $d_{i3}(t)$ принимает значение 0 при отсутствии работы в i-м ПМ и 1 – в противном случае, $d_{i4}(t)$ равно количеству готовых к выполнению параллельных активаций процедур, захваченных i-м ПМ.

Пусть

$$\bar{a} = \begin{cases} 0, & \text{если } a = 0, \\ a - 1, & \text{в противном случае} \end{cases}$$

Тогда

$$F_1(t) = \bigcup_{i=1}^{N_{pm}} \bar{d}_{i4}(t) * \sum_{i=1}^{N_{pm}} (1 - d_{i3}(t)) -$$

количество безработных ПМ в момент t, если в системе есть отложенная работа, и 0 – в противном случае.

Тогда для оценки влияния данной причины можно предложить величину

$$Res_5 = \int_0^T F_l(t)\,dt / (N_{pm} * T).$$

- ***Плохая динамика обратного хода рекурсии (процесс "свертки")***

Анализатор диагностирует эту причину, если время, необходимое для сворачивания вычислительного процесса от ширины N_{pm} до 1, сравнительно велико. В качестве рекомендаций по исправлению этого недостатка можно предложить следующее.

Если процесс скрутки от листьев (т.е. процедур, содержащих основной объем полезных вычислений) к старшим ветвям гамака нагружен дополнительными вычислениями, посмотрите, нельзя ли их сократить, произведя оптимизацию или внеся часть работы в тело листьевой активации. Если процесс порождения параллельных ветвей вычислений реализован не рекурсивно, попробуйте использовать рекурсию, т. к. в этом случае работа равномернее распределяется по процессорам, и при одинаковом объеме листьев разрыв во времени окончания их выполнения меньше. Если параллелизм задачи невелик, листья неравновесные и имеют большой объем, стоит увеличить их число.

Анализатор следующим образом производит оценку влияния данной причины. Определим $T_{обр}$ как время от момента, когда количество работ в системе в последний раз сравняется с числом модулей до момента окончания вычислений. В качестве оценки используется значение

$Res_6 = T_{обр} / T$.

- ***Плохая динамика прямого хода рекурсии (разворачивание до листьевых активаций)***

Данная причина имеет место, если сравнительно велико время, необходимое для того, чтобы рекурсивно-параллельный процесс достиг ширины N_{pm}. Рекомендации пользователю в этом случае заключаются в следующем.

Если процесс прямого хода рекурсии нагружен дополнительными вычислениями, можно попытаться их сократить, произведя оптимизацию или внеся часть работы в тело листьевой активации. Если процесс разворачивания задачи до листьев реализован не рекурсивно, рекомендуется использовать рекурсию, т. к. при этом работа равномернее распределяется по процессорам.

Программа-анализатор для обнаружения этой причины работает следующим образом. Определим $T_{прям}$ как время от момента начала вычислений до момента, когда количество работ в системе первый раз сравняется с числом модулей. В качестве оценки используется значение

$Res_7 = T_{прям} / T$.

- ***Плохая динамика распределения работы по системе***

Данная причина возникает при неудачном выборе конфигурации вычислительной системы. Например, при запуске задачи с числом листьев, равных числу модулей, и ненулевым минимальным запасом работы в деке ("жадными" деками) те модули, на которых происходит выполнение нелистьевых активаций, оставляют себе "запас" активаций. В результате остальные модули не могут получить работу, несмотря на ее наличие в системе. Для устранения данной причины следует изменить конфигурацию вычислительной системы (или ее модели).

Для диагностики используются датчики 3 и 9. $d_{i9}(t)$ принимает значение, равное количеству не активизированной работы, накопленной в i-м модуле к моменту времени t. Смысл датчика 3 был объяснен выше.

Пусть

$$F_2(t) = \bigcup_{i=1}^{N_{pm}} d_{i9}(t) * \sum_{i=1}^{N_{pm}} (1 - d_{i3}(t)).$$

Величина $F_2(t)$ равна количеству безработных модулей, если в деках каких-либо других ПМ есть работа, и равна нулю в противном случае.

В качестве оценки степени влияния данной причины используется значение

$$Res_8 = \int_0^T F_2(t)\,dt / (N_{pm} * T).$$

- ***Большой объем вычислений на последовательных участках алгоритма***

Определим $T_{посл}$ как время, в течение которого количество работ в системе (ширина вычислительного процесса) не больше 1. Оценка влияния последовательных вычислений на качество выполнения параллельной программы производится по формуле

$Res_9 = (T_{посл} / T) * ((N_{pm}-1) / N_{pm})$.

При анализе причин со 2 по 9 программа-анализатор полагает, что при $Res_i > DEL$ соответствующая причина неэффективности имеет место и выдает на экран соответствующее сообщение. Параметр DEL можно легко изменить (пункт "Параметры" в меню анализатора). Однако при уменьшении его значения ниже 5% достоверность результатов анализатора резко падает.

В ряде случаев наличие одной из перечисленных причин снижения эффективности влечет за собой диагностику не только этой, но и одной

или нескольких других "причин". Например, при недостаточном параллелизме задачи анализатор одновременно обнаружит невозможность равномерного распределения работы по системе. Такие "причины" мы будем называть наведенными. Ниже приводится таблица, описывающая такие ситуации.

		Номер основной причины								
		1	2	3	4	5	6	7	8	9
Номер наведенной причины	1									
	2	×								
	3									
	4									
	5									
	6	×	×	×		×				
	7	×		×					×	
	8									
	9									

Символ × означает наличие наведенной причины при возникновении основной. При выводе результатов анализа сообщения о наведенных причинах блокируются.

Вызов программы-анализатора (файл Anal.exe) может содержать в качестве параметра имя файла статистики. В этом случае программа сразу приступает к его обработке. При работе в среде RPMSHELL этот вызов осуществляется путем выбора пункта меню Statist / Analysis.

Программа предоставляет пользователю следующие возможности:

- назначение другого файла статистики для анализа (пункт меню "Файл");
- выбор режима обработки: файл в целом или по отдельным гамакам (пункт "Режим");
- изменение ранее упомянутых параметров DEL и α (пункт "Параметры";
- просмотр значений всех оценок (пункт "Тестовые функции");
- информация по алгоритму вычисления оценок ("Описание"), а также по причинам снижения эффективности параллельного выполнения программы и способам устранения этих причин (пункт "Рекомендации").

Работа с протоколами

Как было отмечено выше, результаты работы программ имитационного моделирования сохраняются в файлах протокола, причем результаты предыдущих экспериментов не удаляются. При работе в среде RPMSHELL приняты следующие соглашения.

Файл протокола модели Ideal имеет расширение .pr0, файл протокола модели Mod64 – расширение .pr1. Кроме того, при соблюдении описанного ниже порядка проведения эксперимента оболочка формирует сводный файл протокола, содержащий всю информацию из двух упомянутых файлов. Этот файл имеет расширение .prt. Имя файла протокола задает пользователь, по умолчанию оно совпадает с именем файла трассы.

Рекомендуемый порядок проведения эксперимента следующий. Перед началом эксперимента файлы, предназначенные для вывода протоколов, лучше удалить, чтобы они не содержали результатов, не относящихся к данному эксперименту, либо задать уникальное имя файла трассы, а следовательно, и файлов протокола (F9). Программа пользователя запускается в режиме построения трассы. После этого следует запуск программы Testgr, а затем серия запусков программы Mod64 для различных конфигураций системы. В результате упомянутые выше файлы протокола будут содержать информацию, относящуюся именно к данному эксперименту. Рекомендуемый порядок, разумеется, не является единственно возможным, в каждом конкретном случае он выбирается исходя из потребностей пользователя.

Файлы протокола – это текстовые файлы, имеющие следующий формат. В начале файла перечисляются параметры, вычисляемые программами моделирования, и дается их расшифровка, а далее через разделитель следуют результаты отдельных запусков этих программ. Такой формат предназначен для машинной обработки посредством программы Protocol. Формат ее запуска следующий:

```
Protocol.exe <••• ••••• •••••••••>
```

При работе в среде RPMSHELL запуск этой программы осуществляется из пункта меню Statist / Protocol. Программа позволяет оформить результаты эксперимента, представленные в протоколе, в виде графиков зависимостей одних величин от других. При этом в качестве как аргументов, так и функций, могут быть любые параметры, перечисленные в заголовке и имеющие численное значение (некоторые параметры – просто строки, например, имя файла графа). Их не может быть более 20.

Для того, чтобы вывести график (рис. 15), в пункте главного меню АРГУМЕНТ необходимо выбрать требуемый аргумент, а затем, перейдя в пункт ФУНКЦИЯ / ПРОИЗВОЛЬНЫЕ, выбрать необходимую функцию. Это можно сделать либо посредством нажатия в соответствующем месте клавиши Enter, либо отметить несколько функций (не более 5) клавишей Insert, а затем выбрать пункт ГРАФИК. При выводе одновременно

нескольких функций следует иметь в виду, что оцифровка оси ординат в этом случае производится так, чтобы они поместились на график все, и не стоит одновременно выводить функции, значения которых существенно различны (на несколько порядков).

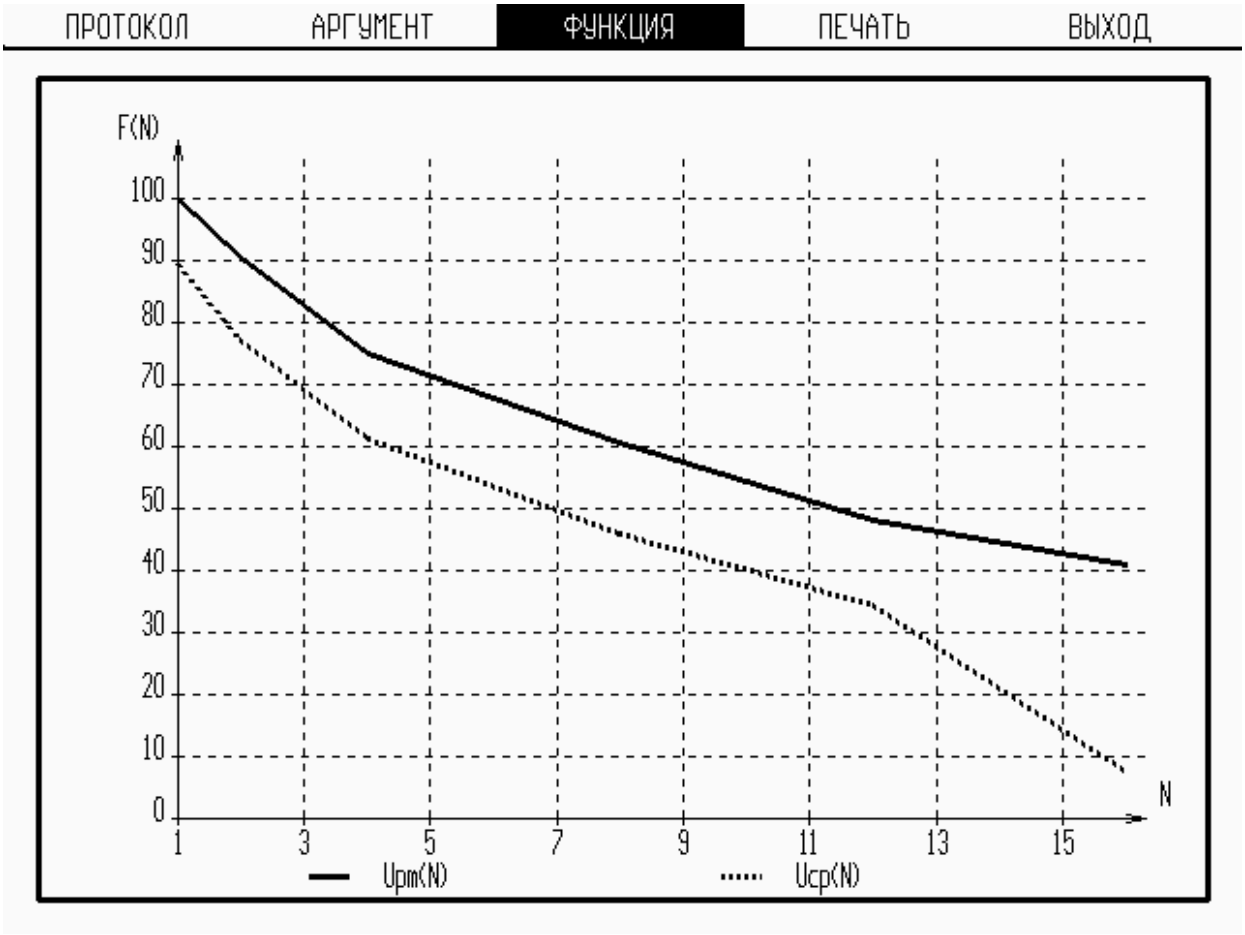


Рис. 15. Графики загрузки системы в зависимости от количества ПМ, построенные средой по результатам серии экспериментов.

Кроме того, в пункте меню ФУНКЦИЯ / СТАНДАРТНЫЕ есть возможность вывести оценки ускорения и оценки качества работы программы как функции от числа процессорных модулей в системе. Для этого используются следующие параметры из файла протокола:

Ts – суммарное время вычислений
Tk – время критического пути
Sa – средний параллелизм
N – количество процессорных модулей
T – реальное время вычислений

Отсутствие некоторых из них приводит, естественно, к невозможности построения соответствующего графика.

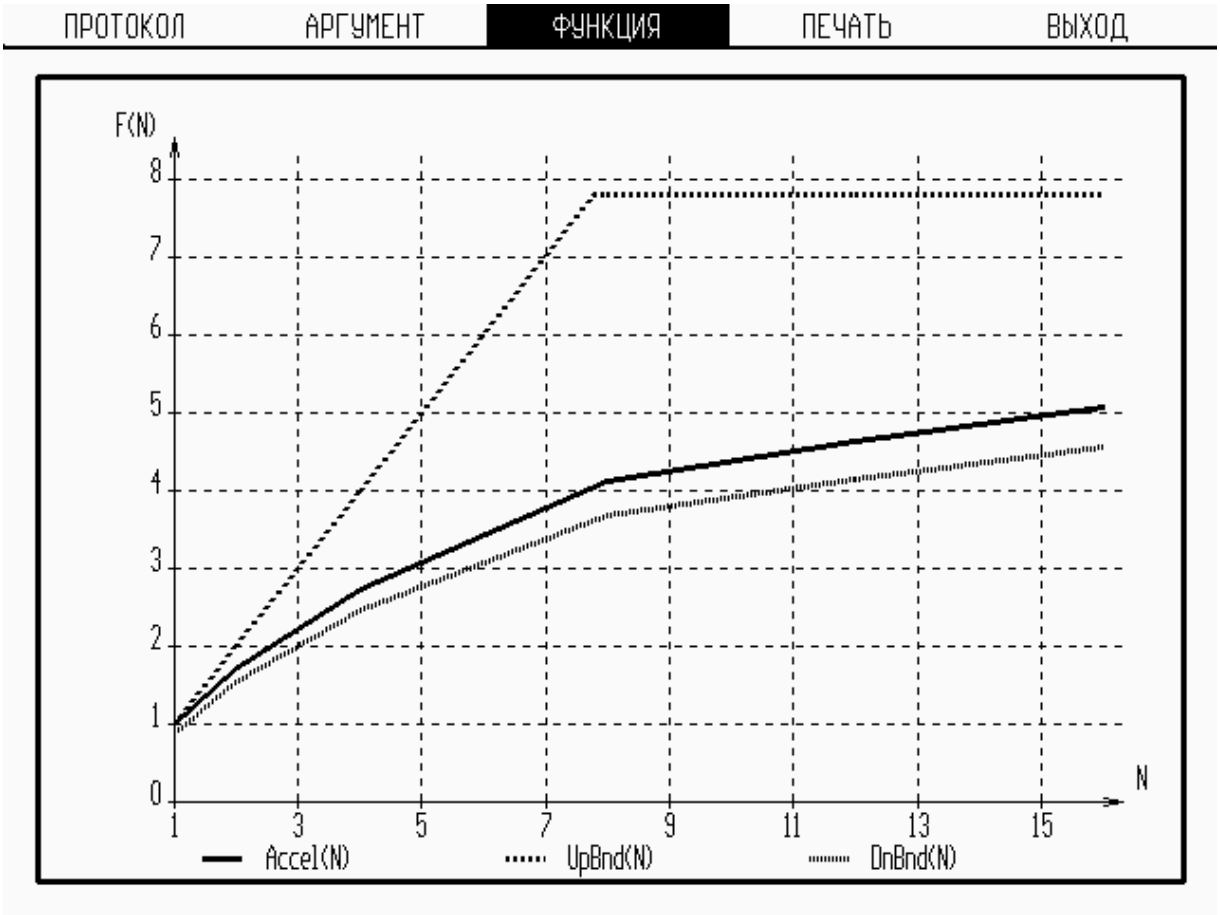


Рис. 16. Графики ускорения в зависимости от количества ПМ, построенные средой по результатам серии экспериментов.

Ускорение работы программы в параллельном режиме (рис. 16) вычисляется как

Accel(N)=T(1)/T(N).

Поскольку вычисление ускорения требует знания T(1) – времени работы программы на однопроцессорной системе, в файле протокола должен присутствовать соответствующий эксперимент. Можно также вывести верхнюю и нижнюю оценку ускорения, которые вычисляются как

UpBnd(N)=min(N, Sa)

и

DnBnd(N)=Ts/T(N)

соответственно .

В пункте меню КАЧЕСТВО (рис. 17) можно вывести оценку степени использования внутреннего параллелизма программы

Soft(N) = Accel(N)/Sa ,

оценку степени использования аппаратуры

Hard(N) = Accel(N)/N,

а также усредненную оценку качества работы программы, которая вычисляется как

Quality(N) = α ∗ Soft(N) + (1-α) ∗ Hard(N).

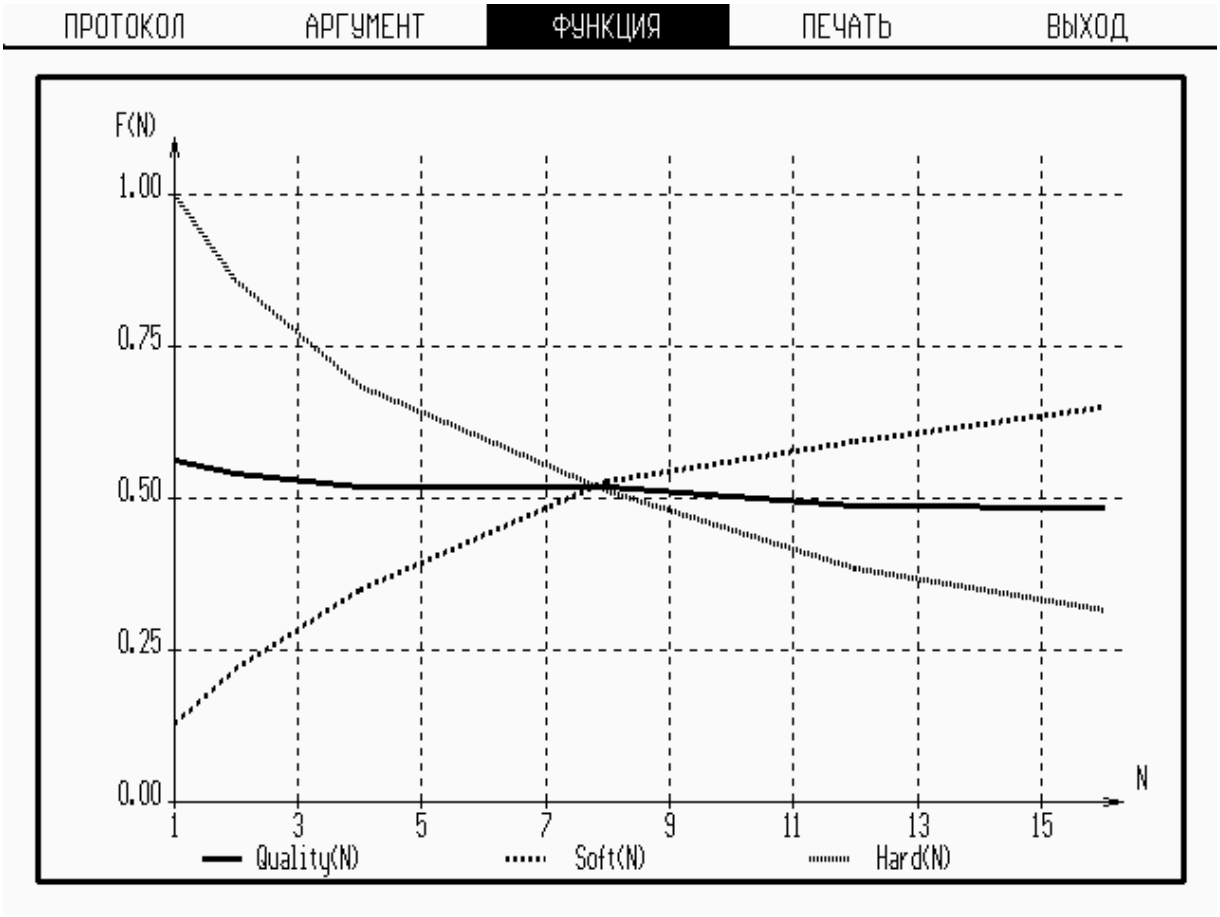


Рис. 17. Графики оценки использования ресурсов, построенные средой по результатам серии экспериментов.

Значение коэффициента α по умолчанию полагается равным 0.5, его можно изменить, выбрав соответствующий пункт меню.

Программа позволяет также вывести построенные графики на печать (пункт меню ПЕЧАТЬ). Пункт меню ПРОТОКОЛ позволяет посредством программы Viewer (она входит в стандартный комплект среды RPMSHELL) вывести на экран содержимое файла протокола.

Стоит отметить также и следующую возможность работы с этой программой. Поскольку файл протокола – это обычный текстовый файл, пользователь может вносить туда различные добавления, например, комментарии. Он может расширить список параметров путем добавления своих. Например, пользователь хочет добавить в протокол программы Testgr свой параметр, имеющий смысл ограничения на объем вычислений в листьевой активации, и назвать его Volume. Тогда он должен поместить в "шапку" файла <имя>.pr0 строку вида

Volume - <Комментарий>,

а в результаты каждого эксперимента добавить элемент, содержащий значение этого параметра вида

Volume=<Значение>.

Теперь при работе программы Protocol эта информация будет обрабатываться так, как будто бы ее сформировала программа Testgr, и можно будет, например, вывести график зависимости средней ширины параллелизма программы от максимального объема листьевой активации.

Поиск ошибок

Данный параграф посвящен описанию методов и программного обеспечения, позволяющего диагностировать некоторые ошибки, специфичные для параллельного выполнения RPC-программы.

Собственно, ошибки, которые могут появиться при разработке рекурсивно-параллельной программы, написанной на языке RPC, можно разделить на следующие основные группы.

Первая группа - это, разумеется, синтаксические ошибки. Те из них, которые нарушают синтаксис языка C, легко обнаруживаются при компиляции программы в любом из возможных режимов, поскольку для этого используются стандартные компиляторы. Поэтому поиск и исправление таких ошибок никаких сложностей не вызывает.

К синтаксическим следует отнести также ошибки, которые нарушают те правила написания программы, которые специфичны для языка RPC. Сюда относятся неверное оформление заголовков рекурсивно-параллельных процедур и их вызовов, использование запрещенных конструкций языка C, использование глобальных переменных там, где этого делать нельзя и тому подобное. Такие ошибки не вызывают, разумеется, никакой реакции трансляторов. Более того, в последовательном режиме программа может правильно отработать. Однако при работе в других режимах или при моделировании могут возникать различные ошибки, до первопричины которых бывает не так просто докопаться. Еще хуже, если ошибка никак внешне не проявляется, но приводит к неправильным результатам.

Для поиска такого рода ошибок используется специальный синтаксический анализатор. Его описанию посвящен отдельный параграф.

Вторая группа ошибок – это ошибки, которые обязательно проявляются во время выполнения программы в одном из допустимых режимов. Самые простые из них – это ошибки синхронизации, то есть попытки завершить выполнение активации процедуры до тех пор, пока не завершились все порожденные ею параллельные процессы. Эти ошибки сравнительно легко обнаруживаются в процессе выполнения программы.

К сожалению, правильная работа программы в последовательном режиме никоим образом не гарантирует, что и в параллельном режиме программа выполнится правильно (или вообще выполнится), поскольку

при последовательном выполнении невозможно отследить взаимодействие одновременно выполняемых параллельных процессов и совместное использование ими общих ресурсов. Ошибки, специфичные для параллельного выполнения программы, – это третья группа ошибок. Основная неприятность здесь заключается в том, что даже при работе в параллельном режиме эти ошибки могут проявляться или не проявляться в зависимости от того, как взаимодействующие параллельные процессы расположены во времени. Последнее обстоятельство делает обнаружение подобного рода ошибок весьма нетривиальной задачей.

Ниже предлагается метод, позволяющий гарантированно обнаружить некоторые ошибки этого типа. Слово "гарантированно" означает, что диагностируется принципиальная возможность такой ошибки, то есть существование такого размещения параллельных процессов во времени, при котором ошибка будет иметь место.

На настоящий момент программным обеспечением, входящим в состав среды RPMSHELL, выявляются следующие ошибки:

- попытка использования элементов блока параметров, переданного дочернему процессу до его завершения;
- попытка одновременного использования одной и той же области общей памяти потенциально параллельными процессами (за исключением случая, когда все они считывают эту область);
- завершение активации до закрытия всех запущенных ею параллельных процессов.

Алгоритм поиска ошибок работы с общедоступной памятью

Излагаемый ниже метод относится ко второму пункту, поскольку обнаружение остальных перечисленных ошибок – чисто технический вопрос.

Собственно, и сформулировать проблему лучше в несколько более общем виде: как обнаружить возможность одновременного использования некого общего ресурса (если это не разрешено) потенциально параллельными процессами. Словом "процесс" мы назовем часть общего вычислительного процесса, оформленную таким образом, что она может быть передана на выполнение другому процессорному модулю. Применительно к программированию на RPC процессом является активация параллельной процедуры.

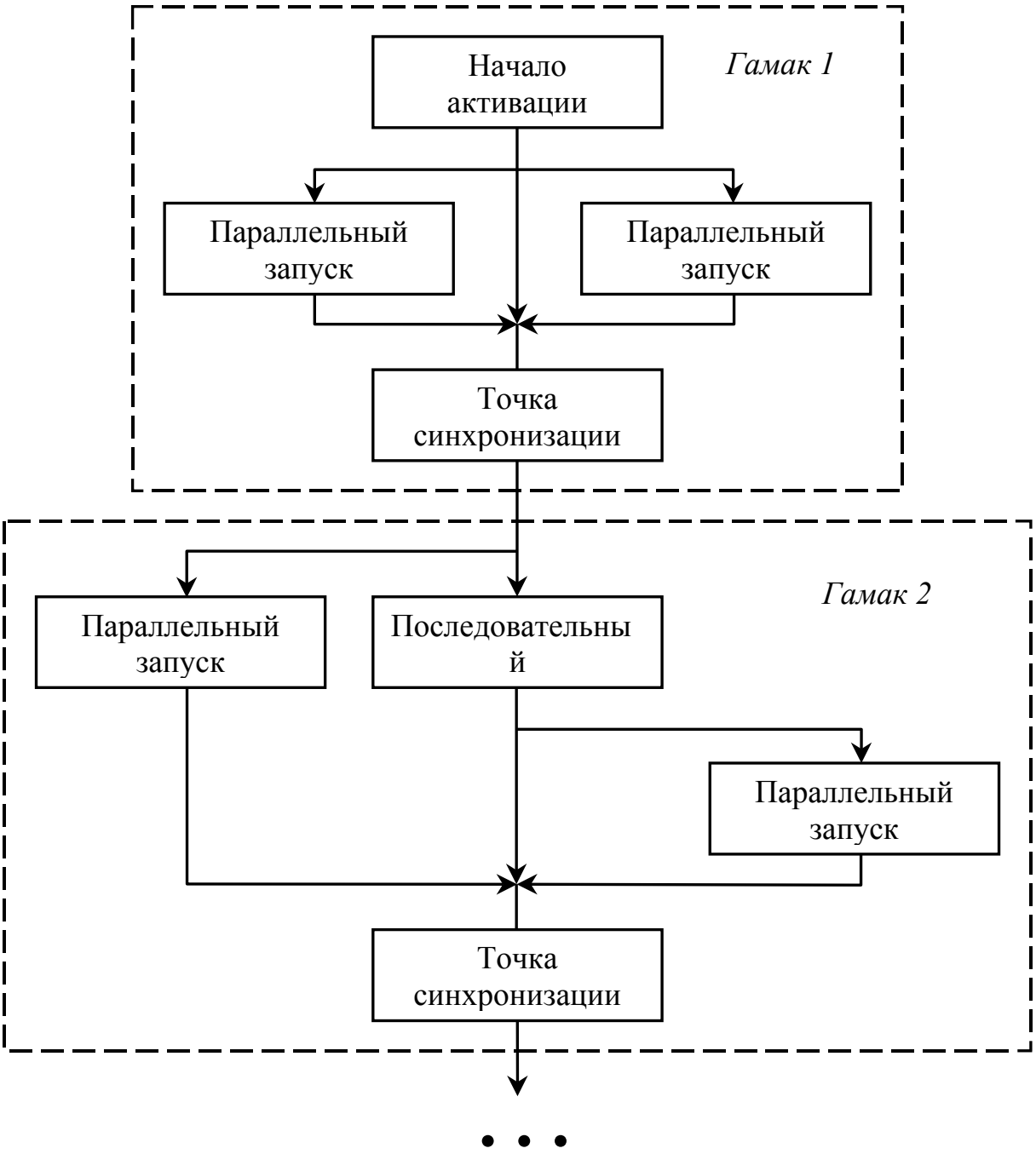


Рис. 18. Граф информационной зависимости между процессами.

Информационная зависимость между процессами, определяющая порядок их выполнения, может быть представлена в виде ориентированного графа, вершинами которого являются параллельные и последовательные запуски процессов, точки их запуска и точки синхронизации, как изображено в примере на рис. 18.

Как видно из рисунка, типичной является ситуация, когда весь процесс можно разделить на некоторое множество так называемых

гамаков, которые включают в себя участок вычислений между двумя точками синхронизации. Точнее таким вид графа будет только в том случае, если в каждой точке синхронизации мы дожидаемся завершения всех запущенных ранее (в данном процессе) дочерних процессов, а в RPC это именно так. В свою очередь каждый из дочерних процессов может быть представлен в виде подграфа аналогичной структуры.

Под потенциальной параллельностью процессов мы понимаем возможность такого расположения их во времени, что они будут выполняться одновременно. Разумеется, даже при параллельном выполнении программы конфликт может и не возникнуть, если потенциально параллельные процессы будут выполняться не одновременно. Не являются потенциально параллельными, например, процессы, принадлежащие различным гамакам, или дочерний процесс с той частью родительской активации, которая следует после соответствующей точки синхронизации.

Ошибкой, очевидно, является попытка потенциально параллельных процессов что-то записать в одну и ту же ячейку общедоступной памяти или ситуация, когда один процесс что-то туда записывает, а другой считывает, поскольку результат в этом случае является непредсказуемым.

Реализованный в среде RPMSHELL подход к поиску таких ситуаций заключается в следующем. Программа компилируется и запускается в так называемом режиме поиска ошибок параллельного исполнения. В процессе последовательного выполнения программы накапливается информация обо всех операциях доступа к общему ресурсу. В нашем случае ресурсом является общая память, а операциями операции доступа к ней все варианты операций Load и Store. Необходимая информация включает в себя:

- полный идентификатор процесса, породившего операцию Load или Store,
- информацию о типе операции доступа и области общей памяти, к которой она адресуется.

Теперь при каждой новой операции доступа достаточно проверить, не может ли она вступить в конфликт со всеми, встречавшимися ранее. При этом определение, являются ли процессы, породившие подозрительные операции, потенциально параллельными, происходит путем анализа полных идентификаторов этих процессов. Ниже предлагается способ идентификации процессов, пригодный для этой цели.

Мы будем идентифицировать процесс уровня вложенности k путем задания набора $A_1, A_2, \ldots, A_k$, где A_i – идентификатор i-го предка данного процесса среди всех процессов, порожденных (i-1)-м предком. В свою очередь идентификатор $A_i = (A_i^h, A_i^p)$ состоит из номера гамака A_i^h и номера процесса в этом гамаке A_i^p. При этом процессы, запущенные

параллельным и последовательным образом, нумеруются отдельно, и A_i^p, следовательно, содержит информацию о том, как именно был запущен данный процесс. Мы введем обозначение

$$\phi(A_i^p) = \begin{cases} 1, & \text{если процесс запущен параллельным образом,} \\ 0, & \text{если процесс запущен как последовательный.} \end{cases}$$

Теперь пусть мы имеем процессы A=(A_1, A_2, ..., A_n) и B=(B_1, B_2 ,..., B_m), причем первый был запущен (при последовательном выполнении) раньше. Определить, являются ли эти процессы потенциально параллельными, можно по следующему алгоритму.

Если $A_i=B_i$ для всех i от 1 до min (n, m), то один процесс является потомком другого, следовательно, они не являются потенциально параллельными.

Иначе существует такое k, что для всех i от 1 до k-1 $A_i=B_i$, и $A_k \neq B_k$. Тогда

Если $A_k^h \neq B_k^h$, то процессы относятся к разным гамакам k-го предка и не могут быть запущены параллельно.

Иначе

Если $\phi(A_k^p) = 0$, то процесс A запущен последовательным образом и не может выполняться одновременно с процессом B, запущенным позднее и не являющимся потомком A.

Иначе процессы потенциально параллельны.

Программные средства поиска ошибок, специфичных для параллельного режима выполнения

В программном обеспечении, входящем в среду RPMSHELL, для реализации описанного алгоритма используется подход, аналогичный тому, который применяется для моделирования параллельного исполнения RPC-программы [2]. Для того, чтобы проанализировать свою программу на наличие упомянутых выше ошибок, пользователь должен откомпилировать и выполнить ее в специальном режиме. При этом операторы языка RPC реализуются как библиотечные функции, которые и выполняют описанные действия.

Выполнение программ в параллельном режиме

В существующей на настоящий момент реализации средств поддержки рекурсивно-параллельного программирования компиляция и сборка программы для работы в параллельном режиме производится с использованием сред turboC либо BorlandC. Процесс формирования исполняемого модуля полностью аналогичен описанному выше процессу

компиляции и сборки для режимов имитационного моделирования и поиска ошибок. По умолчанию именем исполнимого файла для параллельного режима будет p_<имя задачи>.exe. Напомним, что именем задачи считается имя текущей директории.

Полученный исполнимый модуль позволяет выполнить программу в параллельном режиме на локальной сети компьютеров. На момент написания данной работы программы и библиотеки среды RPMSHELL обеспечивали работу только в системе с поддержкой сетевых протоколов IPX/SPX. В ближайшее время планируется закончить разработку средств поддержки параллельного режима выполнения РП-программ с использованием протоколов TCP/IP под управлением Windows 95/98/NT.

Запуск программы на выполнение происходит следующим образом. На всех компьютерах, образующих вычислительную сеть, должен быть вызван исполнимый модуль программы (один и тот же). При запуске копии программы сообщают о своем присутствии всем уже запущенным ранее копиям и устанавливают каналы связи "каждый с каждым". После того, как программа запущена на всех компьютерах, на которых предполагается ее выполнение, на одном из них (любом) дается команда на исполнение. Этот компьютер становится "главным" (только в том смысле, что именно на нем исполняется функция main()) и имеет номер 0. Номера остальных компьютеров определяются порядком их регистрации на "главном" модуле.

При использовании среды RPMSHELL запуск программ удобнее всего производить следующим образом. На всех периферийных компьютерах запускается программа Slave.exe или, если запущена RPMSHELL, выбирается пункт меню Run / On RPM / Slave. После этого все управление может осуществляться с будущего "главного" компьютера.

Предусмотрены следующие варианты запуска параллельного исполнимого кода:

- без использования сети – пункт меню Run / On RPM / No Net. В этом случае программа будет исполняться только на одном компьютере.
- запуск на сети одной и той же копии программы, например, с общего жесткого диска – пункт меню Run / On RPM / Run One Copy.
- копирование исполнимого кода программы, а также других необходимых файлов (в том числе файла конфигурации – если требуется изменить установки по умолчанию), – пункт меню Run / On RPM / Send File, а затем запуск на всех компьютерах сети – пункт меню Run / On RPM / Run Many Copies.

Упомянутые выше установки, содержащиеся в файле конфигурации, могут быть изменены выбором пункта меню Run / On RPM / Edit Config. Доступны следующие настройки:

- включение и выключение работы с сетью;

- настройка параметров, управляющих работой распределительного механизма, в частности "жадностью" ДЕКа;
- управление размерами ДЕКа, программных стеков, очередей ГВП и Send, системных очередей;
- изменение номера прерывания для работы с сетью.

Синтаксический анализ текста РП-программ

Часто при разработке и реализации новых концепций программирования оказывается, что существующие языки высокого уровня не в полной мере удовлетворяют требованиям разработчиков. В то же время зачастую достаточно не слишком больших модификаций одного из общепринятых языков, чтобы он вполне согласовывался с разрабатываемой концепцией. Как правило, достаточно наложить некоторые ограничения на использование конструкций базового языка и дополнить его некоторыми специфическими конструкциями, синтаксически правильными с его точки зрения. Реализовать такие конструкции можно либо как системные вызовы (если реализация концепции предполагает разработку своей операционной системы), либо как вызовы функций специально разработанной библиотеки, возможно, после обработки текста специально написанным препроцессором.

В любом случае разработчик имеет как минимум два существенных плюса. Во-первых, нет необходимости разработки собственного компилятора и отладчика, достаточно воспользоваться одним из существующих для базового языка. Во-вторых, потенциальным пользователям, знающим базовый язык, достаточно просто овладеть навыками работы с надстройкой, тем более со знакомым компилятором. Хотя, разумеется, кроме знания собственно языка необходимо иметь четкое представление об особенностях архитектуры вычислительной системы, способах распределения работы, передачи информации, организации памяти и т.п. В каком-то смысле, наверное, такие языки нельзя считать самостоятельными языками программирования, однако ниже мы будем употреблять именно этот термин, поскольку слово "надстройка" звучит, на наш взгляд, несколько тяжеловесно.

Упомянутый выше прием достаточно распространен в области параллельного программирования, и в частности при разработке программ, предназначенных для выполнения на распределенных вычислительных системах [3]. Наиболее часто используемыми для этих целей базовыми языками являются C, C++, Fortran. Не является исключением и рассматриваемая в данной книге концепция рекурсивно-параллельного программирования, ориентированная на архитектуру параллельной вычислительной системы с динамической балансировкой загрузки. В качестве языка программирования здесь используется описанный выше язык RPC. RPC построен как надстройка над некоторым подмножеством языка C. Ограничения, накладываемые на использование обычных

конструкций С, в основном связаны с особенностями организации памяти в RPM. Однако нарушение правил использования переменных, относящихся к тому или иному классу памяти RPM, никак не может быть обнаружено с помощью С-компилятора. Точно так же не могут быть обнаружены некорректные с точки зрения RPC описания и вызовы параллельных процедур, а также нарушения соглашений об использовании обычных последовательных функций, поскольку с точки зрения С-компилятора они могут быть совершенно корректными.

Ниже рассматриваются методы и программные средства, предназначенные для синтаксического анализа с целью поиска именно ошибочных с точки зрения RPC конструкций, не являющихся ошибочными с точки зрения языка С. При этом, разумеется, не ставилась цель произвести доскональный синтаксический анализ текста и обнаружить ошибки даже в том случае, если программист, используя гибкость С, намеренно обманывает анализатор – предполагалось, что это в его цели не входит.

Следует отметить также, что сформулированная задача в ходе работы получила существенное обобщение, и потому предлагаемый вниманию читателя подход и средства его поддержки могут быть использованы в том числе и для проверки корректности текстов, написанных на языках, являющихся надстройками как над языком С (не только RPC), так и над другими языками программирования. Правда, в последнем случае потребуется, как будет видно в дальнейшем, несколько больше работы.

Организация и основные этапы проверки корректности текста программы

Разработанная и реализованная схема проверки текста программы представлена на рис. 19. Она состоит из нескольких этапов. На рисунке необязательные этапы изображены пунктиром, а альтернативные этапы – штрих-пунктиром. Этапы обработки, реализованные в рамках одной программы-анализатора, обведены линией из точек.

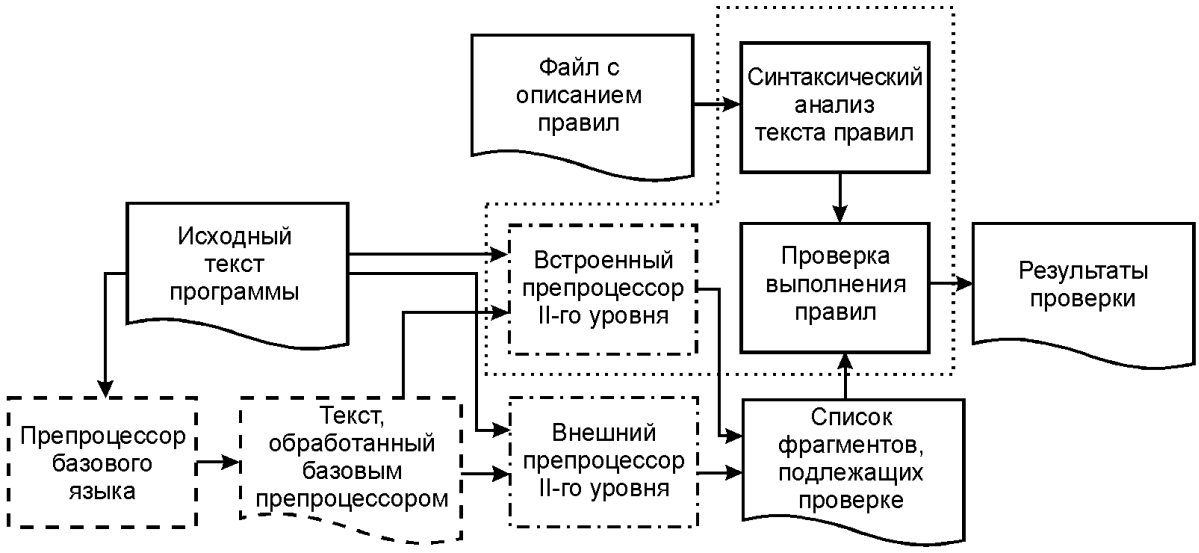


Рис. 19. Основные этапы проверки текста программы на соответствие правилам.

Первый этап (необязательный) состоит в обработке текста препроцессором I-го уровня (препроцессором базового языка). В качестве такового в случае, если базовым является язык C, выступает препроцессор языка C. Цель – выполнение директив препроцессора #include и #define (если необходимо), поскольку при последующей обработке все директивы игнорируются, и, следовательно, не подвергается проверке текст включаемых файлов и не осуществляется подстановка макроопределений. Если выполнение данных директив не повлияет, по мнению разработчика, на результаты проверки (например, подключается только текст на стандартном C, и макроопределения не затрагивают конструкций программы, специфичных для RPC (в нашем случае)), то данный этап может быть опущен.

Правила, которым должны удовлетворять конструкции программы, описываются на специально разработанном языке (будем называть его далее языком описания правил – ЯОП), основные моменты которого рассмотрены ниже. Правила, сформулированные на этом языке, хранятся в виде текстового файла и считываются оттуда программой-анализатором. При этом происходит проверка синтаксической корректности данного описания и трансляция его во внутренний формат.

Следующий этап – обработка текста препроцессором II-го уровня. Его задача – выделить из текста конструкции (фрагменты), подлежащие проверке. Разумеется, процесс выделения этих конструкций привязан к синтаксису базового языка. Имеющаяся в данный момент версия программы ориентирована на использование в качестве такового языка C, и ее встроенный препроцессор будет корректно работать с программами, написанными на расширениях C. Если в качестве базового выступает

другой язык высокого уровня, то для пользования анализатором пользователю придется написать свою программу выделения фрагментов. По этой причине передача результатов работы препроцессора II-го уровня собственно анализатору организована через файл. Такая передача информации в пределах одной программы организована исключительно в целях обеспечения большей гибкости, поскольку только так можно организовать анализ текста программ, написанных на языках, не являющихся расширениями C. Для этого требуется другой препроцессор второго уровня, написанный специально для этого языка, поскольку в тексте программы необходимо контролировать уровень вложенности блоков, наличие комментариев, текстовых констант и т.п., то есть конструкции, синтаксис которых зависит от базового языка. Формат этого файла, служащего входной информацией для собственно анализатора, описан в соответствующей документации, что позволяет при необходимости выполнить такую работу. Соответственно в опциях запуска программы предусмотрена возможность принудительного отключения встроенного препроцессора II-го уровня.

Конструкции, подвергаемые проверке, описаны в файле правил в виде так называемых масок, которые последовательно накладываются на текст программы с целью их выявления. На третьем этапе происходит собственно проверка того, удовлетворяет ли исходный текст программы сформулированным в файле правил требованиям, и вывод результатов проверки.

Возможности и структура языка описания правил

Детальное описание ЯОП потребовало бы слишком много места, потому просто отошлем заинтересованного читателя к соответствующей документации. Ниже мы лишь перечислим основные возможности задания правил проверки и основные элементы данного языка. Он позволяет

- описать искомые конструкции в виде масок-шаблонов, которые должны полностью либо частично совпасть с фрагментом текста программы;
- задавать уровень вложенности, на котором должна либо не должна встретиться конструкция;
- формулировать правила в виде логических выражений с указанием направления зависимости между ними;
- контролировать как выполнение правил, так и использование некоторых типов переменных, ограничивая его определенными конструкциями;
- задавать вид сообщения в случае возникновения той или иной ошибки;
- задавать условия проверки правил как результат проверки правил, проверенных ранее;

- вмешиваться в процесс проверки путем задания специальных модулей обработчиков для определенных событий.

В тексте файла описания правил предусмотрены следующие элементы:

- директивы заголовка;
- директивы управления;
- правила;
- макроопределения;
- обработчики событий.

В рамках данной работы мы ограничимся рассмотрением правил как основного средства решения поставленной выше задачи. Формат правила выглядит следующим образом:

\begin{verbatim} [Вид зависимости] Тип правила {

```
MESSAGE {
NAME = "••• •••••••"
ER1 = "••••••••• •• •••••• •••• 1"
ER2 = "••••••••• •• •••••• •••• 2"
ER3 = "••••••••• •• •••••• •••• 3"
WARNING = "•••••••••••••• • ••••••••• ••••••"
}
VAR (•••••••••• ••••••••••) {
•••••••••
}
RULE #••• ••••••• {
••••• ••••••••• •••••••••
•••••••••• ••••••••••
•••••• ••••••••• •••••••••
} }
```

Здесь вид зависимости показывает, проверяется ли выполнение собственно правила, либо идет проверка уникальности переменных, либо то и другое одновременно. Проверка уникальности переменных используется, когда язык предполагает возможность использования некоторых специальных типов переменных только в определенном контексте. Например, в RPC таковыми являются дескрипторы для работы с общей и статической памятью. Тип правила задает момент, когда происходит проверка выполнения правила: при обработке последнего фрагмента (глобальное правило) или после падения каждого уровня вложенности блоков (локальное правило).

В разделе сообщений (MESSAGE) можно задать текст диагностических сообщений, которые выдаются при возникновении того или иного типа ошибки. Все пункты данного раздела являются необязательными.

В разделе переменных (VAR) операторы представляют собой, вообще говоря, логические выражения, значения которых управляют тем, производится ли проверка выполнения правила относительно фрагментов

и проверка правила относительно переменных. Направление зависимости показывает, какое из выражений в следующем разделе является главным при присвоении значений переменным ЯОП. Последние представляют собой строки символов, обычно соответствующие идентификаторам из текста анализируемой программы.

Наконец, в разделе выражений (ключевое слово RULE) содержится описание правила относительно фрагментов анализируемого текста. Направление зависимости показывает, какое из выражений является ведущим, а какое ведомым при проверке правила относительно фрагментов. Если ведущее выражение ложно, то проверка не осуществляется. Если оно истинно, то значение ведомого выражения должно быть истинным, в противном случае диагностируется ошибка. Сами логические выражения представляют собой набор масок фрагментов для поиска с указанием уровня вложенности, связанных посредством логических операций. С целью установления соответствия одной конструкции другой, например (для RPC), предварительного объявления параллельной процедуры CPP() и оператора ее вызова P_Call(), вводится понятие внутренней переменной программы-анализатора. Последняя представляет собой множество строк, которые встречаются в анализируемой конструкции на указанном месте. В приведенном ниже примере переменные в основном имеют имена AL, BL (по правилам ЯОП их окружают символы ~. Использование переменных позволяет для каждой конструкции P_Call(), например, определить, как должно выглядеть соответствующее ей объявление CPP().

Предлагаемый язык описания правил и программные средства позволяют осуществлять анализ текста программ, написанных на расширениях, а также подмножествах или расширениях подмножеств стандартных языков программирования с целью проверки соблюдения ограничений и правил, которая не может быть выполнена компилятором для базового языка. На разработанном языке были сформулированы правила, которым должны удовлетворять программы, написанные на языке RPC, что позволяет использовать данные средства в программном комплексе рекурсивно-параллельного программирования для синтаксического анализа программ.

Вместе с тем разработанный подход и программные средства обладают достаточной гибкостью, что позволяет его использовать для анализа текста программ, написанных на других расширениях языка C, а также других языков высокого уровня.

Ниже в качестве иллюстрации приводятся сформулированные на ЯОП правила, которым должны удовлетворять программы, написанные на текущей версии RPC. Комментарии к правилам, а также примеры позволяют получить достаточно полное представление о реализованных типах контроля и возможностях ЯОП и программы-анализатора.

Описание правил проверки корректности программы на RPC

В данном пункте использованы фрагменты текста на языке описания правил, содержащиеся в файле Rpm.all, который наряду с RPC-текстом является исходной информацией для программы-анализатора AllTst.exe. Возможно, читатель обратит внимание на несколько неестественную нумерацию правил. Она сложилась исторически, и мы не сочли необходимым здесь что-то менять. Возможно, в дальнейшем в синтаксис RPC будут внесены какие-то изменения, которые повлекут необходимость модификации этих правил. Однако и в этом случае, по-видимому, удобнее, чтобы закрепленные за тем или иным правилом номера оставались неизменными.

Для удобства в текст добавлены комментарии. В качестве иллюстраций используются фрагменты текста на RPC, содержащие тот или иной вид ошибок. Имеется в виду несоответствие этих фрагментов именно требованиям языка RPC, с точки зрения C они совершенно корректны. Вообще при использовании описываемого анализатора предполагается, что текст РП-программы предварительно был обработан C-компилятором, и в нем были устранены все синтаксические (с точки зрения C) ошибки. В противном случае корректная работа программы синтаксического анализа будет невозможна.

```
$head  "V2.0"
$levels 0..1
// •••••••• ••••••• •••••• •• ••••••• • ••••••••••
•••••••.
$warning OFF

// ••••••• 1.
// •••• ••••••••• ••••• •••••-•• •• ••••••••• •••••••:
// P_Call(), P_Send(), H_Call() – •• ••••••••• ••••••
// (•.•. • •••• •••••-•••• •••••••) ••• ••••••••••
••••••••
// Parallel() •• ••••••• ••••••, •• •• •••••• ••••••••
••••••
// •••• ••••••••• ••••••••• •••••••• CPP() •• •••••••
// ••••••. ••••••• ••••••• CPP() •• ••••••• •••••• •••••
// •••••••••• •••••• • ••••••••• ••••••• ••••••••• •••••••
// P_Call(), P_Send(), H_Call() ••• ••••••• ••••••••• •
// •••••••• Parallel()

 [Rule,Var]Global {
      MESSAGE {
        NAME="••••••• •••••••• • •••••• ••••••••••••
•••••••"
        ER1="••••• ••••••••• ••••••• ••••••••• •••••• •••
```

```
                ••••••• ••••••••••• ••••••• P_Call(),
P_Send(),       H_Call(), Parallel(), •• •• ••••••• •••••••
                CPP() ••• •••••-•••• ••••••••••• •••••••."
          ER2="••••• ••••••• ••••••• ••••••• ••••• •••
                ••••••• ••••••••••• ••••••• P_Call(),
P_Send(),       H_Call(), Parallel(), •• •• ••••••• •••••••
                CPP() ••• ••••••••••••• ••••••••••
•••••••."
          ER3="••••••• ••••••••••• ••••••• •••••••
                •••••••••• ••••••• CPP() •••-••••, •••••
                ••••••••• ••••• ••• ••••••• ••••••••••
                ••••••• P_Call(), P_Send(), H_Call(),
Parallel()"
        }
        VAR (=>){
          Var_Check #TRUE
        }
        RULE #rule1 {
          0{c|CPP(~AL~)}
            <=
          {
            1{c|P_Call(~AL~,@)} OR
            1{c|P_Send(~AL~,@,@)} OR
            1{c|H_Call(~AL~,@)} OR
            0{c|Parallel(~AL~,@)}
          }
        }
      }
```

Ниже приводятся примеры фрагментов текста на RPC, содержащие ошибки, на которых анализатор обнаружит несоответствие сформулированному правилу.

Пример 1:

```
#include “seq.h”

struct BP
  { long sss; };

Parallel(proc1,Bpar)
PARAM(BP,*Bpar);
{
. . .
}

void main(){
  NEW_PARAM(BP, np);
  P_Call (proc1, &np);
}
```

В приведенном примере было опущено предварительное описание параллельной процедуры CPP(proc1). Анализатор диагностирует сразу две ошибки: первую (ER1) и вторую (ER2) – и выведет на экран соответствующий текст.

Пример2:

```
#include “seq.h”

struct BP
  { long sss; };

Parallel(proc1,Bpar)
PARAM(BP,*Bpar);
{
. . .
}

CPP(proc1);

void main(){
  NEW_PARAM(BP, np);
  P_Call (proc1,&np);
  proc1(&np); /* •••••• */
}
```

В приведенном примере также содержится две ошибки. Во-первых, описание CPP(proc1) располагается после описания параллельной процедуры proc1(), во-вторых, последний ее вызов выполнен некорректным с точки зрения RPC способом. Анализатор диагностирует две ошибки: ER2 – на строке с описанием Parallel(), и ER3 – на вызове proc1(&np).

```
// ••••••• 3.
// •••••• ••••••••• ••••••••••• P_Send(), P_Call(), H_Call()
// •••••• •••• •••••••••• •• •••••••••• ••••••••• •••
// ••••••• •••••••, ••••••••• ••••• ••••••••• •••••••••
// ••••••••• NEW_PARAM() • ••••••• ••••••• •••••••••, •••
// •••••• ••••••••• ••••••••• ••••••• Parallel()

 [Rule]Local {
      MESSAGE {
         NAME="••••••• ••••••• ••••••••• •••••••••••
               ••••••"
         ER2="•••••• ••••••••• ••••••••••• P_Send(), P_Call(),
               H_Call() •••••• •••• •••••••••• •• ••••••••••••
               ••••••••• ••• ••••••• •••••••, ••••••••• •••••
               ••••••••• ••••••••• ••••••••• NEW_PARAM() •
               ••••••••• ••••••• •••••••••, ••• ••••••
               ••••••••• ••••••••• ••••••• Parallel()"
      }
```

```
      VAR (=>){
         Var_Check #TRUE
         Rule_Check #FALSE
      }
      RULE #rule3 {
        {
           1{c|NEW_PARAM(@,<~AL~[,~CL~>@)} OR
           {
             0{c|Parallel(@,~AL~@)} AND
             0{c|PARAM( @,*~AL~@)}
           }
        }
        <=
        {
           1{c|P_Send(@,<EMPTY,&~AL~[,&~CL~,~AL~>@,@)} OR
           1{c|P_Call(@,<EMPTY,&~AL~[,&~CL~,~AL~>} OR
           1{c|H_Call(@,<EMPTY,&~AL~[,&~CL~,~AL~>}
        }
     }
  }
```

Пример фрагмента текста на RPC, нарушающий данное правило.

```
#include “seq.h”

struct BP
  { DWORD sss; };
CPP(proc1);

Parallel(proc1,bp)
PARAM(BP,*bp);
{ int *aaa;
  P_Call(proc1, *bp); /* •••••• 1 */
  H_Call(proc1, aaa); /* •••••• 2 */
}
```

Ошибка 1 заключается в попытке передать процедуре параметр неверного типа (структуру вместо указателя на структуру). Правильный вызов:

```
P_Call(proc1,bp);
```

Ошибка 2 состоит в том, что переменная aaa не описана должным образом. В обоих случаях будет выдано сообщение ER2 с указанием строки, в которой находится ошибочный фрагмент.

```
// ••••••• 4.
// ••••••••• RPC, •• ••••••••• •••••••••• ••••• C, ••••• ••••
// •••••••••••• ••••• • •••• ••••••• main() ••• • •••• •••••••,
// •••••••••, ••• •••••••••••• (••••••• ••••••••• Parallel() ).
// ••••••• •••••••••••• ••••••••• •••••••: • •••••• ••••••••••
// •••••-•••• •••••••••• •• ••••••• •••••••••• •••••••••
// ••••••••• •••••••, • ••••••• ••• ••••••••••, •••••• •••••
// •••, •••••••••••••• ••••••••, •••••••• • ••••• •••••
// ••••••••.
```

```
$warning ON
 [Rule]Local {
      MESSAGE {
         NAME="••••••• •••••••••••• •••••••••••••• ••••••• •
               RPC"
         ER1="•••••••••••• ••••••••••, •••••••••• ••• RPC,
              •••••••••• • •••• •••••• ••••••• main() •••
              •••••••, ••••••• ••••••••• Parallel()."
      }
      VAR (=>){
        Var_Check #FALSE
      }
      RULE #Rule4 {
        {
           0{h|@ main(@)} OR
           0{c|main(@)} OR
           {
              0{c|Parallel(@,~AL~)} AND
              0{c|PARAM(@,*~AL~)}
           }
        }
        <=
        {
           1{c|P_Call (~AL~@ ,~AL~@)} OR
           1{c|P_Send (~AL~@ ,~AL~@ ,~AL~@)} OR
           1{c|H_Call (~AL~@ ,~AL~ @ ) } OR
           1{c|Wait   ( ) } OR
           1{c|Load_1 (~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Load_V (~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Load_I (~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Load_M (~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Load_G (~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @,
                       ~AL~ @,~AL~ @ ) } OR
           1{c|Store_1(~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Store_V(~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Store_I(~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Store_M(~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|Store_G(~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @,
                       ~AL~ @,~AL~ @ ) } OR
           1{c|G_Alloc(~AL~ @,~AL~ @,~AL~ @,~AL~ @,~AL~ @ ) } OR
           1{c|G_Free (~AL~@) } OR
           1{c|Lock   (~AL~@) } OR
           1{c|Unlock (~AL~@) }
        }
      }
  }
$warning OFF
```

Пример нарушения сформулированного правила:

```
void func1()
{
  Wait();
}
```

Ошибка заключается в попытке вызвать оператор RPC (Wait()) в функции, описанной как последовательная. Кроме основной задачи – обнаружения попыток использования параллельных операторов в последовательных функциях – данное правило позволяет осуществлять контроль за количеством параметров в этих операторах. Например, на фрагмент вида

```
Lock(); /* ••• ••••••••• */
```

анализатор выдаст предупреждение о несоответствии количества аргументов – оператор Lock() должен иметь один параметр.

```
// ••••••• 4.1
// •••••• •••••••• ••••••••• Parallel() •••••• ••••
// ••••••••••, • •••••• •••••••• ••••••••• PARAM() –
// ••••••••• •• ••• •• •••••••••
 [Rule]Global {
      MESSAGE {
         NAME="••••••• •••••••• •••••••••• ••••• ••••••••••"
         ER2="•••••• •••••••• ••••••••• Parallel() •••••• ••••
               ••••••••••, • •••••• •••••••• ••••••••• PARAM() -
               ••••••••• •• ••• •• ••••••••••."
      }
      VAR (<=){
        Var_Check #TRUE
        Rule_Check #FALSE
      }
      RULE #Rul41 {
        {
           0{c|Parallel(@,~AL~)} AND
           0{c|PARAM(@,*~BL~)}
        }
        =>
        {
           0{c|Parallel(@,~BL~)} AND
           0{c|PARAM(@,*~AL~)}
        }
     }
 }
```

По-видимому, примеры для сформулированного правила излишни.

```
// ••••••• 5.
// •••• ••••••••• •• ••••••• •••••• •••••••••, ••• •••••••
// •••••••••, ••••••• ••• handleDM, ••• ••• •••••• •••••••••
// •••• •• ••••, •• ••• ••••• •••••••••• •••••• • ••••••
// •••••-•••• •• •••••••, •••••••••••• •••••• •• •••••
// ••••••••••, •••••• ••••••• ••••••••• ••••••••• ••
// ••• •••••••••
```

```
 [Rule,Var]Global {
      MESSAGE {
          NAME = "••••••• •••••••••••• •••••••••••• •••••
                  ••••••"
```

```
            ER2 = "•••••••• • •••••• •••••••, ••••••• ••••••
                   •••• ••••••••• •• ••••••••• ••••••••• ••••
                   handleDM, •• ••••• •••• •• •••••••• •••••••."
            ER3 = "••••••••••, ••••••• ••• handleDM •••••
                   •••••••••••••• •••••• • ••••••••••• •••••••
                   ••• •••••• • •••••••••••••• ••••• •••••••."
         }
         VAR (=>){
            Var_Check #TRUE
            Rule_Check #FALSE
         }
         RULE #Rule5 {
           {
              0{d|handleDM <~AL~[,~BL~>} OR
              0{d|extern handleDM <~AL~[,~BL~>} OR
           }
           =>
           {
              1{c|G_Alloc(@,@,@,@,<&~AL~[,&~BL~,~AL~>}OR
              1{c|G_Free(<&~AL~[,&~BL~,~AL~>} OR
              1{c|Load_1(@,<&~AL~[,&~BL~,~AL~>@,@,@)} OR
              1{c|Load_V(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Load_I(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Load_M(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Load_G(@,<&~AL~[,&~BL~,~AL~>@,@,@,@,@,@)} OR
              1{c|Store_1(@,<&~AL~[,&~BL~,~AL~>@,@,@)} OR
              1{c|Store_V(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Store_I(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Store_M(@,<&~AL~[,&~BL~,~AL~>@,@,@,@)} OR
              1{c|Store_G(@,<&~AL~[,&~BL~,~AL~>@,@,@,@,@,@)}
           }
        }
     }
```

Пример неправильного текста на RPC.

```
handleDM hdm;

void main()
{
  int aaa;
  G_Free(&hdm); /* ••••••••• */
  hdm++; /* •••••• 1 */
  G_Free(aaa); /* •••••• 2 */
}
```

Ошибка 1 заключается в недопустимом использовании переменной типа handleDM – выводится сообщение ER3.

Ошибка 2 состоит в попытке передать параметр неверного типа – выводится сообщение ER2.

```
// ••••••• 5.1
// •••••••••• •••••••• ••••••••• •••• handleSM • handleDM
// •• ••••••••• ••••••
 [Var,Rule]Global{
```

```
      MESSAGE {
         NAME = "••••••• •••••••• ••••••••••• •••• handleSM •
                 handleDM"
         ER2 = "•••••••••••• ••••••••• ••••••••••• •••• handleSM
                 • handleDM •• ••••••••••• ••••••."
      }
      VAR (<=) {
         Rule_Check #FALSE
         Var_Check #TRUE
      }
      RULE #Rule51 {
         {
            1{d|handleDM ~AL~} OR
            1{d|handleSM ~AL~} OR
            1{c|handleSM_Init(~AL~,@,@)} OR

            1{d|extern handleDM ~AL~} OR
            1{d|extern handleSM ~AL~} OR
            1{c|extern handleSM_Init(~AL~,@,@)} OR

         } => #TRUE
      }
   }
```

Пример:

```
void main()
{
  handleDM hdm;
}
```

Диагностируется ошибка ER2 – описание переменной типа handleDM внутри функции.

```
// ••••••• 6.
// •••• ••••••••••• •• ••••••• •••••• •••••••••, ••• •••••••
// ••••••••••, ••••••• ••• handleSM, ••• ••• •••••• •••••••••
// •••• •• ••••, •• ••• ••••• •••••••••• •••••• • ••••••
// •••••-•••• •• •••••••, •••••••••••• •••••• •• •••••
// ••••••••••••, •••••• ••••••• ••••••••• ••••••••• ••
// ••• ••••••••••

   [Var,Rule]Global {
        MESSAGE {
            NAME = "••••••• •••••••••••• •••••••••• ••••••"
            ER2 = "•••••••• • •••••• •••••••, ••••••• •••••• ••••
                    ••••••••• •• ••••••••• ••••••••• ••••
                    handleSM, •• ••••• •••• •• •••••••• •••••••."
            ER3 = "••••••••••, ••••••• ••• handleSM •••••
                    •••••••••••••• •••••• • •••••••••• ••••••• •••
                    •••••• •• •••••••••• •••••••."
        }
        VAR (=>){
            Var_Check #TRUE
            Rule_Check #FALSE
        }
```

```
RULE #Rule5 {
    {
      0{d|handleSM <~AL~[,~BL~>} OR

      0{d|extern handleSM <~AL~[,~BL~>} OR

      0{c|handleSM_Init(~BL~,@,@)} OR
    }
    =>
    {
      1{ c | S_Alloc( @, @, <&~AL~[,&~BL~,~AL~>@, @ ) }   OR
      1{ c | S_Free(<&~AL~[,&~BL~,~AL~>}                   OR
      1{ c | S_Load_1( @,<&~AL~[,&~BL~,~AL~>@, @, @ ) }   OR
      1{ c | S_Load_V( @,<&~AL~[,&~BL~,~AL~>@, @, @, @ ) }OR
      1{ c | S_Load_I( @,<&~AL~[,&~BL~,~AL~>@, @, @, @ ) }OR
      1{ c | S_Load_M( @,<&~AL~[,&~BL~,~AL~>@, @, @, @ ) }OR
      1{ c | S_Load_G( @,<&~AL~[,&~BL~,~AL~>@, @, @, @, @,
                       @ ) }                               OR
      1{ c | S_CopyTo_1(<&~AL~[,&~BL~,~AL~>@, @, @, @ ) } OR
      1{ c | S_CopyTo_V(<&~AL~[,&~BL~,~AL~>@, @, @, @, @ ) }
                                                           OR
      1{ c | S_CopyTo_I(<&~AL~[,&~BL~,~AL~>@, @, @, @, @ ) }
                                                           OR
      1{ c | S_CopyTo_M(<&~AL~[,&~BL~,~AL~>@, @, @, @, @ ) }
                                                           OR
      1{ c | S_CopyTo_G(<&~AL~[,&~BL~,~AL~>@, @, @, @, @, @,
                        @ ) }                              OR
      1{ c | S_CopyFrom_1(<&~AL~[,&~BL~,~AL~>@, @, @, @ ) }
                                                           OR
      1{ c | S_CopyFrom_V(<&~AL~[,&~BL~,~AL~>@, @, @, @, @ )
       }                                                   OR
      1{ c | S_CopyFrom_I(<&~AL~[,&~BL~,~AL~>@, @, @, @, @ )
       }                                                   OR
      1{ c | S_CopyFrom_M(<&~AL~[,&~BL~,~AL~>@, @, @, @,@ )
       }                                                   OR
      1{ c | S_CopyFrom_G(<&~AL~[,&~BL~,~AL~>@, @, @, @, @,
                          @, @ ) }
    }
  }
```

Пример возможной ошибки полностью аналогичен примеру для правила 5.

Приемы программирования

Принципиально новый подход к разработке алгоритмов и программированию естественным образом приводит к тому, что порожденный программой вычислительный процесс обладает особенностями, которые не встречались ранее, в том числе и весьма неожиданными. Не являются исключением и программы рекурсивно-параллельного типа, ориентированные на архитектуру рекурсивно-параллельной машины с динамическим распараллеливанием. Зачастую

упомянутые особенности весьма отрицательно сказываются на качестве выполнения программы и могут даже свести на нет весь эффект от параллельного выполнения.

Часть причин недостаточно эффективного выполнения рекурсивно-параллельных программ достаточно очевидна, другие удалось выявить только в результате экспериментов по имитационному моделированию параллельного выполнения программ, но даже для достаточно очевидных причин только с помощью моделирования удалось оценить количественно степень их влияния на снижение качества РП-программы. В результате проведенного исследования были сформулированы предлагаемые ниже основные правила написания РП-программ. Разумеется, нельзя полагать, что их соблюдение гарантирует высокое качество программы, однако можно быть уверенным, что их игнорирование заведомо приведет к тому, что работа программы будет весьма неэффективной.

Как показывает опыт, существенное влияние на эффективность выполнения программы оказывают два фактора:

- организация процесса распараллеливания;
- организация хранения общих данных и доступа к ним.

Организация процесса распараллеливания

Написание программы в рекурсивно-параллельном стиле не является достаточным условием ее эффективного выполнения. С точки зрения распараллеливания она должна быть написана таким образом, чтобы накладные расходы на распараллеливание (организацию рекурсивных вызовов процедур, порождение и синхронизацию параллельных процессов) были значительно меньше, чем объемы полезных вычислений. Для этого алгоритм должен преобразовываться так, чтобы основной объем вычислений приходился на "листьевые" активации процедур, а активации, связанные с рекурсивной "раскруткой" и "обратным ходом" рекурсии, были бы максимально облегченными.

Кроме этих общих рекомендаций есть простой прием, повышающий эффективность распараллеливания, – это сочетание рекурсивно-параллельного описания процедуры с циклическими вычислениями в ее "листьевых" активациях. Проиллюстрируем данный прием на примере вычисления суммы значений функции F(i).

```
struct BP_Sum              /*   ••••••••• ••••• ••••••••••   */
{  int i,j,k;              /*   ••••••••• Sum                */
   float s;
}
/* ••••••••• ••••••••• ••••• ••••••• ••••••• F(i) */
Parallel(Sum,bp)           /*   ••••••••• ••••••••••••       */
PARAM (BP_Sum,*bp);        /*       ••••••••• Sum            */
{  NEW_PARAM (BP_Sum,bp1); /* ••••••• ••••• ••••••••• ••• */
   NEW_PARAM (BP_Sum,bp2); /*  •••••••••• ••••••• Sum       */
   int m;
```

```
    if ((bp->i)-(bp->j)<(bp->k))
                          /* ••••••• ••••••••••• ••••••• */
    {  /* ••••••••• •• ••••• •••••• ••••••••• */
       bp->s=0;
       for(m=bp->i; m<bp->j; m++)
          bp->s+=F(m);
    }
    else
    {  /* •••••••••• ••••••• */
       bp1.i=bp->i; bp1.k=bp->k;
       bp1.j=(bp->i + bp->j)/2;  /*  ••••••••• •••••••••  */
       bp2.i=bp1.j+1;            /*  •••••••••• •••••••   */
       bp2.j=bp->j; bp2.k=bp->k;
       P_Call(Sum,&bp1);         /*  ••••••••• 1 ••••••••  */
       H_Call(Sum,&bp2);         /*  ••••••••• 2 ••••••••  */
       /* ••••••• ••• ••••••• */
       Wait();                   /*  ••••••••••••• ••••••••• */
       bp->s=bp1.s+bp2.s;        /*  ••••••• ••• ••••••••   */
    }
} /* Sum */
```

Суть данного приема программирования заключается в том, что, варьируя параметр k, можно изменять объемы вычислений в "листьевых" активациях. Увеличение вычислительной емкости этих активаций (а зачастую основной объем вычислений сосредоточен именно в них) позволяет снизить долю накладных расходов в суммарном объеме вычислений, а следовательно, повысить полезную загрузку центрального процессора и уменьшить время выполнения программы. Под накладными расходами мы понимаем все действия по организации вычислительного процесса необходимым нам образом и не связанные непосредственно с вычислениями: передача активаций параллельных процедур, их параметров, результатов, действия по активизации параллельных процедур, доступ в общую память и т.п.

Заметим, что увеличивать значение параметра k имеет смысл до некоторого предела, т.к. при этом уменьшается количество "листьевых" активаций и, соответственно, параллелизм программы. Если объем вычислений в "листьевых" активациях одинаков, то значение k будет оптимальным, когда количество активаций равно количеству процессорных модулей (ПМ) в системе (для данного примера это возможно только в том случае, если количество ПМ является степенью двойки).

В случае, если это не так, а также если невозможно обеспечить одинаковый объем "листьевых" активаций в силу того, что он зависит от значений обрабатываемых данных и является непредсказуемым, количество "листьевых" активаций должно быть значительно больше, чем количество ПМ. Это связано с тем, что системе необходим запас нераспределенной работы для осуществления динамической балансировки по загрузке ПМ. Следовательно, оптимальное значение k можно

определить только экспериментальным путем. Причем для задач с сильной неравномерностью объемов вычислений в "листьевых" активациях на эффективность функционирования RPM в большей степени будет влиять механизм динамического распределения работ, а не то, как запрограммирована задача.

Сказанное верно и в том случае, когда значительные вычисления должны быть произведены на обратном ходе рекурсии, например, при программировании некоторых алгоритмов сортировки или задачи нахождения всех частичных сумм. Однако в любом случае введение параметра (или нескольких параметров), управляющих глубиной рекурсии, представляется оправданным.

Отметим также, что в приведенном примере параметр k (количество вызовов функции F()) играл роль оценки количества работы, которое необходимо выполнить (емкость активации). В ряде случаев бывает более правильным использовать другие оценки, причем не только для управления глубиной рекурсии, но и (что более важно) для разбиения работы на возможно более близкие по объему вычислений куски. Это требование может оказаться очень важным для обеспечения равномерного распределения работы по системе.

Впрочем, для более тонкой настройки программы можно внести небольшую неравномерность в размер параллельных ветвей. Так, полезным может оказаться разбиение работы на два почти равных куска, размер одного из которых кратен k. При этом больший кусок модуль оставляет себе (использует H_Call() для его активизации), а меньший помещает в дек. В этом случае зависимость времени работы программы от параметра k становится более тонкой, и обычно можно добиться несколько лучшей загрузки процессорных модулей. Ниже приводится модифицированный участок кода из предыдущего примера.

```
{  /* •••••••••• "•••••••••" */
  int n = bp->j – bp->i + 1,
      h = (n/(2*bp->k))*bp->k;
      n-= 2*h;
      if (n > bp->k) h+=n–bp->k;
      bp1.i=bp->i; bp1.k=bp->k;
      bp1.j=bp->i+h-1;          /*  ••••••••• •••••••••  */
      bp2.i=bp1.j+1;            /*  •••••••••• •••••••   */
      bp2.j=bp->j; bp2.k=bp->k;
      P_Call(Sum,&bp1);         /*  ••••••••• 1 ••••••••  */
      H_Call(Sum,&bp2);         /*  ••••••••• 2 ••••••••  */
      /* •••••••• ••• •••••••• */
      Wait();                   /*  ••••••••••••• ••••••••• */
      bp->s=bp1.s+bp2.s;        /*  •••••••• ••• ••••••••   */
   }
```

Доступ в память

Общедоступная память RPM в основном предназначена для хранения структурированных данных. В языке RPC глобальные переменные размещаются в ОП, причем обращение к ним можно осуществлять только через специальные системные вызовы.

Работу с общей памятью можно охарактеризовать следующим образом:

- время выполнения одной операции доступа в ОП относительно велико по сравнению с временем доступа в локальную память (данные, размещаемые в локальной памяти, соответствуют локальным данным процедур языка RPC); поэтому операции доступа оформляются в виде параллельного процесса и могут выполняться на фоне "основных" вычислений; их синхронизация осуществляется через единый для всех параллельных процессов механизм;
- эффективность доступа к ОП тем выше, чем больший блок информации передается между ОП и локальной памятью;

Существует ряд подходов, позволяющих повысить эффективность работы с ОП:

1) При разработке алгоритма необходимо стремиться к увеличению объема полезных вычислений, приходящихся на одну операцию доступа в общую память, что приводит к снижению интенсивности обращений в ОП. Этого можно достичь путем сокращения общего количества операций доступа в ОП, а также за счет увеличения объема блока, передаваемого за одну операцию доступа. Целесообразно закладывать в алгоритм параметрическое управление объемом передаваемого блока, так как увеличение объема блока часто приводит к увеличению объема вычислений в листьевых активациях и сокращению их количества, то есть к ограничению параллелизма программы (что может привести к снижению эффективности распараллеливания, как это было показано выше). Для задач с неравновесными листьевыми активациями оптимальное значение параметра блока определяется экспериментальным путем;

2) За счет оформления операции доступа к ОП в самостоятельный процесс, можно организовать опережающую подкачку данных из ОП, выполняемую на фоне основных вычислений. Проиллюстрируем данный прием на примере вычисления следующего векторного выражения:

```
s[j] = v[k]*b[k][j]+...+v[i]*b[i][j]+...+v[l]*b[l][j];
```

где i изменяется в пределах от k до l, а j - от m до n.

Пусть вектор v[] и матрица b[][] являются глобальными переменными, размещаемыми в ОП. Тогда процедура на языке RPC, вычисляющая вектор s[] и возвращающая его в качестве параметра, будет иметь следующий вид.

```
struct BP_vector
      { int k,l,n,m;
        int NR, LeafSize;
```

```
        float res[MVLR]; };
extern handleDM hV;        /*   •••••••• ••••••    [NL]      */
extern handleDM hM;        /*   •••••••• •••••••   [NL*NR]   */
extern handleDM hRes;      /*   •••••••••••••• ••••••  [NR]  */

CPP(vector);

Parallel (vector,bp)
PARAM (BP_vector,*bp);
/* MVLL -  •••••••••••• •••••• •••••••••• s • sv,v1,v2 */
{ int i,j,w;           /*        •••••• •••••••••          */
  float  sv[MVLL],     /*  ••••••••• ••••••••• ••••••• v[]   */
         v1[MVLL],     /*  ••••••••• ••••••• ••••••  b[][] */
         v2[MVLL],     /*  ••••••••• ••••••• ••••••  b[][] */
         *current,     /*  ••••••••• •• •••••• ••••••••    */
         *next,        /*  ••••••••• •• •••••••• ••••••••  */
         *tmp;         /*         ••••••• ••••••••         */
/* ••••••••• ••••••• •• ••••••• •• •••• */
  Set_M();
/* ••••••••• ••••••••• ••••••••• "next" • "current" */
  current=v1; next=v2;
/* •••••• •• •• ••••••••• ••••••••• •••••••  v[] */
  Load_1(sv, &hV, bp->k, bp->l-bp->k+1);
/* •••••• •• •• •••••••• ••••••••• •••••••• ••••••• b[][] */
  Load_V(current, &hM, bp->k*bp->NR+bp->n, bp->NR,
                                          bp->l-bp->k+1);
  w=0;
  for (j=bp->n; j<=bp->m; j++)
  {
    Wait();
    if (j < bp->m) /*•••••••••• •••••• ••••••••• ••••••••••*/
      Load_V(next, &hM, bp->k*bp->NR+j+1, bp->NR,
                                            bp->l-bp->k+1);
    /* ••••••••• ••• ••••••• •••••••••• */
    bp->res[w]=0;
    for(i=0;i<=(bp->l-bp->k);i++)
       bp->res[w]+=sv[i]*current[i];
    /* ••••• ••••••••• "next" and "current"  */
    tmp=next; next=current; current=tmp; w++;
  } /* for */
} /* vector */
```

Как видно из процедуры vector(), чтение из ОП следующего подвектора исходной матрицы b[][] будет происходить параллельно с обработкой текущего подвектора.

3) Организация процесса параллельного доступа в ОП может вызвать и негативные явления, снижающие эффективность распараллеливания. В частности, существует так называемый эффект "надкусывания" потенциально мигрирующих процессов (активаций процедур). Потенциально мигрирующим будем называть процесс, который порождается в результате вызова параллельной процедуры. Он может быть выполнен на любом ПМ системы. Процесс остается потенциально мигрирующим до тех пор, пока он не будет активизирован на одном из ПМ

системы. С этого момента он не может быть передан на выполнение другому ПМ, т.е., если активация процедуры начала работу на каком-то процессорном модуле, то она до самого завершения будет выполняться именно на этом ПМ.

В разделе, посвященном анализу причин снижения эффективности параллельного выполнения РП-программ, мы подробно рассматривали причины возникновения эффекта "надкусывания". Следует отметить, что если листьевые активации имеют достаточно большой объем вычислений и их количество невелико по сравнению с количеством ПМ в системе, то этот эффект приведет к очень неравномерной загрузке ПМ.

Существует довольно простой способ борьбы с эффектом "надкусывания", а именно – в листьевой активации перед выполнением операций доступа в ОП необходимо запретить ПМ выборку нового процесса из дека (установить режим монопольного захвата ПМ). Это можно сделать путем вызова процедуры с помощью оператора Set_M() (См. текст процедуры vector()). Алгоритм его работы подробно описан в разделе "Механизм порождения и активизации параллельных процессов".

При выполнении процедуры в монопольном режиме ПМ не может активизировать у себя новый потенциально мигрирующий процесс (даже если простаивает), пока полностью не завершится данная процедура. Как показывает практика, эти простои по сравнению с возможной неравномерной загрузкой ПМ сказываются очень незначительно на эффективности функционирования системы.

О распараллеливании рекуррентных соотношений

В данном параграфе мы приведем еще один пример построения рекурсивно-параллельного алгоритма, а именно, покажем, каким образом можно распараллелить вычисления, заданные рекуррентным соотношением вида: $x_k = b_k + a_k x_{k-1}$.

Оно допускает следующую матричную запись

$$P_k X_{k-1} = X_k, \text{ где } P_k = \begin{pmatrix} 1 & 0 \\ b_k & a_k \end{pmatrix}, \; X_k = \begin{pmatrix} 1 \\ x_k \end{pmatrix}$$

Ясно, что $P_k P_{k-1} \ldots P_1 X_0 = X_k$. Теперь совершенно очевидно, каким образом можно распараллелить эти вычисления.

Пусть теперь нам требуется вычислить не только X_k, но всю последовательность $X_1, X_2, \ldots, X_k$. Как организовать параллельную процедуру в этом случае?

Нам потребуются все частичные произведения вида P_1, P_2P_1, $P_3P_2P_1$, ... , $P_kP_{k-1}\ldots P_2P_1$. Для их вычисления организуем рекурсивно параллельную процедуру AllProd (i, j), результатом работы которой

должны быть все частичные произведения вида P_i, P_iP_{i-1}, $P_iP_{i-1}P_{i-2}$, ... , $P_iP_{i-1}...P_j$.

В теле процедуры AllProd (i, j), если требуется распараллеливание, производятся два вызова AllProd (i, l), AllProd (l-1, j), где l=(i+j)/2. В результате первая активация вычислит первую половину искомых произведений. Для получения второй половины потребуется каждое из произведений, вычисленных второй активацией, домножить на $P_iP_{i-1}...P_l$, то есть на последнее из произведений, вычисленное первой активацией. Эти действия при необходимости можно также организовать параллельным образом.

Отметим здесь также, что процесс умножения вектора X_0 на соответствующие произведения матриц, можно совместить с вычислением последних, если вместо X_0 завести матрицу

$$P_0 = \begin{pmatrix} 1 & 0 \\ x_k & 0 \end{pmatrix}$$

Очевидно также, как этот прием можно распространить на рекуррентные соотношения более высокого порядка, например, $x_k = c_k + b_k x_{k-2} + a_k x_{k-1}$. Для этого достаточно ввести аналогичные матрицы P_k и X_k, только размерности, на единицу большей:

$$P_k X_{k-1} = X_k, \text{ где } P_k = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ c_k & b_k & a_k \end{pmatrix}, \; X_k = \begin{pmatrix} 1 \\ x_{k-1} \\ x_k \end{pmatrix}$$

Примеры программ

Умножение вектора на матрицу

Программа на языке RPC приведена ниже. Исходные данные и результат вычисления располагаются в РОП. Заметим, что в этой программе используется процедура vector(), рассмотренная выше.

```
/* ••••••••• ••••••• •• ••••••• */
#include <stdio.h>
#include <stdlib.h>
#include "seq.h"

#define VCoeff 16  /*  •••••••••• •••••••••••• ••••••••• */
#define MVLR 64  /*  ••••. ••••• •••••••, •••••••••••• • •• */
#define MVLL 64  /*  ••••. ••••• •••••••, ••••••••• •• ••    */

struct  sbrw
      { int n, m;
        int NR, NL, LeafSize; };
```

```
struct BP_vector
      { int k,l,n,m;
        int NR, LeafSize;
        float res[MVLR]; };

handleDM hV;              /*    •••••••• ••••••    [NL]      */
handleDM hM;              /*    •••••••• •••••••   [NL*NR]   */
handleDM hRes;            /*    •••••••••••••• ••••••  [NR]  */

CPP(subrow); CPP(calcsr); CPP(vector);

void main(int ac, char *av[], char *env[])
{ int i, j, NL, NR, LeafSize;
  NEW_PARAM (sbrw,bl1);

/* B••• •••••••••• NL, NR, LeafSize */
  if (ac<4)
  { printf("\nArguments required: Nlines Nrows LeafSize");
    return;
  }
  NL=atoi(av[1]); NR=atoi(av[2]);
  LeafSize=atoi(av[3]);

  if (NL<=0 || NR<=0 || LeafSize<=0)
  { printf("\nArguments must be > 0");
    return;
  }

  if ((long)sizeof(*Work)*NL*NR >= 0x10000l)
  { printf("\nSize of matrix is too big");
    return;
  }

  if (NULL==(Work=malloc(sizeof(*Work)*NL*NR)))
  { printf("\nNot enough memory");
    return;
  }

/* ••••••••••• • •••••• •••••••• ••••••• • •••••••
   • ••••••• ••••••••••• */
  G_Alloc (sizeof(float), NL, VCoeff, 0, &hV);
  G_Alloc (sizeof(float), NL*NR, VCoeff, 0, &hM);
  G_Alloc (sizeof(float), NR, VCoeff, 0, &hRes);
  Wait();


/* B••• •••••••• •••••• • ••••••••••• •• • ••• ••••••• */

  bl1.n=0; bl1.m=NR-1;
  bl1.NR=NR; bl1.NL=NL; bl1.LeafSize=LeafSize;
  H_Call (subrow,&bl1);


/* B•••• •••••••••••• •••••• */

} /* main */

Parallel (subrow,bp)
PARAM (sbrw,*bp);
{
  if ((((bp->m-bp->n+1)*bp->NL <= bp->LeafSize)||
```

```
        (bp->m==bp->n))&&(bp->m-bp->n+1 <= MVLR))
    { NEW_PARAM (BP_vector,bl2);
      bl2.k=0; bl2.l=bp->NL-1; bl2.n=bp->n; bl2.m=bp->m;
      bl2.NR=bp->NR; bl2.LeafSize=bp->LeafSize;
      H_Call(calcsr,&bl2);
      Store_1(bl2.res, &hRes, bp->n, bp->m-bp->n+1);
      Wait();
    }
    else
    { NEW_PARAM (sbrw,bl3);
      NEW_PARAM (sbrw,bl4);
      bl3.n=bp->n; bl3.m=(bp->m+bp->n)/2;
      bl3.NR=bp->NR; bl3.NL=bp->NL; bl3.LeafSize=bp->LeafSize;
      P_Call(subrow,&bl3);
      bl4.n=bl3.m+1; bl4.m=bp->m;
      bl4.NR=bp->NR; bl4.NL=bp->NL; bl4.LeafSize=bp->LeafSize;
      H_Call(subrow,&bl4);
      Wait();
    }
  } /* subrow */

  /* ********** ********** ********* **** */
  Parallel (calcsr,bp)
  PARAM (BP_vector,*bp);
  { int j;
    if (((bp->l-bp->k+1)*(bp->n-bp->m+1) <= bp->LeafSize||
         (bp->l==bp->k))&&(bp->l-bp->k+1 <= MVLL))
      H_Call(vector,bp); /* ******** bp ********* */
    else
    { NEW_PARAM (BP_vector,blp1);
      NEW_PARAM (BP_vector,blp2);
      blp1.k=bp->k; blp1.l=(bp->k+bp->l)/2;
      blp1.n=bp->n; blp1.m=bp->m;
      blp1.NR=bp->NR; blp1.LeafSize=bp->LeafSize;
      P_Call(calcsr,&blp1);
      blp2.k=blp1.l+1; blp2.l=bp->l;
      blp2.n=bp->n; blp2.m=bp->m;
      blp2.NR=bp->NR; blp2.LeafSize=bp->LeafSize;
      H_Call(calcsr,&blp2);
      Wait();
      for(j=0; j<=(bp->m-bp->n); j++)
        bp->res[j]=blp1.res[j]+blp2.res[j];
    }
  } /* calcsr */
```

Здесь предполагается, что при запуске через аргументы командной строки программа получает три параметра. Первые два задают размеры матрицы, последний параметр (LeafSize) управляет глубиной рекурсии, задавая объем вычислений в активации, по достижении которого дальнейшее деление задачи не производится. В качестве оценки трудоемкости здесь используется количество операций умножения, которые необходимо произвести для выполнения всех вычислений в данной параллельной ветви программы. Впрочем, как можно видеть из

приведенного текста, из этого правила существует одно исключение. А именно, если объем активации не превышает величины LeafSize, но размеры массивов для временного хранения данных (MVLL) и результатов (MVLR) недостаточны, дробление работы все же производится.

Следует также отметить, что передача результатов через блок параметров в приведенном тексте приводит к существенному расходованию стекового пространства, так как блоки параметров для вызова дочерних процессов в нашем примере отводятся на стеке. Это может привести к тому, что программа (особенно при работе в среде MS DOS) может завершиться аварийно по переполнению стека, если задать слишком маленький размер параметра LeafSize. Впрочем, избежать этого довольно просто, если отводить память под блоки параметров динамически – предлагаем это сделать читателю.

Вычисление множества Мандельброта

Программа с некоторыми упрощениями приведена ниже. Предполагается, что каждый модуль производит вычисления для изображения множества Мандельброта в графическом режиме 640*480, а по завершении процесса пересылает построенное изображение на ПМ с номером 0, который уже выводит полное изображение. Для хранения картинки используется статическая память (все изображение разбито на NArr кусков). Для простоты основные параметры, управляющие ходом вычислений, заданы в виде констант.

```
/* ********** ********* ************ */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <graphics.h>
#include <stdlib.h>
#include <time.h>
#include "seq.h"

#define ESC   27
#define NLIST 400
#define COL   80
#define Wi    8
#define Hi    96
#define Space 400
#define NArr  5

float MB [][4]={
   {-2.25, 0.75, -1.5, 1.5},
   {-0.19920, -0.12954, 1.01480, 1.06707},
   {-0.95, -0.88333, 0.23333, 0.3},
   {-0.713, -0.4082, 0.49216, 0.71429},
   {-1.781, -1.764, 0, 0.013},
   {-0.75104, -0.7408, 0.10511, 0.11536},
```

```
   };
float pmin, pmax, qmin, qmax;
int WasInit=0, Ind=0;
int maxi=640, maxj=480, Col=64, NPM, My, ListsHere=0;
float dp, dq, Max=100;

void quit(void);
void Init(void);
CPP(Calc);

struct BP  { int I,J; };

handleSM hA[NArr]; char *Arr[NArr];

void main(int argc, char *argv[], char *envp[])
{ NEW_PARAM(BP,blp); int i, j;

  if (argc>1)
    Ind=atoi(argv[1]);
  if (Ind>=sizeof(MB)/sizeof(MB[0])) Ind=0;
  pmin=MB[Ind][0]; pmax=MB[Ind][1];
  qmin=MB[Ind][2]; qmax=MB[Ind][3];
  printf("\n•••••••• •••••••••••• (%d) •••••••••:", Ind);
  printf("\npmin=%8.6f, pmax=%8.6f, qmin=%8.6f,
qmax=%8.6f",
    pmin,pmax,qmin,qmax);
  printf("\n••••••• ••••• •••••••");

  v_DS(); getch(); v_ES();

  dp=(pmax-pmin)/(maxi-1); dq=(qmax-qmin)/(maxj-1);

  for (i=0; i<NArr; ++i)
    S_Alloc(Space,COL,hA+i,&Arr[i]);
  Wait();
  for (i=0; i<NArr; ++i)
    for (j=0; j<COL; ++j)
      *(Arr[i]+j*Space+Space-2)=0;

  S_CopyTo_Set(-1,
               &pmin,sizeof(pmin), &pmax,sizeof(pmax),
               &qmin,sizeof(qmin), &qmax,sizeof(qmax),
               &dp,sizeof(dp), &dq,sizeof(dq),
               NULL);
  Wait();

  blp.I=0; blp.J=NLIST-1;
  H_Call(Calc,&blp);

/* •••••••••••• ••••• •••••••• */
  for (i=0; i<NArr; ++i)
     for (j=0; j<COL; ++j)
        if (*(Arr[i]+j*Space+Space-2))
        { v_DS();
          putimage(j*Wi,i*Hi,Arr[i]+j*Space,COPY_PUT);
          v_ES(); }
```

```
  v_DS();getch();v_ES();
}
#define P_(a) Bpar->a
Parallel(Calc,Bpar)
PARAM(BP,*Bpar);
{ NEW_PARAM(BP,blp1);
  NEW_PARAM(BP,blp2);
  float p, q, x, y, r, w;
  int i, j, c, ib, ie, jb, je;
  if (WasInit==0)
    { Init(); N_PM(&NPM, &My); }
  if (P_(J)==P_(I))
  { ib=Wi*(P_(I)%COL); ie=ib+Wi;
    jb=Hi*(P_(I)/COL); je=jb+Hi;
    for (i=ib; i<ie; i++)
    { for (j=jb; j<je; j++)
      { p=pmin+dp*i; q=qmin+dq*j; x=y=c=0;
        for (;;)
        { w=x*x-y*y+p; y=2*x*y+q; x=w; ++c; /* Iteration */
          r=x*x+y*y;
          if (r>Max)
          { v_DS(); putpixel(i,j,c%14+2); v_ES();
            break;}
          if (c>=Col)
          { v_DS(); putpixel(i,j,1); v_ES();
            break;}
        }
      }
      v_DS();
      while (kbhit())
        if (getch()==ESC) quit();
      v_ES();
    }
    j=P_(I)%COL; i=P_(I)/COL;
    v_DS(); getimage(ib,jb,ie-1,je-1,Arr[i]+j*Space);
v_ES();
    *(Arr[i]+j*Space+Space-2)=My;
    ListsHere++;
    S_CopyTo_1(hA+i,j,1,0);
    Wait();
  }
  else
  {
    blp1.I=P_(I); blp1.J=(P_(I)+P_(J))>>1;
    P_Call(Calc,&blp1); /* ••••• 1 */
    blp2.I=blp1.J+1; blp2.J=P_(J);
    H_Call(Calc,&blp2); /* ••••• 2 */
    Wait();
  }
  Wait();
```

```
} /* Calc */
void Init()
{
  /* ............ ........... ...... */
 ...
  WasInit=1;
}
void quit() /* ........ ......... ........ */
{ closegraph();
  exit(0);
}
```

Приложения

Приложение 1. Перечень операторов языка RPC

Предварительное объявление параллельной процедуры proc_name:

CPP(proc_name);

Описание заголовка параллельной процедуры:

Parallel(proc_name,ptr_bl_par)
PARAM(namestruct,*ptr_bl_par);

Объявление блока параметров в вызывающей процедуре:

NEW_PARAM(namestruct,bl_par);

Вызов параллельной процедуры proc_name с блоком параметров bl_par:

P_Call(proc_name,bl_par);
P_Send(proc_name,bl_par,pm);
H_Call(proc_name,bl_par);

Системный вызов синхронизации параллельных процессов:

Wait();

Системные вызовы доступа в общую память:

Load_1(char *la, handleDM far *pH, unsigned long gOff, int vl);
Store_1(char *la, handleDM far *pH, unsigned long gOff, int vl);
Load_V(char *la, handleDM far *pH, unsigned long gOff, int step, int vl);
Store_V(char *la, handleDM far *pH, unsigned long gOff, int step, int vl);
Load_I(char *la, handleDM far *pH, unsigned long gOff, int *iv, int vl);
Store_I(char *la, handleDM far *pH, unsigned long gOff, int *iv, int vl);
Load_M(char *la, handleDM far *pH, unsigned long gOff, int *mv, int vl);
Store_M(char *la, handleDM far *pH, unsigned long gOff, int *mv, int vl);
Load_G(char *la, handleDM far *pH, unsigned long gOff,
        unsigned m, unsigned n, int rinc,int cinc);

```
Store_G(char *la, handleDM far *pH, unsigned long gOff,
        unsigned m, unsigned n, int rinc,int cinc);
```

Динамическое выделение фрагмента памяти в ОП:

```
G_Alloc (unsigned SizeEl, unsigned long NumEl, unsigned VCoeff,
         unsigned FirstPM, handleDM far *pH);
```

Динамическое освобождение фрагмента памяти в ОП:

```
G_Free(handleDM far *pH);
```

Размещение статического массива данных:

```
S_Alloc (unsigned SizeEl, unsigned NumEl, handleSM *pH, void **pPtr);
```

Освобождение памяти из-под статического массива данных:

```
S_Free(handleSM *pH);
```

Операторы копирования статических данных:

```
S_CopyTo_1(handleSM *pH, unsigned Off, unsigned vl, int PM);
S_CopyFrom_1(handleSM *pH, unsigned Off, unsigned vl, int PM);
S_CopyTo_V(handleSM *pH, unsigned Off, int step, unsigned vl, int PM);
S_CopyFrom_V(handleSM *pH, unsigned Off, int step,unsigned vl, int
PM);
S_CopyTo_I(handleSM *pH, unsigned Off, int *iv, unsigned vl, int PM);
S_CopyFrom_I(handleSM *pH, unsigned Off, int *iv,unsigned vl, int PM);
S_CopyTo_M(handleSM *pH, unsigned Off, int *mv, unsigned vl, int
PM);
S_CopyFrom_M(handleSM *pH, unsigned Off, int *mv,unsigned vl, int
PM);
S_CopyTo_G(handleSM *pH, unsigned Off, unsigned m, unsigned n, int
rinc,
                int cinc, int PM);
S_CopyFrom_G(handleSM *pH, unsigned Off, unsigned m, unsigned n, int
rinc,
                int cinc, int PM);
```

Загрузка/выгрузка из статической области данных в сплошной массив ЛП:

```
S_ Load _1(void *la, handleSM *pH, unsigned Off, unsigned vl);
S_ Store _1(void *la, handleSM *pH, unsigned Off, unsigned vl);
S_ Load _V(void *la, handleSM *pH, unsigned Off, int step, unsigned vl);
S_ Store _V(void *la, handleSM *pH, unsigned Off, int step, unsigned vl);
S_ Load _I(void *la, handleSM *pH, unsigned Off, int *iv, unsigned vl);
S_ Store _I(void *la, handleSM *pH, unsigned Off, int *iv, unsigned vl);
S_ Load _M(void *la, handleSM *pH, unsigned Off, int *mv, unsigned vl);
S_ Store _M(void *la, handleSM *pH, unsigned Off, int *mv, unsigned vl);
S_ Load _G(void *la, handleSM *pH, unsigned Off, unsigned m, unsigned n,
          int rinc, int cinc);
```

```
S_ Store _G(void *la, handleSM *pH, unsigned Off, unsigned m, unsigned n,
        int rinc, int cinc);
```

Захват/освобождение замка:

```
Lock(l);
Unlock(l);
```

Определение количества ПМ и номера текущего модуля:

```
N_PM(int *pN, int *pMyN);
```

Установка запрета на выборку из дека:

```
Set_M();
```

Приложение 2. Назначение функциональных клавиш при работе в среде RPMSHELL

F1	Подсказка по назначению пункта меню
Alt-F1	Подсказка по назначению функциональных клавиш
F2	Сохранение конфигурации оболочки
F3	Выбор текстового файла для редактирования/просмотра
Alt-F3	Возврат с старому текстовому файлу
F4	Редактирование текущего текстового файла
Alt-F4	Просмотр текущего текстового файла
F5	Компиляция программы
Alt-F5	Переключение на пользовательский экран
F6	Запуск программы на выполнение
Alt-F6	Запуск программы под управлением отладчика
F7	Запуск программы моделирования
F8	Анализ результатов моделирования
F9	Задание имени графа трассы
Alt-F10	Выход из оболочки с сохранением состояния
Alt-O	Временный выход в DOS
Alt-X	Выход из оболочки

Приложение 3. Перечень файлов, образующих среду RPMSHELL

RpmShell.com	Резидентная часть пользовательской оболочки
Shell.exe	Подгружаемая часть пользовательской оболочки
Seq.h	Заголовочный файл для режима последовательного выполнения программы на RPC
Seq.lib	Библиотека для режима последовательного выполнения
Extend.exe	Препроцессор для работы в режиме последовательного выполнения с построением параллельной трассы
Trace.h	Заголовочный файл для режима построения трассы
Trace.lib	Библиотека для режима построения трассы
Ext_D_Ch.exe	Препроцессор для работы в режиме последовательного выполнения с проверкой корректности
D_Check.h	Заголовочный файл для режима проверки корректности
D_Check.lib	Библиотека для режима проверки корректности
Ext_Par.exe	Препроцессор для работы в режиме параллельного исполнения на локальной сети
Par.h	Заголовочный файл для режима параллельного исполнения на локальной сети
Par.lib	Библиотека для режима параллельного исполнения на локальной сети
Slave.exe, RunSlave.exe, Run_File.exe, SendFile.exe	Программы для рассылки файлов и запуска процессов при работе в параллельном режиме на локальной сети с использованием сетевых протоколов IPX/SPX
Testgr.exe	Программа для определения потенциального параллелизма RPC-программы
Ideal.exe	Программа визуализации результатов работы программы Testgr
Mod64.exe	Программа имитационного моделирования параллельной работы программы на рекурсивно-параллельной машине
Visual.exe	Программа визуализации результатов работы программы Mod64
Anal.exe	Программа анализа результатов программы Mod64 для определения причин снижения эффективности параллельной программы
Protocol.exe	Программа для представления результатов эксперимента в виде графиков и гистограмм
Viewer.exe	Программа для просмотра текстовых файлов

Vis_tr.exe	Программа для просмотра графа трассы
SyntRpc.exe	Программа для анализа исходного текста на соответствие правилам Rpc
Sed.exe	Простейший текстовый редактор (используется по умолчанию)

Кроме того, в состав программного комплекса входит ряд файлов, содержащих документацию и текст некоторых подсказок. Из них стоит специально упомянуть только файл Rpc_V02.doc, в котором содержится подробное описание языка RPC.

Литература

1. Бурцев В.С. О необходимости создания супер-ЭВМ в России //Информационные технологии и вычислительные системы. 1995. № 1. С. 5-11.
2. Левин В.К. Высокопроизводительные мультимикропроцессорные системы // Информационные технологии и вычислительные системы. 1995. № 1. С. 12-21.
3. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Maucheck, Vaidy Sunderam. PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press. 1994. 298 p.
4. Васильчиков В.В., Емелин В.В. Об одном подходе к разработке, исследованию и оптимизации параллельных алгоритмов и программ // Разработка, моделирование и оптимизация сложных информационных систем. Ярославль: ЯрГУ, 1993. С. 70-76.
5. Васильчиков В.В., Емелин В.П., Курчидис В.А., Маматов Ю.А., Рекурсивно-параллельное программирование и работа в среде RPMSHELL. Препринт № 24 ИПВТ РАН. Ярославль, 1994. 81 с.
6. Васильчиков В.В. О поиске ошибок параллельного выполнения RPC-программ // Моделирование и анализ информационных систем. Выпуск 2. Ярославль: ЯрГУ, 1994. С. 26-31.
7. Васильчиков В.В., Воронин А.В. Анализатор эффективности параллельного выполнения RPC-программ // Разработка, моделирование и оптимизация сложных информационных систем. Ярославль: ЯрГУ, 1993. С. 76-86.
8. Васильчиков В.В. Оценки эффективности рекурсивно-параллельного стиля программирования // Моделирование и анализ информационных систем. Выпуск 3. Ярославль: ЯрГУ, 1996. С. 3-14.

Оглавление

Васильчиков Владимир Васильевич

Средства параллельного программирования для вычислительных систем с динамической балансировкой загрузки

Редактор, корректор А.А. Аладьева
Компьютерная верстка И.Н. Ивановой

Лицензия ЛР № 020319 от 30.12.96.

Подписано в печать 23.10.2001. Формат 60x84/16.
Бумага тип. Усл. печ. л. 7,44. Уч.-изд. л. 6,1.
Тираж 75 экз. Заказ

Оригинал-макет подготовлен
редакционно-издательским отделом ЯрГУ.

Отпечатано на ризографе

Ярославский государственный университет.
150000 Ярославль, ул. Советская, 14.