Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Катков С. И.

СИСТЕМА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ СУПЕРПАСКАЛЬ: ЯЗЫК, ТРАНСЛЯТОР, ОТЛАДЧИК

Препринт
81

Новосибирск 2001

Система параллельного программирования, разработанная для языка СуперПаскаль, предназначена для отладки параллельных алгоритмов на последовательных персональных машинах. Ее рекомендуется использовать для обучения параллельному программированию, создания и отладки параллельных алгоритмов.

© Институт систем информатики им А. П. Ершова СО РАН, 2001

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Serge I. Katkov

THE SYSTEM OF PARALLEL PROGRAMMING SUPERPASCAL: LANGUAGE, TRANSLATOR, DEBUGGER.

Preprint
81

Novosibirsk 2001

The parallel programming system is based on the programming language SuperPascal. Using this system, a parallel algorithm can be debugged on PC. The system can be used for teaching parallel programming or creation and debugging of parallel algorithms.

© A. P. Ershov Institute of Informatics Systems, 2001

1. ЯЗЫК СУПЕРПАСКАЛЬ

1.1. Публикационный язык

В начале 60-х гг. произошло разделение языков на публикационные, подчеркивающие ясность идеи, и реализационные [1, 2]. Известный ученый Пер Бринч Хансен уверен, что параллельные компьютеры не будут широко использоваться до тех пор, пока не будет принят единый язык программирования для публикаций параллельных алгоритмов [3]. Необходимо заметить, что Паскаль исторически является публикационным языком для последовательных алгоритмов.

В статье [3] Пер Бринч Хансен формулирует результаты своего трехлетнего опыта по работе с параллельными алгоритмами и делает следующие выводы о будущем научных параллельных вычислений:

- Общецелевые параллельные компьютеры ближайшего будущего будут мультикомпьютерами с множеством процессоров, имеющих только локальную память. Они автоматически будут поддерживать работу по передаче сообщений между двумя процессорами. Топологию вычислительной системы может изменять программист, используя виртуальные каналы. Таким образом, параллельные компьютеры позволят программистам работать в терминах конфигураций процессов и не будет нужды отображать эти конфигурации на фиксированную архитектуру.
- Традиционные задачи в вычислительной математике могут эффективно решаться с использованием детерминированных параллельных вычислений.
- Параллельные алгоритмы могут быть описаны на элегантном публикационном языке и проверены на последовательном компьютере. Затем эти алгоритмы могут быть переписаны на реальный язык с реальной архитектурой. Алгоритм же может быть опубликован как текст программы, написанный на публикационном языке. Такие программы могут служить моделями для тех, кто хочет изучить эти алгоритмы. В этих программах каждая деталь рассмотрена, объяснена и проверена.

По мнению Пер Бринч Хансена публикационный язык для параллельного программирования должен иметь следующие свойства [3]:

- быть расширением широко используемого стандарта языка с детерминированным параллелизмом и средствами обмена сообщениями. Расширение должно быть определено в духе стандарта языка;
- позволять программе создавать произвольную конфигурацию параллельных процессов, связанных каналами. Конфигурации могут определяться итеративно или рекурсивно и создаваться динамически.

СуперПаскаль – это язык параллельного программирования, предложенный Пер Бринч Хансеном для публикации вычислительных алгоритмов [3]. Он является расширением Паскаля с помощью конструкций для создания детерминированных параллельных процессов и для связи синхронизирующими сообщениями. Язык позволяет создавать неограниченные комбинации рекурсивных процедур и параллельных конструкций, однако существуют ограничения на использование переменных и некоторых конструкций языка Паскаль.

Параллельные средства СуперПаскаля сводятся к двум механизмам — образования параллельных процессов и обмена сообщениями.

1.2. Образование параллельных процессов

Параллельные процессы образуются и запускаются двумя операторами — *Parallel* и *Forall*. Результатом работы оператора *Parallel* $S_1 \mid S_2 \mid ... \mid S_n$ *end* будет параллельное выполнение процессов S_1, S_2, ... , S_n до тех пор, пока каждый из них не закончится. Данный оператор позволяет выполнять параллельно различные типы алгоритмов, но такая концепция приемлема лишь в случае небольшого числа процессов, например, уже при ста процессах это не очень практично.

Для разработки параллельных алгоритмов в мультипроцессорной системе необходимо средство, позволяющее работать с переменным (как правило неизвестным) или большим числом идентичных процессов.

Оператор *Forall i:=* E_1 *To* E_2 *Do S* обозначает массив параллельных однотипных процессов (**array of parallel processes**), называемых процессами-элементами (**element processes**), и индексами процессов (**process indicies**) с соответствующими предельными значениями. Нижняя и верхняя границы определяются выражениями E_1 и E_2. Каждый индекс соответствует отдельному процессу, определяемому значением *i* и оператором *S*. Оператор *Forall* также ожидает окончания работы всех экземпляров оператора *S*. Например, оператор *Parallel s(x) | Send (x,value) end (*где x — некоторый ка-

нал) производит запуск процесса s и одновременно передает в канал значение.

Оператор *Forall i := 1 To N do x[i] := a[i]* соответствует оператору присваивания в векторных вычислительных системах.

1.3. Организация связи между процессами

Связь между параллельными процессами осуществляется посредством каналов (***channel***). Для достижения синхронизации и корректности пересылки сообщений организация связи должна удовлетворять следующим трем условиям:

1) канал может передавать в определенный момент времени только одно сообщение между двумя параллельными процессами;
2) перед связью процесс выбирает канал, направление пересылки и тип сообщения;
3) пересылка совершается, когда один процесс готов послать сообщение некоторого типа по определенному каналу, а другой готов принять сообщение этого же типа по этому же каналу;

Канал динамически идентифицируется переменной, называемой ссылкой на канал (***channel reference***), или канальной переменной (***channel variable***). Для работы с каналами был введен новый конструктор типов, имеющий следующий синтаксис:

*Type T = * (T_1, T_2, ..., T_n);*

Значения типа *T* — это неупорядоченное множество канальных ссылок. Каждая канальная ссылка типа *T* обозначает канал, по которому могут передаваться только сообщения типов $T_1, T_2, \ldots, T_n$ (***message types***). Соответственно, следуя правилам языка, можно определить переменную этого типа. В качестве примера рассмотрим описание

```
Type channel = * ( integer, boolean );
Var left, right : channel;
```

Здесь определяется новый тип *channel* и две переменные этого типа — *left* и *right*. Значение переменных — это ссылка на канал, который может пересылать сообщения типа *integer* и *boolean*.

Для работы с каналами введены следующие операции над канальными переменными:

- процедура *Open* — открытие канала и инициализации канальной переменной;
- процедура *Send* — послать сообщение в канал;

- процедура *Receive* — принять сообщение из канала;
- присваивание;
- операции сравнения на равенство и неравенство.

Значение переменной ***v*** типа канал не определено, пока не выполнено *Open(v)* или не было присваивания *v := c,* где ***c*** определено. *v* и *c* должны быть ссылками одинаковых типов. Допускается запись *Open* (v_1, v_2, ... , v_n), которая эквивалентна *Begin Open(* v_1 *); Open(* v_2,..., v_n*) end.*

Типы переменных v_1, v_2, ... , v_n могут быть различными.

Канал существует до тех пор, пока программа не завершит свое исполнение.

Передача сообщения между двумя процессами осуществляется процедурами *Send* и *Receive*.

Send (b, e) означает пересылку значения выражения *e* в канал, определяемый выражением *b*. Выражение *b* должно быть каналом типа *T*, а выражение *e* — сообщением типа из *T*. Аналогично *Receive (c, v)* означает прием значения из канала, определяемого выражением *c* в переменную *v*. Выражение *c* — канал типа *T*, тип переменной *v* — сообщение типа из *T*.

Передача сообщения при помощи вышеописанных процедур состоится при следующих условиях:

- выражения *b* и *c* одного типа *T* и определяют один канал;
- выражение *e* и переменная *v* одного типа, который является сообщением типа из *T* (см. описание *T*).

Выполнение операции передачи прерывает данный процесс, пока другой процесс не будет готов выполнить операцию приема с согласованными параметрами, и наоборот.

После завершения связи посылающий и принимающий процессы продолжают свое выполнение независимо.

Допускается запись *Send (b,* e_1, e_2,..., e_n*)*, которая эквивалентна

Begin Send(b, e_1*); Send(b,* e_2,..., e_n*) end*

Аналогично, *Receive(* c, v_1, v_2,..., v_n*)* эквивалентно

Begin Receive(c, v_1*); Receive(* c, v_2,..., v_n*) end*

Логическое выражение *v = c* истинно, если значения *v* и *c* определены и ссылаются на один и тот же канал, и ложно в противном случае.

Выражение *v <> c* эквивалентно *Not (v = c).*

Во время исполнения программы могут возникать следующие ошибки (***run-time error***), связанные с организацией передачи сообщений:

- неопределенная ссылка на канал (***undefined channel reference***): выражение-параметр не определяет канал;

- несовместимость (конфликт) каналов (***channel contention***): два параллельных процесса пытаются послать (получить) сообщение по одному каналу одновременно;
- неправильный тип сообщения (***message type error***): два параллельных процесса пытаются связаться по каналу, но типы передаваемого значения и выходной переменной не совпадают.

1.4. Контроль несовместимости (interference control)

Относительные скорости асинхронизации параллельных процессов, вообще говоря, не известны. Ошибки, зависящие от асинхронности процессов, называются ***time-dependent***. А процессы, порождающие их, — конфликтующими, или несовместимыми. Суть ошибки заключается в том, что два процесса могут одновременно использовать одну и ту же часть глобальной памяти, и когда один процесс изменяет эту память, то другой, в зависимости от времени использования этой памяти, возможно, получает неверное значение из этой памяти. Когда программа с временнозависимыми ошибками выполняется повторно с одними и теми же входными параметрами, результаты, как правило, различны и непредсказуемо отличаются друг от друга. Невоспроизводимое поведение затрудняет определение конфликтной ситуации систематическим тестированием. Наиболее оптимальный вариант решения проблемы — введение ограничений, которые запрещают ситуации, порождающие конфликты. Соблюдение этих ограничений должно контролироваться компилятором.

Контроль заключается в том, что для каждого процеса строится множество изменяемых (***target***) и множество только используемых переменных (***expression variables***). Вместе они определяют контекст переменных (***variable context***) процесса *S*. По контексту переменных для каждого процесса можно проверить совместимость. Очевидно, что следующее правило исключает возможность возникновения ошибки, связанной с совместным использованием переменных: изменяемые переменные одного процесса не могут быть изменяемыми или используемыми переменными другого.

Использование таких типов данных, как массивы, порождает некоторую неопределенность. Изменение всего лишь одного элемента относит все множество к числу изменяемых переменных и, следовательно, делает невозможным построение параллельных процессов, оперирующих с общими структурами данных. Для обработки таких ситуаций применяется директива [***sic***] (см. пример 1). Компилятор воспринимает эту директиву как утвер-

ждение правильности следующего параллельного оператора и опускает контроль над использованием переменных.

Пример 1. Директива sic.

```
[sic] { 1<=k<m }
Parallel
    Receive (up, u[0,k]) |
    Send (down, u[m,k]) |
    Receive (left, u[k,0]) |
    Send (right, u[k,m])
end
```

В примере 1 при условии *1<=k<m* конфликтная ситуация не возникает и директива [***sic***] уведомляет об этом компилятор. Аналогично с использованием оператора *Forall* :

```
    [sic]
Forall i:=1 To p-1 Do Open(c[i])
```

Для обеспечения корректной работы параллельных процессов Бринч Хансен предлагает следующие ограничения на синтаксические конструкции Паскаля:

- исключение указателей;
- исключение оператора *goto* (соответственно отсутствуют метки);
- исключение процедурных и функциональных параметров;
- исключение объявления *forward*;
- рекурсивные процедуры и функции не используют неявные параметры.

Кроме этих основных ограничений Пер Бринч Хансен ввел еще множество мелких. Учитывая то, что всякая имитация параллельного исполнения на последовательном компьютере уже достаточно трудоемка, время собственно трансляции не является столь критичным для нашей системы. Исходя из этого мы решили дополнить систему компонентой, которая за счет достаточно глубокого анализа проверяла бы наличие или отсутствие конфликтов параллелизма без введенного Пер Бринч Хансеном множества ограничений. Эта компонента в настоящее время разрабатывается, а сам входной язык системы освобожден от дополнительных ограничений.

1.5. Примеры программ на языке СуперПаскаль

Для иллюстрации языка приведем несколько алгоритмов:

1) проверки простоты натурального числа (два метода);
2) нахождения обратной матрицы;
3) решения системы линейных уравнений методом циклической редукции;
4) сортировки массива (два метода).

Для каждого из примеров в конце данной работы приведен полный текст программы.

1.5.1. Проверка простоты натурального числа

Требуется проверить, является ли натуральное число *a* простым? Для наглядности примера считаем, что $a <= p$.

Алгоритм заключается в том, что параллельно выполняются *p* процессов, которые проверяют, делится ли нацело *a* на *i*, где *i* — порядковый номер. Результат получается как логическое умножение всех проверок. Каждый параллельный процесс получает от правого соседа число *a* и, зная свой порядковый номер, проверяет делимость нацело, а затем результат передается правому соседу. Функция, проводящая тесты, описывается как

procedure test(a:integer; seed:integer; var composite:boolean)

Результатом работы процедуры является проверка, можно ли нацело разделить значение *a* на значение *seed*. При успешном завершении теста в *composite* заносится *true*, в противном случае — *false*. Параллельные вычисления организованы как кольцевая сеть, включающая процесс *master* и процессы *pipeline* (конвейер), связанные двумя каналами связи (рис. 1).

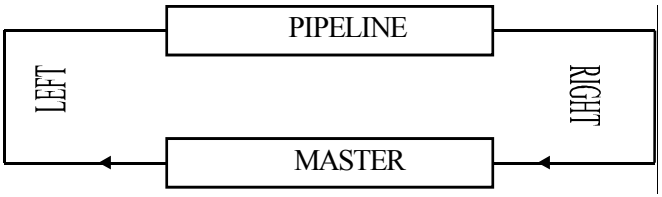


Рис. 1

Данная конструкция реализуется следующим образом:

*Type channel = *(integer, boolean);*

```
procedure ring( a:integer; var prime:boolean)
    var left,right:channel;
    begin
      open( left, right);
      parallel
        pipeline( left, right) |
        master( a, prime, left, right)
      end;
    end;
```

На рис. 2 *pipeline* представляет собой *p* параллельных узлов, соединенных между собой *p+1* каналами.

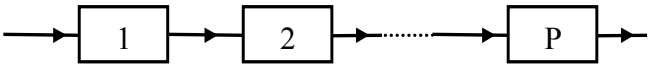


Рис. 2

В свою очередь конструкция *pipeline* может быть реализована двумя способами. Первый способ — это процедура, использующая оператор *Forall*:

```
procedure pipeline( left, right: channel)
    type row = array  [0..p]  of  channel;
    var c:row; i:integer;
    begin
      c[0]:=left; c[p]:=right;
      for i:=1 to p-1 do
        open( c[i]);
      forall i:=1 to p do
        node( i, c[i-1], c[i]);
    end;
```

С другой стороны, совместное использование рекурсии и концепции параллельных процессов позволяет создавать достаточно интересные алгоритмы. Еще один вариант процедуры *pipeline* доказывает это утверждение:

```
    procedure pipeline( min,max:integer; left,right:channel)
    var middle:channel;
    begin
      if min<max then
        begin
```

```
      open( middle);
      parallel
        node( min, left, middle)|
        pipeline( min+1, max,
          middle, right)
      end
    end
  else
    node( min, left, right)
end;
```

В обоих случаях процедура *node* является узловым процессом, в котором собственно и происходит проверка *test* и который может быть реализован следующим образом:

```
procedure node( i:integer; left,right:channel)
    var a:integer; j:integer;
      composite:boolean;
    begin
     receive( left, a);
     if i<p then
       send( right, a);
     test( a, i, composite);
     send( right, composite);
     for j:=1 to i-1 do
       begin
         receive( left, composite);
         send( right, composite);
       end;
    end;
```

1.5.2. Нахождение обратной матрицы

Пусть дана матрица $\boldsymbol{A}$ и необходимо найти матрицу $\boldsymbol{X}$ такую, что $\boldsymbol{AX} = \boldsymbol{E}$, где $\boldsymbol{E}$ — единичная матрица. Для решения данной задачи поставим эквивалентную задачу $\boldsymbol{Ax_i} = \boldsymbol{e_i}$, где $\boldsymbol{x_i}$ — столбцы матрицы $\boldsymbol{X}$, а $\boldsymbol{e_i}$ — единичные векторы. Эквивалентную задачу решим методом Гаусса, для чего сначала приведем матрицу $\boldsymbol{A}$ к верхнетреугольному виду, проводя аналогичные преобразование с матрицей $\boldsymbol{E}$. Данный процесс можно осуществлять параллельно. Для каждой строчки матрицы $\boldsymbol{A}$ образуем свой узел-процесс. Каждый узел связан каналами с предыдущим узлом (строчкой) и последующим.

Входными параметрами, как и выходными, являются соответствующие строчки матриц ***A*** и ***E***.

```
Type   vect = array[1..N] of real;
         Channel = *(vect);
for i:=1 to N-1 do open(c[i]);
c[0]:=nil;c[N]:=nil;
forall i:=1 to N do b[i]:=Node(i,c[i-1],c[i],A[i],E[i]);
```

Таким образом, мы используем топологию процессов, называемую кольцевой сетью (см. рис 1, 2), т.е. процессы последовательно соединены каналами. Первый узел ***n-1*** раз посылает свою строку правому соседу. Остальные процессы производят следующие операции: принимают от левого соседа вектор, затем процесс, если он не последний, отправляет этот вектор дальше правому соседу, далее находит первый ненулевой элемент в пришедшем векторе и зануляет у себя этот элемент, используя операции умножения вектора на число и сложение векторов. После этого процесс (если он не последний) отправляет свой вектор правому соседу. Если первый ненулевой элемент не найден, тогда матрица вырождена и обратной матрицы не существует.

```
function Node(i:integer;cl,cr:Channel;var a,f:vect):boolean;
var j,k:integer;
   va,vf:vect;
   kfc:real;
begin
  Node:=true;
  for j:=1 to i-1 do
    begin
     receive(cl,va,vf); приняли вектор слева
     if i<N then send(cr,va,vf); отправили направо
     k:=1;
     while (k<=N) AND (va[k]=0) do k:=k+1; нашли 1 ненулевой
     if k=N+1 then Node:=false если нет, то матрица вырождена
     else
      begin
       kfc:=a[k]/va[k];
       mul(va,-kfc);
       mul(vf,-kfc);
       add(a,va);
       add(f,vf); занулили элемент
```

```
    end
  end;
  if i<N then send(cr,a,f); отправили свой вектор дальше
end;
```

Таким образом, имеем почти верхнетреугольную матрицу ***A.*** Приводим ее к верхнетреугольному виду обычной сортировкой строк, проводя последовательно соответствующие изменения с матрицей ***F (преобразованная E).*** Теперь осталось параллельно решить систему $Ax_i = f_i$, где A — верхнетреугольная матрица.

```
forall i:=1 to N do NodeTri(i,A,F,X);
```

Так как A — верхнетреугольная, то все процессы независимы и никакой связи между ними нет, значит, нет необходимости использовать каналы.

1.5.3. Метод циклической редукции решения системы линейных уравнений

Пусть дана система линейных уравнений вида $-a_i x_{i-1} + b_i x_i - c_i x_{i+1} = f_i$, где матрица неразложима, a_i, b_i, c_i положительны и есть диагональное преобладание. Требуется найти решение этой системы линейных уравнений. Для применения метода циклической редукции выразим x_i:

$$x_i = (f_i + a_i x_{i-1} + c_i x_{i+1})/b_i.$$

Подставим x_{i+1} и x_{i-1} по этой формуле:

$$-a_i' x_{i-2} + b_i' x_i - c_i' x_{i+2} = f_i',$$

где

$$a_i' = (a_i a_{i-1})/b_{i-1},$$
$$b_i' = -(a_i c_{i-1})/b_{i-1} + b_i - (c_i a_{i+1})/b_{i+1},$$
$$c_i' = (c_i c_{i+1})/b_{i+1},$$
$$f_i' = (a_i f_{i-1})/b_{i-1} + f_i + (c_i f_{i+1})/b_{i+1}.$$

Продолжая такие подстановки, придем к системе вида

$$-a_i' x_0 + b_i' x_i - c_i' x_{n+1} = f_i',$$

где будем считать, что $x_0 = x_{n+1} = 0$, таким образом, получим решение $x_i = f_i'/b_i'$

Заметим, что при пересчете коэффициентов для каждой неизвестной используется значения только «соседних» коэффициентов. Таким образом, можно для каждой неизвестной создать свой процесс, который будет получать значения коэффициентов для пересчета от своих соседей. Для организации связи создадим матрицу каналов:

```
Type  vect=array [1..N] of real;
      Channel=*(real);
      Arrch=array [1..N] of Channel;
      Matr=array [1..N] of Arrch;
Var ch: Matr;

for i:=1 to N do
   for j:=i+1 to N do
      open(ch[i,j]);
  for i:=1 to N do
    for j:=1 to i-1 do
      ch[i,j]:=ch[j,i];
```

Далее создаем процессы:

```
forall i:=1 to N do
   x[i]:=Node(i,a[i],b[i],c[i],f[i],ch[i]);
```

Процесс *Node* описан следующим образом:

```
function Node(i:integer;a,b,c,f:real;ch:Arrch):real;
```

Каждый процесс выполняет $log_2 n$ раз пересчет коэффициентов

```
for s:=1 to K do.
```

Для пересчета каждый процесс сначала обменивается значениями коэффициентов с соседями:

```
if i-sh<1 then
     begin al:=0;bl:=0;cl:=0;fl:=0;end
    else
     receive(ch[i-sh],al,bl,cl,fl);
    if i+sh<=N then
     send(ch[i+sh],a,b,c,f);

    if i+sh>N then
     begin ar:=0;br:=0;cr:=0;fr:=0;end
    else
     receive(ch[i+sh],ar,br,cr,fr);
    if i-sh>0 then
     send(ch[i-sh],a,b,c,f);
```

Здесь *sh* — расстояние до соседа на текущей итерации. Далее производится пересчет коэффициентов. По окончании процесс возвращает значение решения.

Схема процессов и связей между ними на первых трех шагах представлена на рис. 3.

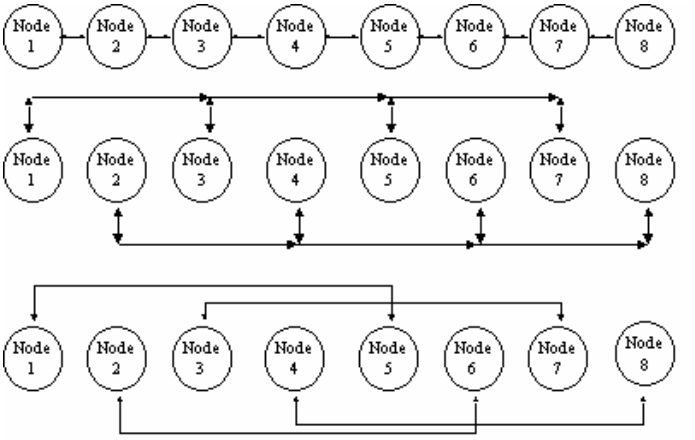


Рис.3

1.5.4. Сортировка массива (два метода)

Для иллюстрации реализации рекурсивного метода сортировки массива целых чисел на параллельных вычислительных системах приведем пример программы на языке СуперПаскаль. Алгоритм заключается в том, что массив на каждой итерации делится пополам, а в листах получившегося дерева производится сортировка методом пузырька. При возврате отсортированных частей массива происходит их слияние. Таким образом, используется топология дерева вычислительных процессов. Видно, что алгоритм не использует каналы.

```
procedure Master(k:integer;Var v:vect;l:integer);      к — текущий уровень
Var v1,v2:vect;
   i,j:integer;
begin
  if k<Depth then              Depth — глубина деления массива
   begin
    for i:=1 to l do      деление массива
```

```
    if i<=(l div 2) then v1[i]:=v[i]
    else v2[i-(l div 2)]:=v[i];
  parallel
    Master(k+1,v1,l div 2)|
    Master(k+1,v2,l div 2)
  end;
  sliv(v1,v2,v,l); слияние отсортированных частей
 end
else
 begin
ортировка методом пузырька
end;
end;
```

Второй метод иллюстрирует применение топологии гиперкуба. Организовывается система каналов, реализующая гиперкуб (рис.4):

```
Type  vect=array [1..N] of integer;
    Channel=*(integer,vect);
    ArrChan=array [1..12] of Channel;      гиперкуб размерности 3
```

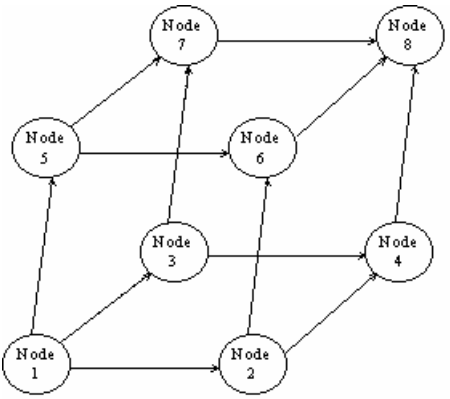


Рис.4

Происходит запуск вершин гиперкуба, и в первую его вершину передается исходный вектор и его длина:

```
Parallel
  Forall i:=1 to 8 do
    Node(i,c[i],c[i+1],c[i+2],c[i+4]);
begin
    Send(c[1],v,N);
    Receive(c[1],v);
  end
```

Далее каждая вершина гиперкуба производит операции деления массива пополам, сортировку своей части и слияние своей и вернувшейся частей:

```
procedure Node(k:integer;c1,c2,c3,c4:Channel);
описание вспомогательных переменных
begin
  Receive(c1,v,Len); принимаем вектор и его длину
  if k<2 then        через k определяется условная глубина вершины гипер-
куба. От нее зависит, сколько раз вершина будет делить массив и переда-
вать ее дальше.
    begin
      for i:=1 to Len div 2 do
        v1[i]:=v[i+(Len div 2)];
      Len:=Len div 2;
      Send(c2,v1,Len);
    end;
Аналогично для вершин глубины 1 и 2

сортировка методом пузырька своей части массива

  if k<5 thenприем и слияние с чужой частью
    begin
      Receive(c4,v2);
      v1:=v;
      Len:=Len*2;
      sliv(v1,v2,v,Len);
    end;

Send(c1,v);  Возврат массива на верхний уровень
end;
```

2. СИСТЕМА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Система программирования, разработанная для языка СуперПаскаль, состоит из двух основных компонент, одна из которых выполняет функции транслятора, а другая — отладки с визуализацией параллельного исполнения. Трансляция осуществляется путем конвертирования программ из языка СуперПаскаль в язык ТурбоПаскаль.

Все параллельные конструкции СуперПаскаля транслируются конвертором в описание имитирующих паскалевских объектов либо в вызов процедур динамической поддержки, которые имитируют параллелизм СуперПаскаля соответствующим квазипараллелизмом. Для реализации квазипараллелизма существуют очередь активных процессов (один из которых является исполняемым) и очередь пассивных процессов. Очередь активных процессов представляет собой множество процессов, которые в данный момент готовы к исполнению. На этом множестве определен линейный порядок, который определяет очередность попадания процесса на процессор. В начальный момент исполнения программы очередь активных процессов содержит единственный процесс ***MAIN*** и пополняется двумя путями: либо создается новый процесс, либо у процесса из очереди пассивных процессов исчезают причины находиться там. Процесс покидает список активных процессов в одном из следующих случаев:

- ожидает сообщение, которое еще не послано;
- посылает сообщение;
- создал процессы-сыновья и ожидает завершения их работы;
- завершает свое исполнение.

Во всех случаях, кроме последнего, процесс переходит в очередь пассивных процессов, а в последнем случае — просто исчезает из очереди. Если исполняемый процесс покидает очередь активных процессов, то новый исполняемый процесс определяется из этой очереди согласно выбранному в ней линейному порядку.

Все средства имитации параллельных конструкций СуперПаскаля (представление канальных типов и процедуры реализации квазипараллелизма) связаны в специальный модуль на языке ТурбоПаскаль. Реализация этого модуля соответствует MS DOS. Для изменения платформы достаточно изменить реализацию части его процедур.

Существенной частью любой системы параллельного программирования является отладчик, который бы давал представление о процессе парал-

лельного исполнения. Соответствующая компонента системы для СуперПаскаля построена так, что исполняемая программа может накапливать историю параллельного исполнения, а вызываемый по запросу пользователя процессор визуализирует эту накопленную историю. История параллельного исполнения накапливается в специальном протокольном файле. Действия по его построению включаются в объектную программу на ТурбоПаскале при специальном режиме трансляции, задаваемом пользователем. В протокольном файле отражаются все события, связанные с параллельным (и только с параллельным) исполнением. К таким событиям относятся все операции с каналами (открытие, посылка сообщения, запрос на получение сообщения и его удовлетворение) и с процессами (выполнение параллельного оператора, создание процесса, его уничтожение, переключение процессов и завершение исполнения программы). Протокольный файл содержит всю необходимую информацию о событии (значение сообщений, идентификационные номера процессов, номера соответствующих строк исходной программы на языке СуперПаскаль).

После завершения программы история параллельного исполнения может быть визуализирована. В ней выделяется два типа информации — дерево параллельных процессов и состояние каналов.

Дерево параллельных процессов представляет иерархию существующих (или существовавших) процессов, вершина дерева соответствует или процессу, или параллельной команде исходной программы (в последнем случае она специальным образом помечается). Вершина, соответствующая процессу, обладает атрибутом, соответствующим состоянию процесса (текущий процесс, активный ожидающий процесс, процесс, порождающий потомка, ждущий приема посланного сообщения, ждущий посылки сообщения в свой канал, ожидающий завершения работы потомков, уничтоженный процесс). Состояние процессов при визуализации отображается различными цветами.

Состояние каналов — это список всех открытых каналов с указанием того, задействован ли канал передачей сообщения и кем, а если задействован, — то каково значение передаваемого сообщения (если оно послано). Текущий статус канала как раз и соответствует информации о том, какова текущая исполняемая команда над каналом, что за процесс, работающий с ним, и каково значение сообщения в канале.

Параллельный отладчик, а точнее - визуализатор истории параллельного исполнения, на основании протокольного файла и исходной программы визуализирует всю накопленную к данному моменту информацию. Эта визуализация происходит либо пошагово, либо при движении по выбранным

параллельным событиям. При пошаговой визуализации движение происходит по последовательности тактов параллельного исполнения, где такт соответствует текущему параллельному событию. Произвольное движение происходит по выбранным пользователем событиям, отраженным в протокольном файле.

Для акта визуализации выделяются четыре окна — для дерева процессов, для состояния каналов (всего или текущего статуса канала), для скроллинга исходной программы, где текущая команда выделяется цветом, и для скроллинга последовательности параллельных событий. Текущее состояние экрана при работе отладчика иллюстрируется в приложении 5.

Таким образом, отладчик позволяет получить полное и наглядное представление о процессе параллельного исполнения. Представление обо всех исполнениях программ можно получить привлечением стандартного отладчика системы ТурбоПаскаль, но уже, естественно, в терминах ТурбоПаскаля.

2.1. Конвертор

Конвертор — одна из основных частей системы — это транслятор, для которого входной информацией является текст программы на языке СуперПаскаль. Результат работы конвертора — текст программы, написанный на языке ТурбоПаскаль [4].

Транслятор состоит из пяти основных компонент: лексического, синтаксического, контекстного анализаторов, анализатора параллельных конструкций и генератора. Учитывая специфику транслятора, можно было бы организовать контекстный анализатор с наименьшими затратами, положившись на компилятор ТурбоПаскаля. С другой стороны концепция СуперПаскаля такова, что очень важен контроль над типами. Для большей уверенности в правильности программы наряду с параллельными конструкциями обрабатываются и большинство других структур.

Формат вызова транслятора выглядит следующим образом:

Sp [-d] [-b]<infile>[outfile],

где

<infile> — входной файл, имя программы на СуперПаскале;

[outfile] — выходной файл, имя программы на ТурбоПаскале (необязательный параметр, значение по умолчанию — Result.pas);

[-d] — опция отладки, означающая, что в программу на ТурбоПаскале необходимо включить код, отвечающий за сохранение отладочной информации (необязательный параметр, по умолчанию отключен);

[-b] — вывод сообщений конвертора в формате Borland IDE(необязательный параметр, по умолчанию отключен).

Использование опции отладки подробно будет описано в п.2.2. Использование опции *–b* позволяет внедрить конвертор в среду *TurboPascal,* для чего необходимо проделать следующие операции:

- в среде *TurboPascal* в меню *Options* выбрать пункт *Tools*;
- в появившемся окне нажать кнопку *New;*
- заполнить все поля, за исключением *Command line,* по своему усмотрению; поле *Command line* должно содержать следующую строчку:

  -b $MEM(64)$EDNAME$SAVE ALL$CAP MSG(pipe.exe)

Теперь можно использовать конвертор путем нажатия выбранной «горячей клавиши» в среде *TurboPascal*.

Для корректной работы конвертора необходимо, чтобы в текущей директории находился файл *system.*, а для использования конвертора в среде *TurboPascal* необходим файл *pipe.exe,* поставляемый вместе с конвертором.

Важный момент, который надо отметить, — это работа с внешними модулями. Конвертор не поддерживает модули, написанные пользователем на языке СуперПаскаль, но можно использовать внешние модули. Для этого их необходимо подключить к программе, используя стандартное слово *uses*, и создать одноименный файл без расширения с описанием всех используемых в нем процедур, функций, констант, переменных и т.д. Примером такого файла может служить файл *system.*

2.1.1. Модуль эмуляции параллелизма

Интерфейс модуля параллелизма приведен в приложении 6. Структура, приведенная в нем, описывает канал. Опишем значение полей:

typ — номер типа пересылаемого сообщения. Нумерация ведется в соответствии с описанием типа канала, т.е. в описании *Type T=*(T1,...,Tn) T1* — это первый тип и т.д. Положительное значение поля соответствует тому, что в канал помещено сообщение, а отрицательное значение — тому, что из канала хотят получить сообщение;

p — хранилище пересылаемой информации;

who — идентификационный номер процесса, обращающегося к каналу;

next — переменная для организации списка.

Таким образом, следующие предложения СуперПаскаля трактуются как

Type T=(T1,...,Tn);* –> *Type T=SP_Channel;*

Var ch : T; −> *Var ch : T;*

Процедура открытия канала выглядит как

Open(cp); −> *SP_Open(ch);*

а смысл ее — в выделении памяти под канал и инициализация его. Одновременно производится проверка на то, чтобы канал дважды не открывался.

Процедуры *SP_Begin* и *SP_Exit* соответственно инициализируют и завершают работу эмуляции параллелизма и выполняются в начале и в конце работы программы.

Теперь рассмотрим функции пересылки сообщений. Операторы трактуются следующим образом:

Send(ch,x); −> *SP_Send(ch,ADDR(x),SizeOf(x),ТИП(x));*
Receive(ch,x); −> *SP_Receive(ch,ADDR(x),SizeOf(x),ТИП(x));*

здесь *ТИП(x)* — значение, которое будет помещено в переменную *typ*;

При исполнении оператора *SP_Send* реализуется следующий алгоритм:

проверяется, существует ли канал *ch*:
если он не существует, то выдается сообщение об ошибке;
иначе проверяется, послал ли уже кто-либо туда сообщение:
Если да, то выдается сообщение об ошибке;
Если же кто-то ждет получения сообщения,
то проверяются на совпадение типы посылаемого и принимаемого сообщения. В случае удачи происходит передача сообщения, и канал освобождается. В противном случае выдается сообщение об ошибке.

Если же канал не занят, то передаваемое сообщение копируется в ящик *p*, в поле *who* сохраняется номер процесса, пославшего сообщение, а в поле *typ* — номер типа сообщения.

При исполнении оператора *SP_Receive* реализуется следующий алгоритм:

Проверяется, существует ли канал *ch*:
если он не существует, то выдается сообщение об ошибке;
иначе проверяется, ждет ли уже кто-либо оттуда сообщение:
Если да, то выдается сообщение об ошибке;
Если же кто-то послал туда сообщения,
то проверяются на совпадение типы посылаемого и принимаемого сообщения. В случае удачи происходит передача сообщения, и канал освобождается. В противном случае выдается сообщение об ошибке.

Если же канал не занят, то в поле *who* сохраняется номер процесса, ожидающего сообщение, а в поле *typ* — номер типа сообщения.

Теперь о параллелизме. Модуль эмуляции параллелизма поддерживает две очереди процессов (см. приложение 6): очередь активных процессов, ожидающих выполнения, и процессов, ожидающих каких-либо событий. Первая — это традиционная очередь процессов, возникающая в операционной системе при нехватке процессоров. Наша очередь организована по принципу *FIFO* (*First Input First Output*), хотя ее нетрудно изменить. Вторая очередь (пассивных процессов) организуется в результате синхронизации процессов. В модуле эмуляции параллелизма существует специальная подпрограмма, переключающая процессы. Оператор вида

Parallel S1|S2|...|Sn end;

конвертор заменяет следующей конструкцией:

```
If SP_Fork Then
  Begin
    S1;
    SP_Endthread;
  End;
.....
.....
If SP_Fork Then
  Begin
    Sn;
    SP_Endthread;
  End;
SP_Wait;
```

Функция *SP_Fork* создает процесс, сохраняет его окружение, помещает его в очередь активных процессов и возвращает значение *Ложь*. Когда же функция получает управление как процесс, то возвращает значение *Истина*. Процедура *SP_Endthread* в свою очередь уничтожает процесс и переключает следующий процесс из очереди активных процессов на процессор. *SP_Wait* — это процедура, которая помещает родительский процесс в очередь процессов ожидания, сохраняет количество его потомков и переключает следующий процесс из очереди активных процессов на процессор. Аналогичным способом реализован оператор *Forall*:

Forall i:=Ex1 To Ex2 Do S(i);

заменяется на

```
    For i:=Ex1 To Ex2 Do
  If SP_Fork Then
   Begin
     S(i);
     SP_Endthread;
   End;
   SP_Wait;
```

Заметим, что переменная *i* здесь локальная переменная, и это требование удовлетворяется конвертором.

В заключение хотелось бы отметить три факта:

- Для изменения платформы, на которой выполняется программа, необходимо заменить лишь модуль эмуляции параллелизма, т. е. ту часть, которая отвечает за переключение процессов.
- Конвертор имеет опцию "транслировать, с включением отладочной информации". В таком случае программа во время своего исполнения будет сохранять информацию о параллельных событиях в специальном файле протокола.
- Одним из способов отладки программы является использование различных способов организации очереди процессов на выполнение. В случае правильно работающей программы, результат не должен изменяться при изменении способа организации очереди.

2.1.2. Сообщения об ошибках

Перечислим ошибки, которые фиксируются во время исполнения параллельной программы (*run-time error*):

- ситуация *DEADLOCK*, пуста очередь активных процессов. Это означает, что все процессы находятся в ожидании синхронизации;
- в операциях с каналами используется несуществующий (неоткрытый) канал;
- при операции синхронизации наблюдается несовпадение типов между принимаемым и посылаемым значением;
- при операции посылки или приема значения из канала обнаруживается, что канал уже занят такой же операцией;
- переполнение поддерживаемого числа процессов (каждому процессу присваивается уникальный идентификационный номер - число типа *longint,* в системе *TurboPascal* не больше 2147483647);

2.2. Отладчик

2.2.1. Назначение отладчика

Предлагаемый отладчик представляет собой визуализатор выполнения параллельной программы. Отладчик реализован в среде *WINDOWS*, что позволяет использовать все графические возможности системы. Здесь хотелось бы отметить, что отладчик предназначен для отладки собственных программ, т. е. пользователь должен представлять себе, какую топологию вычислителя он хотел получить и как должна работать его программа. С помощью отладчика пользователь может получить информацию о том, как в действительности исполнялась его программа и какую в действительности топологию он построил в своей программе. Для успешной работы с отладчиком необходимо приобрести некоторые навыки, что, по нашему мнению, не сложно. На данный момент отладчик предназначен для работы с относительно небольшими параллельными программами. Ограничение связано с размером дерева процессов, которое хранится в локальной «куче». С точки зрения реализации это, наверное, главный недостаток. Однако, так как система предназначена для отладки алгоритмов, можно предположить, что пользователю не потребуются большие деревья процессов. Например, для отладки алгоритма проверки простоты целого числа достаточно проверить работу алгоритма на простых числах не более тридцати, что успешно позволяет сделать наш отладчик.

На данном этапе ведутся работы по увеличению количества визуализируемых элементов, что позволит пользователю получать более подробную информацию.

2.2.2. Концепции отладчика и принцип работы

Отладчик использует две основные концепции:

- восприятия только параллельных событий;
- посмертной отладки (***Mortal Debugging***).

Первое означает, что отладчик не обращает внимание на последовательные участки программы, чтобы не отвлекать пользователя от параллелизма программы. Для отладки последовательных участков программы можно использовать стандартный отладчик *TurboPascal* , но в таком случае вы лишаетесь возможности визуализации параллелизма. Вторая концепция означает, что отладка ведется после окончания работы программы, с использованием накопленной информации о ходе исполнения программы.

Для накопления информации о ходе исполнения программы необходимо конвертировать текст программы, используя опцию отладки -d. Тогда при исполнении программы информации будет помещаться в специальный файл протокола (см. далее). Для использования отладчика необходимо иметь исходный файл программы, файл протокола и файл описания каналов, который создается во время конвертирования программы с опцией отладки.

2.2.3. Файл протокола

Файл протокола — это информация о выполнении параллельной программы, которая собирается во время выполнения программы. Если программа, написанная на языке СуперПаскаль, оттранслирована с опцией "отладочная информация", то каждое параллельное событие во время исполнения программы будет отражено в протокольном файле. Основное предназначение файла протокола — это входная информация для отладчика. Подробнее рассмотрим параллельные события:

- *Открытие канала*. В этом случае в протокольный файл помещается следующая строка:

  LnOmNp,

  где *n* — это номер строки в исходном тексте программы, *m* — уникальный номер канала и *p* — номер в списке каналов из файла их описания.
- *Сообщение посылается в канал*. В протокольный файл помещается следующая информация:

  LnSmTp
  Uq
  value**,**

  где *n* — номер строки в исходном тексте программы, *m* — номер канала, куда посылается сообщение, *p* — номер типа, *q* — длина сообщения, *value* — содержание посылаемого сообщения.
- *Послан запрос на получение сообщения из канала*. Здесь получаем

  LnQmTp,

  где *n* —номер строки в исходном тексте программы, *m* — уникальный номер канала, а *p* — номер типа.
- *Получен ответ на запрос*. Теперь в файле протокола получаем

  LnRmTp,

  где *n* —номер строки в исходном тексте программы, *m* — уникальный номер канала, а *p* — номер типа.

- *Получена команда на распараллеливание процесса*. В зависимости от оператора распараллеливания получаем
  LnP — для оператора Parallel,
  LnF — для оператора Forall,
  где *n* — номер строки в исходном тексте программы.
- *Создание нового процесса*. Тогда в протокольный файл помещается следующая информация:
  LnNm,
  где *n* — номер строки в исходном тексте, а *m* — уникальный номер создаваемого процесса.
- *Переключение процессов*. При переключении процессов в файл протокола заносится строка
  nHm,
  здесь процесс с уникальным идентификационным номером *n* помещается в очередь ожидания события, а процесс *m* попадает на исполнение (возможное префиксное **W** означает, что «засыпает» отец).
- *Уничтожение процесса*. Дает нам следующую строку в протокольном файле:
  DnHm,
  где *n* — номер уничтожаемого процесса, а процесс *m* попадает на исполнение.
- *Ошибка параллельного исполнения программы*. Результатом этого события будет сообщение
  ERROR.
- *Конец выполнения программы*. Результатом этого события будет сообщение
  END.

Таким образом, все параллельные события отображаются в файл протокола, который затем может быть использован отладчиком.

2.2.4. Внешний вид отладчика и его возможности

Интерфейс отладчика (см. приложение 5) содержит четыре окна. Первое окно представляет графическое изображение управляющего дерева параллельного исполнения программы. Управляющее дерево является наиболее важной частью отображаемой отладчиком информацией. Каждая вершина дерева соответствует либо некоторому процессу, либо содержит метки *PARALLEL* и *FORALL*, позволяющие разделить два последовательных распараллеливания, например: два подряд идущих оператора *Parallel*. Каждая

вершина в зависимости от своего состояния изображается определенным цветом. Всего в отладчике используется семь цветов, обозначающих соответственно:

- текущий процесс;
- ожидание запуска на исполнение;
- создание потомка;
- процесс послал сообщение и находится в состоянии синхронизации;
- процесс ожидает сообщение и находится в состоянии синхронизации;
- процесс ожидает завершение работы своих потомков;
- процесс уничтожен или не существует.

Все цвета пользователь может изменять (пункт меню *Option*|*Color*). Управляющее дерево можно изображать как целиком, так и частично (пункт меню *Option*|*Full Tree*), т. е. только ту часть, которая в текущий момент существует. Текущий момент в данном случае означает текущий шаг. Дело в том, что отладчик делит временное пространство на такты. Каждый такт соответствует параллельному событию (см. 2.2.3). В качестве дополнительного сервиса, предоставляемого пользователю, существует возможность выбрать шрифт (пункт меню *Option*|*Font*), которым печатаются метки вершин. В частности, таким образом пользователь может менять размер дерева, что позволит ему увидеть больше вершин одновременно. В этом окне пользователь, нажав правую кнопку мыши (пункт меню *Run*|*Step*), может потактно просматривать исполнение программы. В меню также содержатся команды:

- загрузки отлаживаемой программы (пункт меню *File*|*Load*);
- выгрузки отлаживаемой программы (пункт меню *File*|*Clear*);
- перезагрузки отлаживаемой программы (пункт меню *Run*|*Restart*);
- управления остальными окнами (пункт меню *Window*);
- выхода из программы (пункт меню *File*|*Exit*).

Второе окно содержит текст исходной программы. Здесь пользователь также может менять шрифт текста (пункт меню *Option*|*Font*). Текущая исполняемая строка в этом окне выделяется цветом и изображается в центре окна. Окно исключительно информационное.

Третье окно содержит список параллельных событий. Хотелось бы отметить, что отладчик можно представить себе как интерпретатор, программой для которого является файл протокола. Третье окно и содержит эту программу в удобном для пользователя виде. Здесь пользователь может менять шрифт текста (пункт меню *Option*|*Font*), а также может, нажав

дважды левую кнопку мыши, указать отладчику, что нужно выполнить все такты до текущего события.

Четвертое окно — это окно состояния каналов. Состояние каналов — это список всех каналов, для каждого из которых указано его состояние (занят или нет). Если канал занят, то, нажав правую кнопку мыши, можно посмотреть, кем он занят и каково содержимое передаваемой информации.

СПИСОК ЛИТЕРАТУРЫ

1. Forsythe G.E. *Algorithms for scientific computing* // Communs. ACM. — 1966. — Vol. 9. — P. 255—256.
2. Perlis A.J. *A new policy for algorithms?* //Communs. ACM. — 1966. — Vol. 9. — P. 255—256.
3. P. Brinch Hansen *SuperPascal - a publication language for parallelscientific computing* // Concurency: Practical and Experience. — 1994. — Vol.6б, N 5. — P. 461—483.
4. *ТурбоПаскаль 5.0: Описание языка*. Ч.1. — Б.М., 1990.

ПРИЛОЖЕНИЕ 1

Полный текст программы проверки простоты натурального числа.

```
Program testbh;
Type
        channel=*(Integer,Boolean);
Var
         prime:Boolean;
        x:Integer;
Procedure test(a:Integer;seed:Integer;Var composite:Boolean);
Begin
If ((a mod seed =0) and (a <> seed) and (seed<>1))Then
        composite:=true
Else
        composite:=false;
End;
Procedure master(a:Integer;Var prime:Boolean;Var left,right:channel);
Var
        i:Integer;
        composite:Boolean;
Begin
Send(left,a);
prime:=false;
i:=1;
While i<=a Do
        Begin
        i:=i+1;
        Receive(right,composite);
        If (composite) then
                prime:=true;
        End;
End;
Procedure node(i:Integer;Var left,right:channel);
Var
        a:Integer;
        j:Integer;
        composite:Boolean;
Begin
Receive(left,a);
```

```
If i<a Then
        Send(right,a);
test(a,i,composite);
Send(right,composite);
j:=1;
While j<=i-1 Do
        Begin
        j:=j+1;
        Receive(left,composite);
        Send(right,composite);
        End;
End;
Procedure pipeline(min,max:Integer;Var left,right:channel);
Var
        middle:channel;
Begin
If min<max Then
        begin
        Open(middle);
        Parallel
                node(min,left,middle)|
                pipeline(min+1,max,middle,right)
                End
        End
Else
        node(min,left,right)
End;
Procedure ring(a:Integer;Var prime:Boolean);
Var
        left,right:channel;
Begin
Open(left,right);
Parallel
        pipeline( 1, a, left,right) |
        master(a,prime,left,right)
        End
End;

Begin
```

```
read(x);
ring(x,prime);
If not prime Then
        writeln('X is a prime integer')
Else
        writeln('X is not a prime integer');
End.
```

П Р И Л О Ж Е Н И Е 2

Полный текст программы нахождения обратной матрицы.

```
Program obrat;
Const N=3;
Type
        vect=Array [1..N] Of Real;
        matr=Array [1..N] Of vect;
        Channel=*(vect);
Var
        A,F,X:matr; {Исходные данные}
Procedure mul(var v:vect;m:Real);
Var
        i:Integer;
Begin
For i:=1 To N Do v[i]:=v[i]*m;
End;
Procedure add(var v1,v2:vect);
Var
        i:integer;
Begin
Ffor i:=1 To N Do v1[i]:=v1[i]+v2[i];
End;
Function cmp(x,y:vect):Boolean;
Var
        I,j:integer;
Begin
i:=1;
j:=0;
While (i<=N) and (j=0) Do
        Begin
        If abs(x[i])>abs(y[i]) Then
                j:=1
        else
                If abs(x[i])=abs(y[i]) Then
                        j:=0
                Else
                        j:=-1;
        i:=i+1
        End;
```

```
If j=-1 then
        cmp:=TRUE
else
        cmp:=False;
End;
Procedure sort(Var A,F:matr);
Var
        i,j:integer;
        tmp:vect;
Begin
For i:=1 To N-1 Do
        For j:=N DownTo i+1 Do
                If cmp(a[i],a[j]) Then
                        Begin
                        tmp:=a[j];
                        a[j]:=a[i];
                        a[i]:=tmp;
                        tmp:=f[j];
                         f[j]:=f[i];
                         f[i]:=tmp;
                        End
End;
Procedure NodeTri(k:Integer;A:matr;F:matr;Var x:matr);
Var
        i,j:integer;
Begin
For i:=N Downto 1 Do
        Begin
        For j:=N DownTo i+1 do
                F[i,k]:=F[i,k]-A[i,j]*x[j,k];
        X[i,k]:=F[i,k]/A[i,i];
        End
End;
Function Node(i:Integer;cl,cr:Channel;Var a,f:vect):Boolean;
Var
        j,k:integer;
        va,vf:vect;
        kfc:Real;
Begin
```

```
Node:=true;
For j:=1 To i-1 Do
        Begin
        Receive(cl,va,vf);
        If i<N Then
                Send(cr,va,vf);
        k:=1;
        While (k<=N) and (va[k]=0) Do
                        k:=k+1;
        If k=N+1 Then
                Node:=false
        Else
                Begin
                kfc:=a[k]/va[k];
                mul(va,-kfc);
                mul(vf,-kfc);
                add(a,va);
                add(f,vf);
                End
        End;
If i<N Then
        Send(cr,a,f);
End;
Function Master:Boolean;
Var
        i,j:Integer;
        c:Array [0..N] Of Channel;
        b:Array [1..N] Of Boolean;
        r:Boolean;
Begin
Master:=true;
For i:=1 To N Do
For j:=1 To N Do
If i=j Then
        F[i,j]:=1
Else
        F[i,j]:=0;
For i:=1 To N-1 Do
        open(c[i]);
```

```
c[0]:=nil;c[N]:=nil;
Forall i:=1 To N Do
        b[i]:=Node(i,c[i-1],c[i],A[i],F[i]);
r:=true;
For i:=1 To N Do if not b[i] Then
        r:=false;
If r then
        begin
        sort(A,F);
        Forall i:=1 To N Do NodeTri(i,A,F,X);
        End
Else
        Master:=false
End;
Procedure myread;
Var
        i,j:integer;
Begin
    Ffor i:=1 To N Do
        For j:=1 To N Do
                read(A[i,j]);
End;
Procedure mywrite;
Var
        i,j:integer;
Begin
    writeln('X:');
Ffor i:=1 To N Do
        begin
        For j:=1 To N Do
                write(X[i,j]);
        writeln('');
        End;
End;

Begin
myread;  {Ввод данных}
If master Then  {Вычисление}
        mywrite; {Вывод данных}
End.
```

ПРИЛОЖЕНИЕ 3

Полный текст программы решения системы линейных уравнений.

```
    program reduction;
{Метод прогонки для лин.ур-ний с трех диагон.матр.}
{Матрица A - неразложима, a,б,c-положительны(-a*x[i-1]+b*x[i]-
c*x[i+1]=f)}
{и есть диагональное преобладание}
    Const
            N=4; {Кол-во неизвестных}
            K=2; {K=log2(N)}
    Type
            vect=Array [1..N] Of Real;
            Channel=*(Real);
            Arrch=Array [1..N] Of Channel;
            Matr=Array [1..N] Of Arrch;
    Var
            a,b,c,f,x:vect; {Исходные данные}
    Function Node(i:Integer;a,b,c,f:Real;ch:Arrch):Real;
    Var
            s,sh:Integer;
            al,bl,cl,fl,ar,br,cr,fr:Real;
            an,bn,cn,fn:Real;
            tmp:Real;
    Begin
    sh:=1;
    For s:=1 To K Do
            Begin
            {Обмен данными с соседями}
            If i-sh<1 Then
                    Begin
                        al:=0;bl:=0;cl:=0;fl:=0;
                        End
            Else
                    Receive(ch[i-sh],al,bl,cl,fl);
            If i+sh<=N Then
                    Send(ch[i+sh],a,b,c,f);
            If i+sh>N then
                    Begin
```

```
                    ar:=0;br:=0;cr:=0;fr:=0;
                    End
            Else
                    Receive(ch[i+sh],ar,br,cr,fr);
            If i-sh>0 Then
                    Send(ch[i-sh],a,b,c,f);
{Пересчет коэффициентов}
            If al*a=0 Then
                    an:=0
            Else
                    an:=al*a/bl;
            If a*cl=0 Then
                    tmp:=0
            Else
                    tmp:=a*cl/bl;
            If c*ar=0 Then
                    bn:=b-tmp
            Else
                    bn:=b-tmp-c*ar/br;
            If cr*c=0 Then
                    cn:=0
            Else
                    cn:=cr*c/br;
            If a*fl=0 Then
                    tmp:=0
            Else
                    tmp:=a*fl/bl;
            If c*fr=0 Then
                    fn:=f+tmp
            Else
                    fn:=f+tmp+c*fr/br;
            a:=an;b:=bn;c:=cn;f:=fn;
            sh:=sh*2;
            End;
    Node:=f/b;
    End;
    Procedure Master;
    Var
            ch:matr;
```

```
        i,j:integer;
Begin
For i:=1 To N Do
        For j:=i+1 To N Do
                Open(ch[i,j]);
    For i:=1 To N Do
        For j:=1 To i-1 Do
                ch[i,j]:=ch[j,i];
Forall i:=1 To N Do
        x[i]:=Node(i,a[i],b[i],c[i],f[i],ch[i]);
End;
Procedure myread;
Var
        i:integer;
Begin
For i:=1 To N Do
        Begin
        read(a[i]);
        read(b[i]);
        read(c[i]);
        read(f[i]);
        End;
End;
Procedure mywrite;
Var
        i:integer;
Begin
For i:=1 To N Do
        writeln(x[i]);
End;

Begin
myread;  {Ввод данных}
master;  {Вычисление}
mywrite; {Вывод данных}
End.
```

ПРИЛОЖЕНИЕ 4

Полный текст программы сортировки массива.
Топология дерева:

```
Program sort;
Const
        N=16;
        DIM=3;
Type
        vect=Array [1..N] Of Integer;
Var
        v:vect;
Procedure sliv(v1,v2:vect;Var v:vect;l:Integer);
Var
        i,j,k:Integer;
Begin
i:=1;j:=1;k:=1;
While k<=l Do
        Begin
        If i>(l div 2) Then
                Begin
                v[k]:=v2[j];
                j:=j+1;
                End
        Else
                If j>(l div 2) Then
                        Begin
                        v[k]:=v1[i];
                        i:=i+1;
                        End
                Else
                        If v1[i]<v2[j] Then
                                Begin
                                v[k]:=v1[i];
                                i:=i+1
                                End
                        Else
                                Begin
                                v[k]:=v2[j];
```

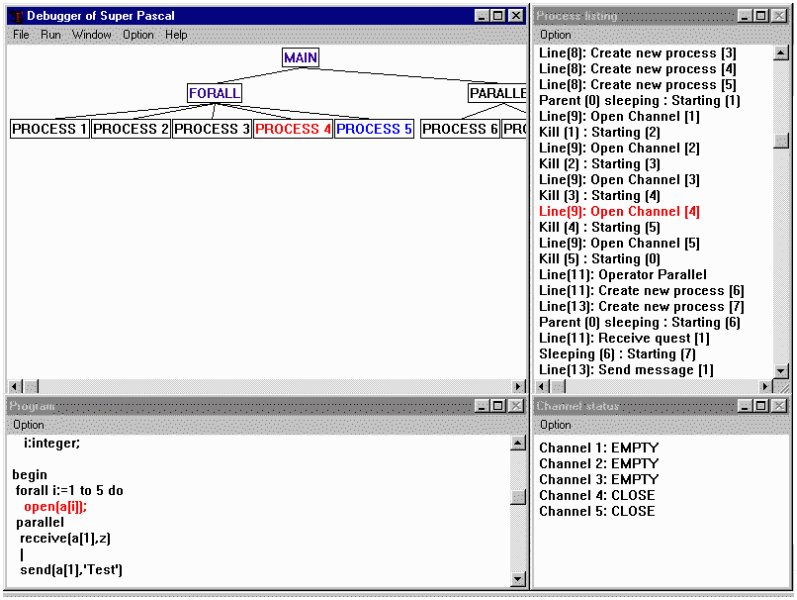
```
                                        j:=j+1
                                        End;
        k:=k+1;
        End;
End;
Procedure myswap(Var x,y:Integer);
Var
        tmp:Integer;
Begin
    tmp:=x;
x:=y;
y:=tmp;
End;
Procedure Master(k:Integer;Var v:vect;l:Integer);
Var
        v1,v2:vect;
        i,j:Integer;
Begin
If k<DIM Then
        begin
        For i:=1 To l Do
                If i<=(l div 2) Then
                        v1[i]:=v[i]
                else
                        v2[i-(l div 2)]:=v[i];
        Parallel
                Master(k+1,v1,l div 2)|
                Master(k+1,v2,l div 2)
                end;
        sliv(v1,v2,v,l);
        End
Else
        Begin
        For i:=1 To l-1 Do
                For j:=l DownTo i+1 do
                        If v[i]>v[j] Then
                                myswap(v[i],v[j]);
            End;
End;
```

```
Procedure myread;
Var
        i:Integer;
Begin
For i:=1 To N Do
        v[i]:=N+1-i;
End;
Procedure mywrite;
Var i:Integer;
Begin
For i:=1 To N Do
        writeln(v[i]);
End;

Begin
myread;
master(0,v,N);
mywrite;
End.
```

ПРИЛОЖЕНИЕ 5

Внешний вид сообщений отладчика.



П Р И Л О Ж Е Н И Е 6

Интерфейс модуля параллелизма..

```
Interface
Type
        sp_channel = ^sp_chann;
sp_chann = Record
        num :Iinteger;
        message : Pointer;
        who : Longint;
        next: sp_channel;
        End;
Typ = Word;

Procedure sp_open(Var channel:sp_channel);
Procedure sp_send(channel:sp_channel;src:Pointer;size:Word;t:Typ);
Procedure
sp_receive(channel:sp_channel;src:Pointer;size:Word;t:Typ);
Procedure sp_setindex(channel:Pointer;value:Longint);
Function  sp_fork:Boolean;
Procedure sp_wait;
Procedure sp_exit;
Procedure sp_begin;
Procedure sp_endthread;
```

С. И. Катков

СИСТЕМА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ СУПЕРПАСКАЛЬ: ЯЗЫК, ТРАНСЛЯТОР, ОТЛАДЧИК

Препринт
81

Рукопись поступила в редакцию 07.02.01
Рецензент В. А. Евстигнеев
Редактор Л. А. Карева

Подписано в печать 01.03.01
Формат бумаги 60 × 84 1/16 Объем 2.7 уч.-изд.л., 3 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6