М.В. Якобовский
Е.Ю. Кулькова

РЕШЕНИЕ ЗАДАЧ НА МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ

Москва
2004

УДК 681.324.012:519.6

Якобовский М.В., Кулькова Е.Ю. Решение задач на многопроцессорных вычислительных системах с разделяемой памятью. - М.: Станкин, 2004.- 30 с.

Учебное пособие посвящено вопросам использования многопроцессорных вычислительных систем с общей памятью. На примере алгоритма численного интегрирования одномерной функции рассматриваются вопросы построения эффективных параллельных алгоритмов, обеспечения защиты разделяемых переменных и обеспечения динамической балансировки загрузки процессоров.

Для специалистов по математическому моделированию, аспирантов и студентов, использующих в своей практике многопроцессорные системы.

Объем – 2 а.л.

Введение

На примере задачи вычисления определенного интеграла от аналитически заданной функции рассматриваются алгоритмы, минимизирующие время решения задачи на многопроцессорной вычислительной системе. Их использование обеспечивает снижение времени решения задачи относительно «наилучшего» доступного последовательного алгоритма. Определение эффективности параллельного алгоритма относительно собственной же последовательной версии, как правило, не корректно. Обладающий высокой эффективностью параллельный алгоритм может по времени выполнения проигрывать лучшему из доступных последовательных алгоритмов решения той же задачи. Далее рассматриваются алгоритмы, минимизирующие именно время решения задачи, оперирующей относительно небольшим объемом данных. Применяемые подходы могут быть полезны при решении широкого круга проблем.

Интегрирование одномерной функции на многопроцессорной системе с общей памятью

В качестве модельной задачи рассматривается проблема вычисления с точностью ε значение определенного интеграла (1):

$$J(A,B)=\int_A^B f(x)dx, \tag{1}$$

Пусть на отрезке [*A,B*] задана равномерная сетка, содержащая *n+1* узел:

$$x_i = A + \frac{B-A}{n} i, \quad i = 0,\cdots,n \tag{2}$$

Тогда, согласно методу трапеций, можно численно найти определенный интеграл от функции $f(x)$ на отрезке [A,B]:

$$J_n(A,B)=\frac{B-A}{n}\left(\frac{f(x_0)}{2}+\sum_{i=1}^{n-1} f(x_i)+\frac{f(x_n)}{2}\right) \tag{3}$$

Будем полагать значение *J* найденным с точностью ε, если выполнено условие (4):

$$\left|J_{n1}-J_{n2}\right| \le \varepsilon \left|J_{n2}\right|,\ n_1 > n_2, \tag{4}$$

где *n1* и *n2* - количество узлов, образующих две разные сетки.

Последовательные алгоритмы

Метод трапеций

Рассмотрим алгоритм, реализующий метод трапеций:

```
IntTrap01(A,B)
{
n=1
J2n=(f(A)+f(B))(B-A)/2

do
{
Jn= J2n

n=2n

s=f(A)+f(B)

for(i=1;i<n;i++)
       s+=2f(A+(B-A)i/n);

J2n=s(B-A)/n;
}
while(|J2n- Jn|≥ε J2n)

return J2n
}
```

Алг. 1. Последовательный алгоритм "метод трапеций"

Алгоритм 1 неэффективен в силу ряда причин, среди которых выделим две:

1. в некоторых точках значение подынтегральной функции вычисляется более одного раза. Например, в точке *A* функция будет вычислена *k* раз, где *k* - число выполнений цикла *do - while*;
2. на всем интервале интегрирования используется равномерная сетка, тогда как число узлов сетки на единицу длины на разных участках интервала интегрирования, необходимое для достижения заданной точности, зависит от вида функции *f (x)*. Например, число точек, необходимых для достижения заданной точности при определении интеграла (6) возрастает при $A \to 0$, несмотря на уменьшения длины отрезка интегрирования (рис. 1, таб. **1**).

Модифицируем алгоритм, с тем что бы избежать указанных недостатков.

Метод рекурсивного деления

В дальнейшем будем рассматривать интегрирование функции (5), вид которой приведен на Рис. 1:

$$f(x)=\frac{1}{x^2}\sin^2\left(\frac{1}{x}\right),\quad 0<A<<1 \tag{5}$$

Точное значение интеграла (1) от функции (5) дается выражением (6):

$$J(A,B)=\int_A^B \frac{1}{x^2}\sin^2\left(\frac{1}{x}\right)dx=-\frac{1}{2x}+\frac{1}{4}\sin\left(\frac{2}{x}\right)\Bigg|_A^B$$

$$J(A,B)=\frac{1}{4}\left(2\frac{B-A}{AB}+\sin\left(\frac{2}{B}\right)-\sin\left(\frac{2}{A}\right)\right) \tag{6}$$

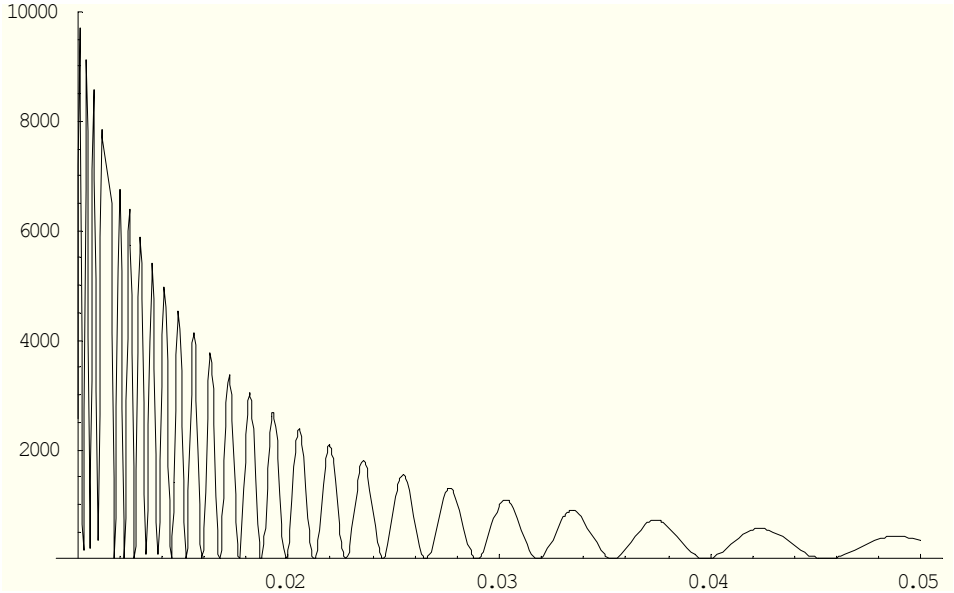

Рис. 1. График *f(x)*

Таблица 1

Результаты вычисления интеграла на разных отрезках

A	B	Npoints	eps real	time, c
0.00001	0.0001	1 553 568 289	-2.77E-11	434.55
0.0001	0.001	1 726 123 903	1.90E-10	470.99
0.001	0.01	360 075 831	2.05E-11	74.12
0.01	0.1	79 973 845	-2.22E-12	16.44
0.1	1	105 108 653	8.67E-11	21.42
1	10	396 149	-6.00E-11	0.094
10	100	412 331	-6.30E-11	0.096

Как уже было указано, использование для вычислений непосредственно метода трапеций неэффективно, поскольку предполагает равномерное сгущение сетки на всем отрезке интегрирования, без учета характера изменения подынтегральной функции. Построим алго

ритм, свободный от указанного недостатка, для чего воспользуемся простым соотношением (7):

$$J(A,B) = J(A,C) + J(C,B), \quad C = \frac{A+B}{2} \tag{7}$$

Разобьем интервал интегрирования на две части и независимо проинтегрируем функцию *f(x)* на каждой из них, выбрав соответствующие шаги интегрирования. Процедуру разбиения отрезков можно рекурсивно повторить до получения отрезков, на которых подынтегральная функция имеет простой вид и может быть аппроксимирована отрезком прямой с заданной точностью. Таким образом, выигрыш от применения рассмотренного алгоритма рекурсивного разбиения (Алг. 2) достигается за счет возможности использования сеток с разным числом узлов на разных участках интервала интегрирования.

```
main()
{
J= IntTrap03(A,B,(f(A)+f(B))*(B-A)/2)
}

IntTrap03(A,B,fA,fB)
{
J=0

C=(A+B)/2

fC=f(C)

sAB=(fA+fB)*(B-A)/2

sAC=(fA+fC)*(C-A)/2
sCB=(fC+fB)*(B-C)/2

sACB=sAC+sCB

if(|sAB-sACB|≥ε|sACB|)

      J=    IntTrap03(A,C,fA,fC)+
            IntTrap03(C,B,fC,fB)
else

      J=    sACB

return J
}
```

Алг. 2. Последовательный алгоритм "рекурсивного деления"

При наличии достаточного количества процессоров и нулевого времени порождения параллельных процессов можно было бы непосредственно запускать пары подпрограмм *IntTrap03* для обработки половинок разбиваемого интервала. Но, поскольку в реальной системе число процессоров ограничено и время, необходимое для порож

дения параллельных процессов существенно не равно нулю, такой подход неэффективен. Запуск процессов, число которых значительно превышает число физических процессов системы при решении рассматриваемой задачи, приводит к увеличению времени ее решения по сравнению с последовательной программой. Следует разработать метод, позволяющий распределить вычислительную работу между ограниченным числом процессов. Основная сложность построения параллельного алгоритма на основе рассмотренного последовательного алгоритма, кроется в том, что, хотя несколько ветвей программы могли бы одновременно обрабатывать разные части интервала интегрирования, координаты концов этих отрезков хранятся в программном стеке процесса (они попадают туда в момент вызова процедуры *IntTrap03*) и фактически недоступны программисту. Если бы они были расположены в некотором явно описанном массиве, можно было бы передать их по частям для обработки разным нитям программы.

Рассмотрим соответствующий алгоритм - метод локального стека. Алгоритм использует массив данных, доступ к которым осуществляется по принципу стека - первым удаляется последний из добавленных элементов (соответствующие процедуры приведены в листинге Алг. 4). В рассматриваемом алгоритме вместо стека можно было использовать и другие способы хранения заданий, например очередь. Стек выбран исключительно из соображений простоты реализации. Следует отметить, что в процедурах Алг. 4 для сокращения текста программ опущен код, обеспечивающий контроль переполнения и исчерпания стека.

Метод локального стека

```
IntTrap04(A,B)
{
J=0

fA=f(A)
fB=f(B)

sAB=(fA+fB)*(B-A)/2

while(1)
{
C=(A+B)/2

fC=fun(C)

sAC=(fA+fC)*(C-A)/2
sCB=(fC+fB)*(B-C)/2
sACB=sAC+sCB

if(|sAB-sACB|≥ε|sACB|)
       {
       PUT_INTO_STACK( A, C, fA, fC, sAC)
       A=C
       fA=fC
       sAB=sCB
```

```
      }
else
      {
      J+=sACB
      if(STACK_IS_NOT_FREE)
            break

      GET_FROM_STACK( A, B, fA, fB, sAB)
      }
}

return J
}
```

Алг. 3. Метод локального стека

```
// данные, описывающие стек

sp=0  //    указатель вершины стека

struct
      {
      A,B,fA,fB,s
      }
      stk[1000]   // массив структур в которых
                  // хранятся отложенные задания

// макроопределения доступа к данным стека

#define STACK_IS_NOT_FREE (sp>0)

#define PUT_INTO_STACK(A,B,fA,fB,s)
      {
      stk[sp].A=A
      stk[sp].B=B
      stk[sp].fA=fA
      stk[sp].fB=fB
      stk[sp].s=s
      sp++
      }

#define GET_FROM_STACK(A,B,fA,fB,s)
      {
      sp--
      A=stk[sp].A
      B=stk[sp].B
      fA=stk[sp].fA
      fB=stk[sp].fB
      s=stk[sp].s
      }
```

Алг. 4. Процедуры доступа к локальному стеку

Времена выполнения алгоритмов 2 и 3 отличаются примерно на 5% в пользу последнего, таким образом, алгоритм "локального стека" будет в дальнейшем использоваться в качестве наилучшего из имеющихся в нашем распоряжении последовательных методов. Именно относительно алгоритма 3 будет оцениваться эффективность создаваемых параллельных алгоритмов.

Параллельные алгоритмы

Относительно несложно разработать параллельные алгоритмы решения обсуждаемой задачи на основе метода геометрического параллелизма и метода коллективного решения.

Метод геометрического параллелизма

Согласно методу геометрического параллелизма отрезок интегрирования разбивается на ***p*** частей, где ***p*** - число используемых процессоров. Далее, каждому из процессоров предоставляется обработка одного из интервалов, причем разные процессоры вычисляют значения интеграла на разных интервалах. Рассмотрим соответствующий параллельный алгоритм 5.

```
main()
{
…
 for(i=0;i<p;i++)
            StartParallelProcess
                  ( IntTrap04, A+(B-A)*i/p, A+(B-A)*(i+1)/p, &(s[i]) )

 WaitAllParallelProcess

 J=0
 for(i=0;i<p;i++)
      J+=s[i]
}
```

Алг. 5. Параллельный алгоритм: метод геометрического параллелизма

На данном этапе рассмотрения алгоритмов основное внимание уделяется изучению основных свойств алгоритмов, обнаружению присущего им внутреннего параллелизма. Подробности и методы порождения параллельных процессов остаются на втором плане. Ключевое слово ***StartParallelProcess*** в алгоритме 5, подразумевает, что процедура-функция *IntTrap04* будет запущена ***p*** раз, при этом каждой копии будут переданы параметры, определяемыми соответствующим значением *i*. Ключевое слово ***WaitAllParallelProcess*** предполагает, что стоящие после него инструкции программы не будут выполняться, до тех пор, пока все запущенные параллельные процессы не выполнятся полностью и частичные суммы не будут записаны в соответствующие ячейки массива *s*. Дальнейшее вычисление интеграла сводится к нахождению суммы элементов массива *s* и выполняется последовательно одним процессом.

Алгоритм 5 эффективен при условии равномерного распределения всего объема вычислений по отрезку интегрирования. Однако существует множество функций при интегрировании которых указанное условие не соблюдается. В их числе рассматриваемая нами на отрезке $[10^{-5},1]$ функция 5. При разбиении интервала $[10^{-5},1]$ на ***p*** равных частей практически весь объем вычислений, необходимых для опре

деления интеграла с точностью $\varepsilon = 10^{-5}$, сосредоточен в первой части: $\left[10^{-5},\ 10^{-5} + \frac{1-10^{-5}}{p}\right]$.

Таблица 2

Параметры расчета интеграла на разных отрезках

p	интервал 1	интервал2	время1, с	время2, с
10	[1e-5, 0.10000900000]	[0.10000900000, 1]	37.679	0.004
100	[1e-5, 0.01000990000]	[0.01000990000, 1]	37.274	0.037
1 000	[1e-5, 0.00100999000]	[0.00100999000, 1]	36.989	0.369
10 000	[1e-5, 0.00010999900]	[0.00010999900, 1]	34.064	3.364
100 000	[1e-5, 0.00001999990]	[0.00001999990, 1]	18.869	18.822

Из таблицы **2** следует полная непригодность метода геометрического параллелизма для решения поставленной задачи даже на системе из двух процессоров. При разбиении на две ***равные*** части, на первую из них приходится практически вся вычислительная нагрузка. Для обеспечения равной вычислительной нагрузки на два процессора следует разбить интервал интегрирования на две ***неравные*** части, причем одна из них должна быть в $\mathbf{10^5}$ раз меньше другой.

Метод коллективного решения

Из таблицы **2** так же следует неприменимость и метода коллективного решения (Алг. 6).

```
main()
{
// управляющий процесс

// Предполагается, что число интервалов n не меньше числа процессоров
p:
// n≥p

// Порождение p параллельных процессов, каждый из которых
// выполняет процедуру slave

for(k=0;k<p;k++)
      StartParallel(slave #k)

// Передача порожденным параллельным процессам координат
// концов отрезков интегрирования для определения частичных сумм

i=0 // число интервалов, уже переданных для обработки

for(k=0;k<p;k++)
      {
      Send(slave #k, A+(B-A)*i/n, A+(B-A)*(i+1)/n)
      i++
      }

J=0   // J - значение интеграла на всем интервале [A,B]
      // s - значение частичной суммы

// Пока есть отрезки, не переданные для отработки, следует дождаться
```

```
// сообщения от любого из процессов slave, вычислившего
// частичную сумму на переданном ему отрезке, получить
// значение этой суммы, прибавить к общему значению интеграла
// и передать освободившемуся процессу очередной отрезок

while(i<n)
      {
      Recv(slave #any, s)
      J+=s
      Send(slave #any, A+(B-A)*i/n, A+(B-A)*(i+1)/n)
      i++
      }

// Получить результаты вычислений переданных отрезков
// и прибавить их к общей сумме

for(k=0;k<p;k++)
      {
      Recv(slave #any, s)
      J+=s
      }
}

slave()
{
// подчиненный процесс, вычисляющий значение интеграла на отрезке [a,b]


      while(1)
            {
            Recv(main,a,b)
            s=IntTrap04(a,b)
            Send(main,s)
            }
}
```

Алг. 6. Параллельный алгоритм: метод коллективного решения

Оценим эффективность Алг. 6. Время выполнения последовательного алгоритма:

$$T_1 = \sum_{i=1}^{n} \tau_i \,, \tag{8}$$

где τ_i - время интегрирования отрезка i.

Время выполнения параллельного алгоритма:

$$T_p = \frac{1}{p} \sum_{i=1}^{n} (\tau_i + 2t_s), \tag{9}$$

где t_s - время передачи координат $[a,b]$ или время возврата результата вычислений s. Поскольку объем передаваемых данных и в первом и во втором случае мал, правомерно считать времена их передачи одинаковыми и равными латентности.

Соотношение (9) предполагает, что время интегрирования на каждом из n интервалов одинаково: $\tau_i = \tau_0$. В этом случае эффективность составит:

$$E_p = \frac{n\tau_0}{p\frac{1}{p}n(\tau_0 + 2t_s)} = \frac{\tau_0}{\tau_0 + 2t_s} = \frac{1}{1+\frac{2t_s}{\tau_0}} \qquad (10)$$

Согласно (10) эффективность близка к 100% при соблюдении условия $\tau_0 >> 2t_s$ и при равенстве времен обработки разных фрагментов интервала интегрирования. Данные таблицы **2** свидетельствует об обратном: при равной длине фрагментов отрезка интегрирования, требуется разное время для обработки каждого из них. В связи с этим соотношение (9) преобразуется следующем образом:

$$T_p^{'} = \max_p \sum_{i \in I_p} (\tau_i + 2t_s),$$

где I_p - множество номеров отрезков, обработанных процессом p.

Для того, что бы все процессы были загружены равномерно, следует разбить исходный интервал на значительное число равных отрезков. Причем, их количество на несколько порядков превысит общее число интервалов, необходимых при решении задачи с помощью последовательного алгоритма. Таким образом:

- параллельный алгоритм выполнит в целом больше операций, чем последовательный;

- соотношение $\tau_i >> 2t_s$ не выполняется для большинства из этих отрезков, поскольку среди отрезков есть такие, на которых интеграл можно с достаточной точностью найти без измельчения сетки, что требует времени значительно меньше, чем необходимо для синхронизации (передачи данных).

В результате эффективность метода коллективного решения крайне низка, фактически его использование ведет к увеличению времени решения задачи. Таким образом, методы геометрического параллелизма и коллективного решения практически непригодны для решения поставленной задачи.

Метод глобального стека

В дальнейшем будем рассматривать системы с общей памятью. В качестве параллельных процессов будем использовать треды - легковесные процессы, нити. Для защиты разделяемых переменных будем использовать семафоры.

Метод глобального стека не требует наличия управляющего процесса. Процесс, выполняющий запуск расчета, никаким образом не участвует ни в самой процедуре расчета, ни в принятии решения об окончании вычислений. В этом принципиальное отличие метода глобального стека от метода коллективного решения: порожденные процессы совершенно равноправны в своих возможностях и каждый из них в равной степени выполняет вычислительные и управляющие функции. Укрупненная блок-схема алгоритма приведена на Рис. 2.

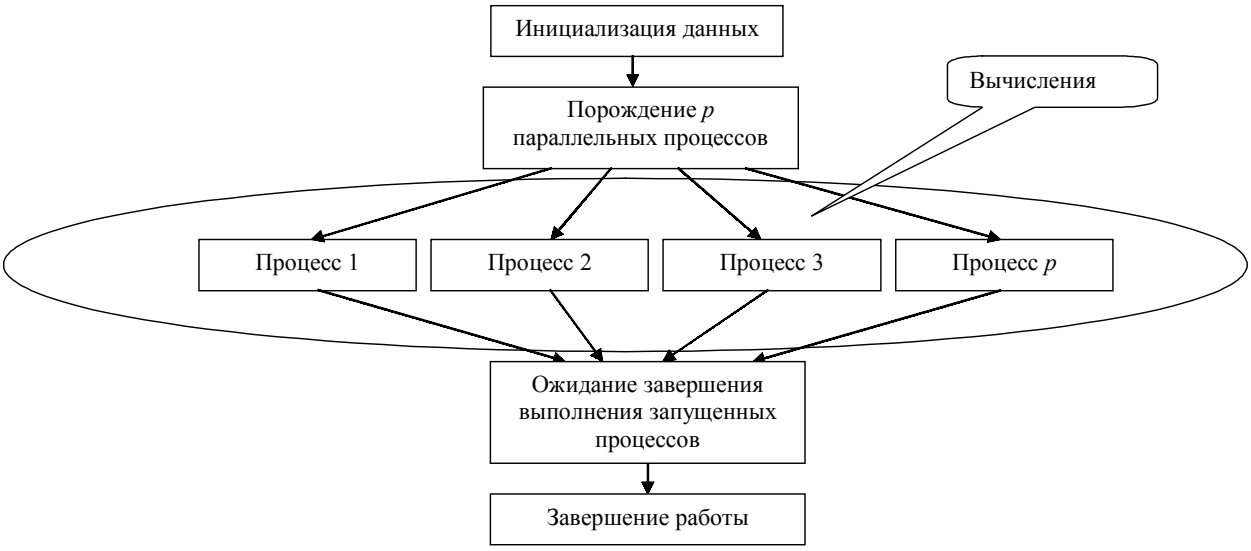

Рис. 2. График *f(x)*

Основные вычисления выполняются параллельными процессами Процесс1 … Процесс*p*, каждый из которых выполняет один и тот же алгоритм.

Пусть в нашем распоряжении есть доступный всем параллельным процессам список отрезков интегрирования, организованный в виде стека. Назовем его глобальным стеком. Предположим, что перед запуском параллельных процессов в глобальный стек помещается единственная запись - координаты концов отрезка интегрирования - в дальнейшем "отрезок".

Тогда, предлагается следующая схема алгоритма, выполняемого каждым из параллельных процессов:

Пока в глобальном стеке есть отрезки:

- взять один отрезок из глобального стека
- выполнить алгоритм локального стека (Алг. 3), выполняя при записи в локальный стек следующие действия:

- если в локальном стеке есть несколько отрезков, а в глобальном стеке отрезки отсутствуют, то
  - переместить часть отрезков из локального стека в глобальный стек.
- по исчерпании локального стека добавить полученную частичную сумму к общему значению интеграла.

Данный несложный алгоритм вызывает ряд вопросов, например:

- должен ли процесс закончить работу, если в глобальном стеке отрезков нет?
- какую часть отрезков следует перемещать из локального стека в глобальный стек?
- как обеспечить корректную запись отрезков разными процессами в глобальный стек?
- в какой момент следует завершить работу параллельных процессов?

Легко видеть, что отсутствие отрезков в глобальном стеке не является достаточным основанием для завершения процесса, обнаружившего этот факт. В самом деле, перед запуском интегрирующих процессов в глобальный стек помещается только один отрезок. В начале работы он будет взят из стека только одним их процессов, например Процессом1, после чего остальные процессы могут обнаружить, что стек пуст. Могут, но не обязательно обнаружат, поскольку нет никаких весомых оснований утверждать, что какой-либо процесс, например Процесс2, обратится к глобальному стеку раньше, чем Процесс1 разместит в нем некоторое количество новых отрезков. Обе ситуации возможны и в алгоритме должна быть предусмотрена их корректная обработка.

Запускающий процесс выглядит достаточно простым и коротким (Алг. 7). После инициализации переменных выполняется запуск *np* интегрирующих процессов. Затем запускающий процесс переходит в состояние ожидания окончания работы запущенных процессов и в вычислениях участия не принимает.

```
// sdat. - глобальные переменные, разделяемые
//          всеми процессами slave_thr
//          и запускающим процессом main

main()
{
 // запускающая программа

 sdat.ntask=0 // число отрезков в глобальном стеке интервалов
 sdat.nactive=0 // число процессов, обрабатывающих в
                // данный момент один из интервалов
```

```
 Sem_init(sdat.sem_sum, 1)  //доступ к глобальной сумме открыт
 Sem_init(sdat.sem_list, 1) //доступ к глобальному стеку открыт
 Sem_init(sdat.sem_task_present, 0) // отрезков в
                                    // глобальном стеке нет

 sdat.s_all=0 // Глобальная сума - переменная,
                   // накапливающая частичные суммы

 PUT_INTO_GLOBAL_STACK[sdat.ntask]
            (A,B,fun(A),fun(B),(fun(A)+fun(B))*(B-A)/2.)

 sdat.ntask++ // в глобальный стек занесен первый из отрезков

 // откроем семафор, свидетельствующий о том,
 // что в глобальном стеке интервалов есть отрезки
 sem_post(&sdat.sem_task_present)

 // запуск np параллельных процессов, выполняющих функции slave_thr

 StartThrTask(p, slave_thr);
 WaitThrTask();


 // Параллельное выполнение функций slave_thr завершено

 J=sdat.s_all; // значение интеграла
}
```

Алг. 7. Запускающий процесс

Уточним структуру интегрирующего процесса (Алг. 8).

```
slave_thr()
{

 // начало цикла обработки стека интервалов
 while(sdat.ntask>0)
      {
      // чтение одного интервала из списка интервалов

      sdat.ntask-- // указатель глобального стека
      GET_FROM_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)

      // начало цикла интегрирования одного интервала
      while(1)
            {
            c=(a+b)/2
            fc=fun(c)

            sac=(fa+fc)*(c-a)/2
            scb=(fc+fb)*(b-c)/2
            sacb=sac+scb

            if(!BreakCond(sacb,sab))
                  {
                  s+=sacb
                  if(!sp) // локальный стек пуст
                        break // выход из цикла интегрирования
                              // одного интервала
                  sp--
```

```
                GET_FROM_LOCAL_STACK[sp]( a, b, fa, fb, sab)
                }
          else
                {
                PUT_INTO_LOCAL_STACK[sp]( a, c, fa, fc, sac)
                sp++
                a=c
                fa=fc
                sab=scb
                }

          // перемещение части локального стека
          // в общий список интервалов

          if((sp>SPK) && (!sdat.ntask))
                {
                while((sp>1) && (sdat.ntask<sdat.maxtask))
                      {
                      sp--
                      GET_FROM_LOCAL_STACK[sp](a,b,fa,fb,sab)
                      PUT_INTO_GLOBAL_STACK[sdat.ntask]
                                              (a,b,fa,fb,sab)
                      sdat.ntask++
                      }
                }
          }
     // конец цикла интегрирования одного интервала
     }
 // конец цикла обработки стека интервалов

 sdat.s_all = sdat.s_all + s
}
```

Алг. 8. Параллельный алгоритм: метод глобального стека (версия 1)

Алгоритм мало отличается от метода локального стека (Алг. 3). Отличия сосредоточены в его последней части:

- в конце цикла интегрирования одного интервала присутствует фрагмент переноса отрезков из локального стека в глобальный стек;

- по окончании цикла интегрирования выполняется суммирование частичных сумм, полученных всеми процессами.

Данный алгоритм неработоспособен по следующим причинам:

1. несмотря на проверку в начале цикла обработки стека интервалов (*sdat.ntask>0*) может возникнуть ситуация, в которой одна и та же запись будет извлечена более чем одним процессом;
2. возможно, не все частичные суммы будут добавлены к общему значению интеграла.

Рассмотрим последовательность действий, иллюстрирующую возникновение первой ошибки:

№ шага	состояние переменных	Процесс1	Процесс2
0	*sdat.ntask=1*	*while(sdat.ntask>0)*	*while(sdat.ntask>0)*
2		*sdat.ntask--*	
3	*sdat.ntask=0*		*sdat.ntask--*
4	*sdat.ntask=-1*	*GET_OF_GLOBAL_STACK[sdat.ntask] - чтение записи с номером -1*	*GET_OF_GLOBAL_STACK[sdat.ntask] - чтение записи с номером -1*

Два процесса успешно выполнили проверку наличия в глобальном стеке записей, затем первый процесс уменьшил указатель стека, ранее указывающий на первую свободную ячейку. После шага 2 указатель стека направлен на ячейку, хранящую отрезок, что пока еще правильно. На шаге 3 второй процесс выполнил такое же действие. В результате указатель стека направлен на ячейку -1. В результате возникают сразу две проблемы: во-первых, на шаге 4 оба процесса прочтут одну и ту же запись; во-вторых, такой записи нет – она за пределами стека. Попытка чтения несуществующей записи приведет, в лучшем случае, к аварийной остановке программы, в худшем - к непредсказуемым неверным результатам.

Рассмотрим возникновение второй ошибки:

№ шага	состояние глобальных переменных	Процесс1 выполняемые действия состояние локальных переменных	Процесс2 выполняемые действия состояние локальных переменных
0	*sdat.s_all=0*	*s=1* *sdat.s_all= 0 + 1*	*s=2* *sdat.s_all= 0 + 2*
2	*sdat.s_all=* *1 или 2*		

В результате выполнения указанных действий переменная *sdat.s_all* может получить как верное, так и не верное значение.

Обе ошибки вызваны отсутствием мер, предотвращающих одновременную модификацию разделяемых переменных несколькими процессами. Избежать второй ошибки достаточно просто: достаточно предусмотреть использование семафора, запрещающего одновременный доступ к разделяемой переменной *sdat.s_all* более чем одному процессу:

```
main()
.
Sem_init(sdat.sem_sum,1) //доступ к глобальной сумме открыт
.

slave_thr()
.
 // Начало критической секции сложения частичных сумм
 //
 sem_wait(sdat.sem_sum)
 sdat.s_all+=s
```

```
sem_post(sdat.sem_sum)
//
// Конец критической секции сложения частичных сумм
```

Алг. 9. Критическая секция накопления глобальной суммы

Будем предполагать, что используемые нами семафоры являются бинарными и, в отличии от обычных переменных, могут принимать только два значения: 0 и 1, причем 1 соответствует открытому семафору, а 0 - закрытому. Инициализация семафора выполняется с помощью операции *sem_init(семафор, 0 или 1)*. При выполнении над закрытым семафором операции *sem_wait(семафор)* процесс (или процессы), выполнивший ее, блокируется вплоть до открытия семафора. При выполнении операции *sem_wait(семафор)* над открытым семафором процесс, выполнивший ее продолжает свое выполнение, а состояние соответствующего семафора меняется на закрытое. При выполнении над закрытым семафором операции *sem_post(семафор)* соответствующий семафор переходит в открытое состояние, если нет заблокированных им процессов, в противном случае некоторый из заблокированных процессов продолжает свое выполнение, а семафор остается в закрытом состоянии. Мы будем считать, что выполнение операции *sem_post(семафор)* над открытым семафором запрещено. Другие операции над семафорами нами рассматриваться не будут.

Алг. 9 обеспечивает корректное выполнение любого количества процессов при сложении частичных сумм, но посмотрим, к чему приведет аналогичное решение для разграничения доступа к глобальному стеку отрезков интегрирования (Алг. 10).

```
main()
.
Sem_init(sdat.sem_list, 1) //доступ к глобальному стеку открыт
.

slave_thr()
{
.while(1)
      {
      // Начало критической секции чтения из глобального
      // стека очередного интервала интегрирования
      //
      sem_wait(sdat.sem_list)

      if(sdat.ntask≤0)
            {
            sem_post(sdat.sem_list) // разрешить другим процессам
                                    // доступ к глобальному стеку
            break
            }

      sdat.ntask-- // указатель глобального стека
      GET_FROM_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)

      sem_post(sdat.sem_list)
```

```
        //
        // Конец критической секции чтения из глобального
        // стека очередного интервала интегрирования
        .
        }
.
}
```

Алг. 10. Критическая секция накопления глобальной суммы

При выполнении алгоритма Алг. 10 возможна следующая последовательность действий:

№ шага	состояние переменных	Процесс1	Процесс2
0	*sdat.ntask=1* *sdat.sem_list=1*	*sem_wait(sdat.sem_list)*	
2	*sdat.ntask=1* ***sdat.sem_list=0***	*sdat.ntask--*	
3	***sdat.ntask=0*** *sdat.sem_list=0*	*GET_OF_GLOBAL_STACK[0]*	
4	*sdat.ntask=0* ***sdat.sem_list=1***	*sem_wait(sdat.sem_list)*	
5	*sdat.ntask=0* *sdat.sem_list=1*		*sem_wait(sdat.sem_list)*
6	*sdat.ntask=0* ***sdat.sem_list=0***		*if(sdat.ntask≤0)* *{* *sem_post(sdat.sem_list)* *break - выйти из цикла* *}*
7	*sdat.ntask=0* ***sdat.sem_list=1***		*процесс завершил работу*

В результате Процесс2 заканчивает работу раньше, чем Процесс1 полностью вычислит интеграл на заданном отрезке интегрирования. Таким образом, может оказаться, что Процесс1, после завершения Процесса2, породит некоторое количество отрезков и запишет их в глобальный стек. Процесс2 мог бы принять участие в их обработке, но не сделает этого, поскольку уже завершится к этому моменту. Обратим внимание на то, что в результате выполнения программы будет получен правильный результат - Процесс1 обработает все отрезки, но время выполнения программы будет больше, чем могло бы быть при работе двух процессов. Можно сделать вывод о том, что условие выхода из *цикла обработки стека интервалов* выбрано неудачно. Интегрирующие процессы не должны заканчивать работу до тех пор, пока все отрезки интервала интегрирования не будут полностью обработаны.

Отрезок интегрирования может находиться в нескольких состояниях:

- находится в глобальном стеке интервалов;
- обрабатывается некоторым интегрирующим процессом;

- находится в локальном стеке интервалов некоторого процесса;

- полностью обработан: известно значение интеграла на этом отрезке и оно прибавлено к локальной частичной сумме соответствующего процесса.

"Время жизни" отрезка, после того, как некоторый процесс начал его обработку, относительно невелико - отрезок разбивается на две части и перестает существовать, породив два новых отрезка. Таким образом, требование "*все отрезки интервала интегрирования полностью обработаны*" означает, что:

- функция проинтегрирована на всех отрезках, покрывающих исходный интервал интегрирования;

- полученные на отрезках интегрирования значения интегралов добавлены к частичным суммам соответствующих процессов.

Таким образом, потребуются следующие основные общие данные:

	семафоры доступа:
sdat.sem_list	семафор доступа к глобальному стеку отрезков
sdat.s_all	семафор доступа к значению интеграла

	семафоры состояния:
sdat.sem_task_present	семафор наличия записей в глобальном стеке отрезков

	переменные:
sdat.list_of_tasks	глобальный стек отрезков
sdat.ntask	число записей в глобальном стеке отрезков - указатель глобального стека отрезков
sdat.nactive	число активных процессов
sdat.s_all	значение интеграла

Окончательная версия алгоритма, свободная от рассмотренных выше недостатков приведена в листинге Алг. 11. Процедуры доступа к глобальному стеку приведены в листинге Алг. 12, процедуры доступа к локальному стеку полностью аналогичны процедурам с листинга Алг. 4.

```
int slave_thr()
{
 // все переменные, начинающиеся с sdat. являются глобальными,
 // к ним имеет доступ каждый из запущенных процессов slave_thr
 // и запускающая программа main

 // sp, s - локальные переменные процессов slave_thr

 sp=0 // указатель локального стека - локальный стек пуст

 s=0  // частичная сумма интегралов, вычисленных на отрезках,
      // обработанных данной копией процесса

 // начало цикла обработки стека интервалов
 while(1)
      {
      // ожидание появления в глобальном стеке интервалов для обработки

      sem_wait(sdat.sem_task_present)

      // чтение одного интервала из списка интервалов

      // Начало критической секции чтения из глобального
      // стека очередного интервала интегрирования
      //
      sem_wait(sdat.sem_list)

      sdat.ntask-- // указатель глобального стека
      GET_OF_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)
```

```
if(sdat.ntask)
      sem_post(&sdat.sem_task_present)

if(a<=b) // очередной отрезок не является терминальным
      sdat.nactive++  // увеличить число процессов, имеющих
                      // интервал для интегрирования

sem_post(sdat.sem_list)
//
// Конец критической секции чтения из глобального
// стека очередного интервала интегрирования

if(a>b) // отрезок является терминальным
      break // выйти из цикла обработки стека интервалов

// начало цикла интегрирования одного интервала
while(1)
      {
      c=(a+b)/2
      fc=fun(c)

      sac=(fa+fc)*(c-a)/2
      scb=(fc+fb)*(b-c)/2
      sacb=sac+scb

      if(!BreakCond(sacb,sab))
            {
            s+=sacb
            if(!sp) // локальный стек пуст
                  break // выход из цикла интегрирования
                        // одного интервала
            sp--
            GET_FROM_LOCAL_STACK[sp]( a, b, fa, fb, sab)
            }
      else
            {
            PUT_TO_LOCAL_STACK[sp]( a, c, fa, fc, sac)
            sp++
            a=c
            fa=fc
            sab=scb
            }

      // перемещение части локального стека
      // в общий список интервалов

      if((sp>SPK) && (!sdat.ntask))
            {
            // Начало критической секции заполнения глобального
            // стека отрезками интегрирования
            //
            sem_wait(sdat.sem_list)

            if(!sdat.ntask)
                  {
                  // установить семафор наличия
                  // записей в глобальном стеке

                  sem_post(sdat.sem_task_present)
                  }

            while((sp>1) && (sdat.ntask<sdat.maxtask))
                  {
```

```
                        sp--
                        GET_FROM_LOCAL_STACK[sp](a,b,fa,fb,sab)
                        PUT_TO_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)
                        sdat.ntask++
                        }

                  sem_post(sdat.sem_list)
                  //
                  // Конец критической секции заполнения глобального
                  // стека отрезками интегрирования
                  }
            }
            // конец цикла интегрирования одного интервала

      // Начало критической секции заполнения глобального
      // стека терминальными отрезками (a>b)
      //
      sem_wait(&sdat.sem_list)
      sdat.nactive--

      if( (!sdat.nactive) && (!sdat.ntask) )
            {
            // запись в глобальный стек списка терминальных отрезков
            for(i=0;i<nproc;i++)
                  {
                  PUT_TO_GLOBAL_STACK[sdat.ntask](2,1,-,-,-)
                  sdat.ntask++;
                  }

            // в глобальном стеке есть записи
            sem_post(sdat.sem_task_present)
            }

      sem_post(sdat.sem_list)
      //
      // Конец критической секции заполнения глобального
      // стека терминальными отрезками
      }
      // конец цикла обработки стека интервалов

 // Начало критической секции сложения частичных сумм
 //
 sem_wait(&(sdat.sem_sum))
 sdat.s_all+=s
 sem_post(&(sdat.sem_sum))
 //
 // Конец критической секции сложения частичных сумм
}
```

Алг. 11. Параллельный алгоритм: метод глобального стека

```
PUT_INTO_GLOBAL_STACK[sdat.ntask]
                  (A,B,fA,fB,sAB)
      {
      sdat.list_of_tasks[sdat.ntask].a=A;
      sdat.list_of_tasks[sdat.ntask].b=B;
      sdat.list_of_tasks[sdat.ntask].fa=fA;
      sdat.list_of_tasks[sdat.ntask].fb=fB;
      sdat.list_of_tasks[sdat.ntask].s=sAB;
      }
```

```
PUT_FROM_GLOBAL_STACK[sdat.ntask]
                    (A,B,fA,fB,sAB)
      {
      A=sdat.list_of_tasks[sdat.ntask].a;
      B=sdat.list_of_tasks[sdat.ntask].b;
      fA=sdat.list_of_tasks[sdat.ntask].fa;
      fB=sdat.list_of_tasks[sdat.ntask].fb;
      sAB=sdat.list_of_tasks[sdat.ntask].s;
      }
```

Алг. 12. Процедуры доступа к глобальному стеку

Инициализация глобальных переменных выполняется в запускающей программе Алг. 7 перед запуском интегрирующих процессов ***slave_thr***. В начале работы процессы ***slave_thr*** инициализируют свои локальные стеки, устанавливая в них нулевое количество записей, так же они устанавливают нулевые значения локальных сумм.

В дальнейшем основная работа выполняется внутри цикла обработки стека интервалов. В листинге Алг. 11 жирным шрифтом выделены строки описывающие критические интервалы и доступ к ним. Каждый из процессов ожидает появления записей в глобальном стеке:

```
sem_wait(sdat.sem_task_present)
```

Использование данного семафора предотвращает доступ к заведомо пустому стеку, однако не отменяет необходимости использования семафора доступа ***sdat.sem_list***, гарантирующего корректную модификацию глобальных переменных несколькими процессами. Таким образом, чтение записи выполняется в критической секции, ограниченной строками:

```
sem_wait(sdat.sem_list)
sem_post(sdat.sem_list)
```

Внутри критической секции выполняется чтение и изъятие из глобального стека одного из отрезков. Далее открывается семафор наличия в глобальном стеке отрезков, если они есть:

```
if(sdat.ntask)
    sem_post(&sdat.sem_task_present)
```

Затем выполняется анализ типа отрезка. Если правый конец меньше или равен левому, то отрезок интерпретируется как сигнал окончания работы («терминальный» отрезок). В противном случае увеличивается счетчик активных процессов – процессов, получивших интеграл интегрирования:

```
if(a<=b)
       sdat.nactive++
```

После выхода из критической секции осуществляется либо обработка очередного интервала в цикле интегрирования одного интервала, либо выход из цикла обработки стека интервалов, если получен терминальный отрезок.

Внутри цикла интегрирования одного интервала осуществляется контроль числа отрезков, размещенных в локальном стеке процесса. Если оно превышает наперед заданную величину ***SPK*** и в глобальном стеке нет отрезков, то осуществляется перенос части отрезков из локального стека в глобальный. Перенос осуществляется в пределах критической секции. Следует обратить внимание на то, что проверка

```
if((sp>SPK) && (!sdat.ntask))
```

выполняется вне критической секции, следовательно, возможна ситуация, в которой в момент доступа процесса внутрь критического интервала переменная ***sdat.ntask*** уже не будет иметь нулевое значение – то есть в глобальном стеке другим процессом уже будут размещены некоторые отрезки. Именно поэтому семафор наличия в стеке отрезков устанавливается только в том случае, если стек реально пуст (мы договорились, что выполнять операцию ***sem_post*** над открытым семафором нельзя):

```
if(!sdat.ntask)
      sem_post(sdat.sem_task_present)
```

По окончании цикла обработки очередного отрезка выполняется проверка наличия заданий в глобальном стеке отрезков и наличия активных процессов:

```
if( (!sdat.nactive) && (!sdat.ntask) )
```

Эти действия выполняются внутри критической секции, таким образом, если обе проверки дали отрицательный результат, можно быть уверенным в том, что все фрагменты заданного интервала интегрирования уже обработаны, можно переходить к суммированию частичных сумм. Однако только один процесс «владеет» информацией о том, что интегрирование фрагментов полностью завершено, следует сообщить б этом остальным процессам, напомним, что они находятся в состоянии ожидания открытия «семафора наличия отрезков в глобальном стеке». Таким образом, единственным легальным способом завершить их работу, является размещение в глобальном стеке терминальных отрезков и открытие соответствующего семафора:

```
for(i=0;i<nproc;i++)
      {
      PUT_TO_GLOBAL_STACK[sdat.ntask](2,1,-,-,-)
      sdat.ntask++;
      }
sem_post(sdat.sem_task_present)
```

Процесс, получивший терминальный отрезок, добавляет внутри критической секции найденную им частичную сумму к общему значению интеграла и заканчивает свою работу.

```
sem_wait(&(sdat.sem_sum))
sdat.s_all+=s
sem_post(&(sdat.sem_sum))
```

Рассмотренный алгоритм демонстрирует достаточно высокую эффективность при выполнении на 2, 3 и 4х процессорах, что подтверждают приведенные ниже результаты определения интеграла $J = \int_{10^{-5}}^{1} \frac{1}{x^2} \sin^2\left(\frac{1}{x}\right)$ с точностью $\varepsilon = 10^{-5}$.

Время выполнения

Np	*1*	*2*	*3*	*4*
tiger.jscc.ru	*31.39*	*15.61*	*10.29*	*7.83*
ga03.imamod.ru	*37.48*	*19.00*	*-*	*-*

Ускорение

Np	*1*	*2*	*3*	*4*
tiger.jscc.ru	*1*	*2.01*	*3.05*	*4.01*
ga03.imamod.ru	*1*	*1.97*		

Эффективность

np	*1*	*2*	*3*	*4*
tiger.jscc.ru	*100%*	*101%*	*102%*	*100%*
ga03.imamod.ru	*100%*	*99%*		

В приложениях приведен перечень использованных в примерах функций работы с нитями, семафорами и подпрограмма измерения времени.

В листинге Алг. 7 запуск параллельных процессов упрощенно описан строками:

```
StartThrTask(np, slave_thr);
WaitThrTask();
```

В листинге Алг. 13 приведен подробный пример, иллюстрирующий последовательность запуска параллельных процессов, каж

дый из которых выполняет процедуру, указанную параметром ***tsub***. В рассматриваемом примере запускается процедура ***slave_thr*.**

```
typedef int ThrSub(void);

pthread_t *tid_procs;

StartThrTask(int np, ThrSub *tsub)
{
 int i;

// --- распределить массив дескрипторов запускаемых процессов ---

 tid_procs = (pthread_t *)malloc(np*sizeof(pthread_t));

// --- запустить np процессов ---

 for (i = 0; i < np; i++)
      pthread_create(&(tid_procs[i]), NULL, tsub, NULL);

#ifdef      HAVE_THR_SETCONCURRENCY_PROTO
      int   thr_setconcurrency(int);

      thr_setconcurrency(np);
#endif
}

WaitThrTask(void)
{
 int i;

 for(i = 0; i < np; i++)
      pthread_join(tid_procs[i], NULL);
}
```

Алг. 13. Процедуры запуска и завершения параллельных нитей

Приложения

Перечень использованных в примерах функций Posix

Семафоры

Для использования функций управления семафорами следует включить в текст программы заголовочный файл ***semaphore.h***:

```
#include <semaphore.h>
```

int sem_init(sem_t *sem, int pshared, unsigned int value);

Обращение к функции ***sem_init*** приводит к инициализации семафора и установке его в закрытое состояние, если ***value*** =0, и в открытое, если ***value*** =1. Семафоры ***sem_t*** являются семафорами общего вида, поэтому при инициализации могут быть указаны и другие значения ***value***, но в рамках рассмотренных примеров используются только 0 или 1. В простейшем случае ***pshared*** = 0.

Пример вызова:

```
sem_t sem;
sem_init(&sem, 0, 0);
```

int sem_wait(sem_t * sem);

Обращение к функции ***sem_wait*** приводит к остановке процесса, вплоть до изменения состояния семафора ***sem*** на не нулевое, если его значение в момент обращения равно 0. В противном случае значение семафора ***sem*** уменьшается на 1 и выполнение процесса продолжается.

Пример вызова:

```
sem_wait(&sem);
```

int sem_post(sem_t * sem);

Обращение к функции ***sem_post*** приводит к увеличению значения семафора ***sem*** на 1.

Пример вызова:

```
sem_post(&sem);
```

Процессы

Для использования функций порождения нитей следует включить в текст программы заголовочный файл ***pthread.h***:

```
#include <pthread.h>
```

int pthread_create(pthread_t * tid, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);

Обращение к функции ***pthread_create*** приводит к запуску процесса выполняющего функцию ***start_routine***. В качестве первого параметра необходимо задать адрес дескриптора порождаемого процесса ***tid***. В простейшем случае остальные параметры могут быть установлен в NULL.

Пример вызова:

```
pthread_t  tid;
pthread_create(&tid, NULL, ThreadProcess, NULL);
```

int pthread_join(pthread_t tid, void **thread_return);

Обращение к функции ***pthread_join*** приводит к приостановке вызвавшего ее процесса до тех пор, пока нить, указанная дескриптором ***tid***, не закончит свое выполнение. Через параметр ***thread_return*** передается значение, возвращаемое исполняемой нитью процедурой. В простейшем случае этот параметр может быть установлен в NULL, в этом случае возврата значения не происходит.

Пример вызова:

```
pthread_join(tid, NULL);
```

int thr_setconcurrency(int p);

В случае присутствия этой функции в используемой версии операционной системы ее вызов обеспечивает возможность одновременного выполнения ***p*** нитей.

Пример вызова:

```
#ifdef      HAVE_THR_SETCONCURRENCY_PROTO
      thr_setconcurrency(p);
#endif
```

Компиляция и сборка программы

При создании исполняемого кода программы, возможно, потребуется указать две стандартные библиотеки, содержащие функции работы с нитями и семафорами. Например, компиляцию и сборку программы, исходный текст которой расположен в файле ***prog.c***, можно выполнить следующим образом:

*gcc prog.c **-lpthread –lrt*** *-o prog.x*

Измерение времени

Для измерения интервалов времени удобно использовать функцию ***vmtime***, текст которой приведен в листинге Алг. 14. Она возвращает значение времени истекшего с некоторого, неизменного в ходе выполнения программы, момента в прошлом, выраженное в секундах.

```
double vmtime(void)
{
 double z;
 struct timeval tst;
 struct timezone tz;

 gettimeofday(&tst,&tz);
 z = (double)tst.tv_sec+1e-6*tst.tv_usec;
 return z;
}
```

Алг. 14. Процедуры запуска и завершения параллельных нитей

Библиографический список

1. Языки программирования. Под ред. Ф.Женуи, Пер. с англ. В.П.Кузнецова, Под ред. В.М.Курочкина. - М.: Мир, 1972, 406 стр.
2. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Пер. с англ. - Мир, 1981, 368 с.
3. *Стивенс. У.* Unix: взаимодействие процессов. Мастер-класс./ Пер. с англ.Д.Солнышков. – Спб: Питер, 2002.-576 с. ISBN 5-318-00534-9.
4. *Якобовский М.В.* Распределенные системы и сети. Учебное пособие. – М.: МГТУ “Станкин”, 2000, 118 с, ил
5. Транспьютеры. Архитектура и программное обеспечение. Пер. с англ. / Под ред. Г. Харпа. - М.: Радио и связь, 1993, 304 с., ил.

Содержание