

Лекция 8

Параллельные алгоритмы численного интегрирования

Якобовский Михаил Владимирович
проф., д.ф.-м.н.
Институт прикладной математики
им. М.В.Келдыша РАН, Москва

Постановка задачи

Вычислить с точностью ε значение определенного интеграла

$$J(A, B) = \int_A^B f(x) dx$$

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

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

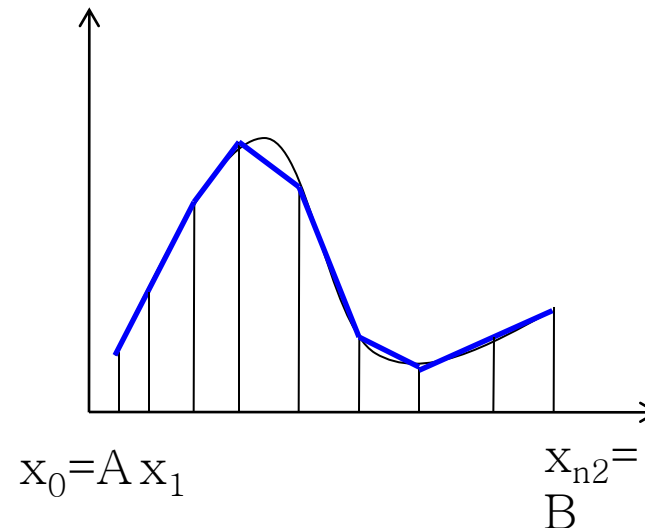
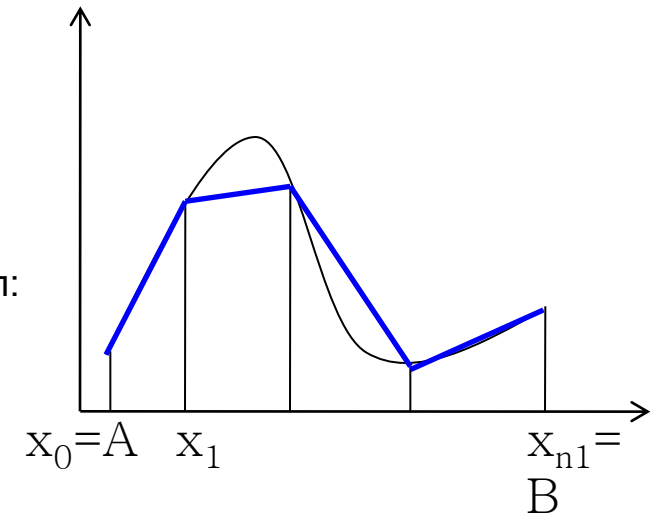
Тогда, согласно методу трапеций, можно численно найти определенный интеграл от функции на отрезке $[A, B]$:

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

Будем полагать значение J найденным с точностью ε , если выполнено условие:

$$|J_{n1} - J_{n2}| \leq \varepsilon |J_{n2}|$$

$$n_2 > n_1$$



Последовательный алгоритм интегрирования

```
IntTrap01 (A, B)
```

```
{
```

```
  n=1
```

```
   $J_{2n} = (f(A) + f(B)) (B-A) / 2$ 
```

```
  do
```

```
  {
```

```
     $J_n = J_{2n}$ 
```

```
    n=2n
```

```
     $s = f(A) + f(B)$ 
```

```
    for (i=1; i<n; i++)
```

```
       $s += 2f(A + (B-A) i/n);$ 
```

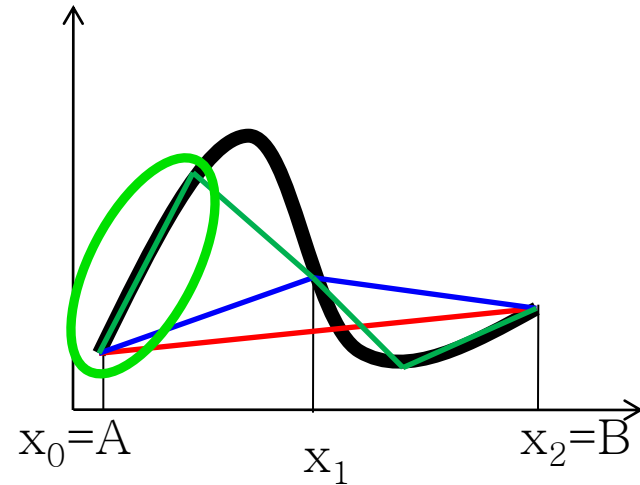
```
     $J_{2n} = s (B-A) / n;$ 
```

```
  }
```

```
  while ( $|J_{2n} - J_n| \geq \varepsilon J_{2n}$ )
```

```
  return  $J_{2n}$ 
```

```
}
```



Недостатки:

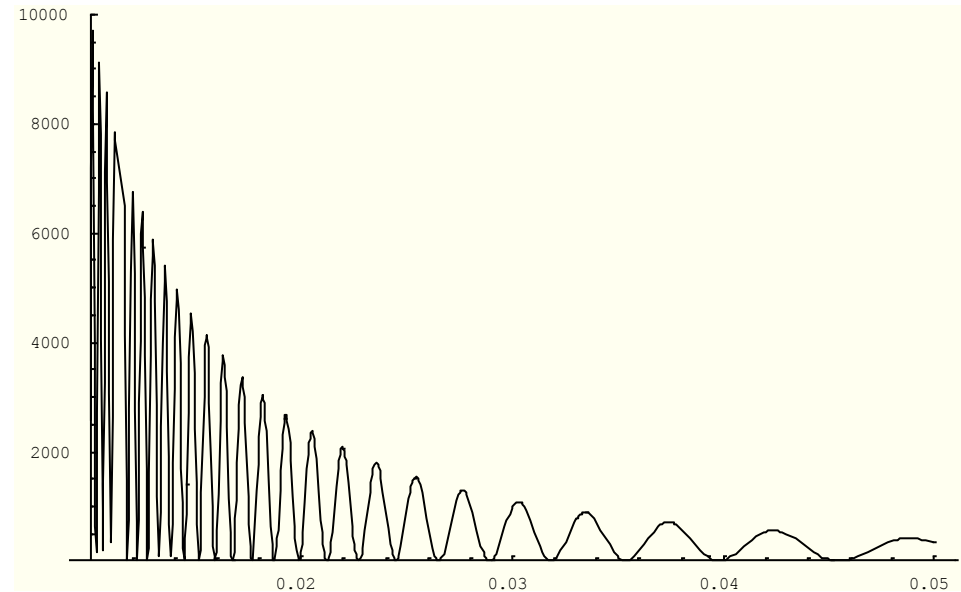
- в некоторых точках значение подынтегральной функции вычисляется более одного раза
- на всем интервале интегрирования используется равномерная сетка, тогда как число узлов сетки на единицу длины на разных участках интервала интегрирования, необходимое для достижения заданной точности, зависит от вида функции

Пример функции

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

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

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

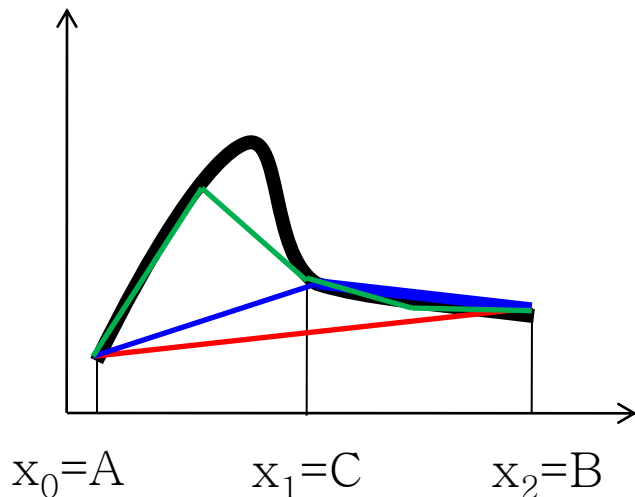


Результаты вычисления интеграла на разных отрезках [A,B]

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

Адаптивный алгоритм

```
main()  
{  
  J= IntTrap03( A, B, f(A), f(B) )  
}
```



Недостаток:

координаты концов отрезков
хранятся в программном стеке
процесса и фактически
недоступны программисту

```
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
```

```
}
```

Преимущества:

- нет повторных вычислений функции
- малое число вычислений на гладких участках

Метод локального стека

```
IntTrap04 (A,B)
```

```
{  
  J=0
```

```
  fA=f (A)
```

```
  fB=f (B)
```

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

```
  while(1)
```

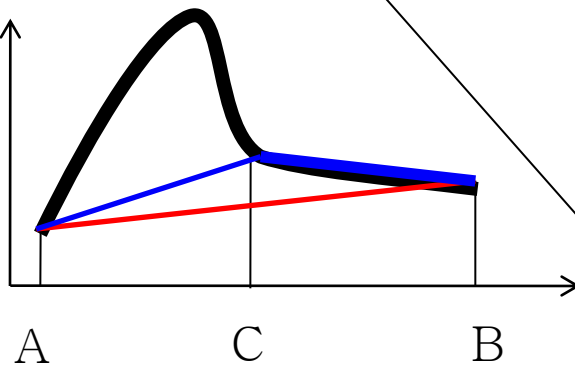
```
  {
```

```
    Обработка отрезка
```

```
  }
```

```
  return J
```

```
}
```



```
  C=(A+B) /2
```

```
  fC=f (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
```

```
  {
```

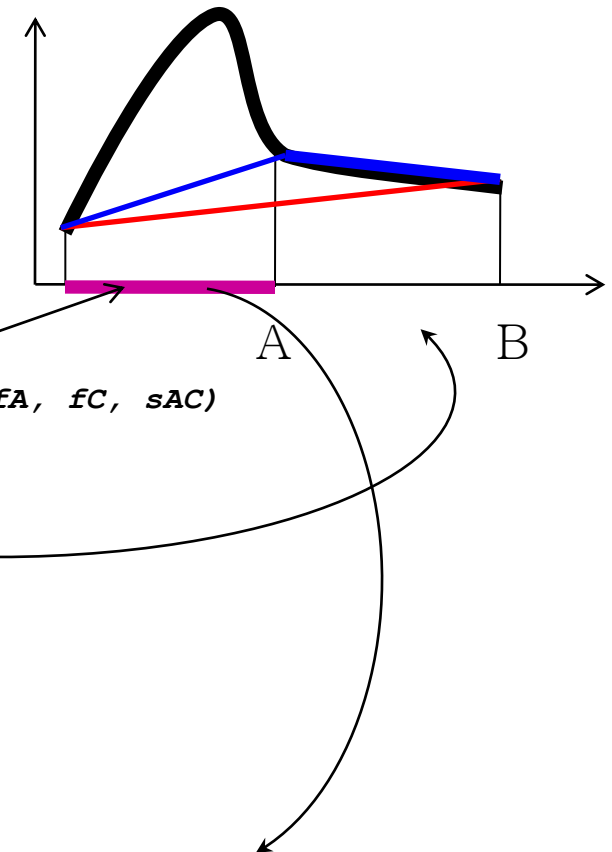
```
    J+=sACB
```

```
    if (STACK_IS_FREE)
```

```
      break
```

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

```
  }
```



Процедуры и данные обеспечивающие стек

```
// данные, описывающие стек
```

```
// указатель вершины стека
```

```
sp=0
```

```
// массив структур в которых
```

```
// хранятся отложенные задания
```

```
struct
```

```
{
```

```
A,B,fA,fB,s
```

```
}
```

```
stk[1000]
```

```
// макроопределения доступа к стеку
```

```
#define STACK_IS_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
```

```
}
```

К вопросу о времени выполнения

- ❑ Тестирование показало, что при расчете с помощью алгоритма локального стека *IntTrap04* время работы было меньше, примерно на 5%, чем при использовании *IntTrap03*
- ❑ Примем алгоритм *IntTrap04* за «наилучший» последовательный

Параллельный алгоритм интегрирования

- ❑ Метод геометрического параллелизма?
- ❑ Метод коллективного решения?
- ❑ ?

Метод геометрического параллелизма

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

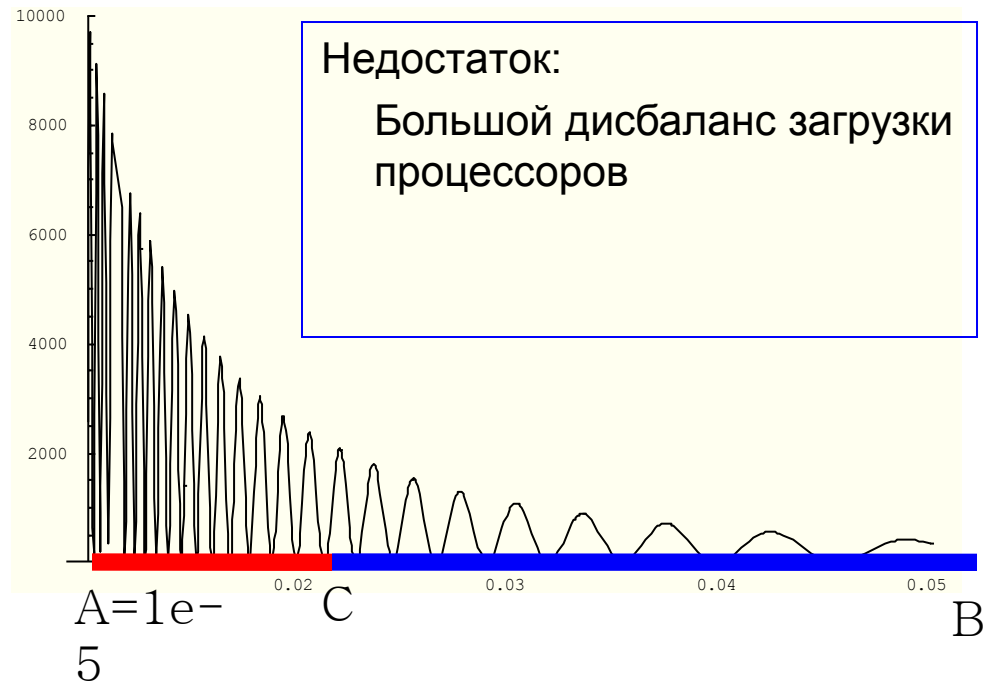
  WaitAllParallelProcess
  J=0

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

Недостаток:

Значительный дисбаланс
загрузки процессоров

Расчет интеграла на разных отрезках



$$f(x) = \frac{1}{x^2} \sin^2\left(\frac{1}{x}\right)$$

$p,$ $(B-C)/(C-A)$	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

Метод коллективного решения

```
main ()
{
    // Порождение p параллельных процессов,
    // каждый из которых выполняет процедуру slave

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

    i=0 // число переданных для обработки интервалов
        // n - число отрезков интегрирования

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

    // J - значение интеграла на всем интервале [A,B]
    J=0
```

Недостаток:

$n = ?$

- Либо большой дисбаланс загрузки процессоров
- Либо большой объем лишних вычислений

Пока есть отрезки, не переданные для обработки, следует дожидаться 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)
{
    k = Recv(slave ANY, s)
    J+=s
    Send(slave k, 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)
            }
}
```

Практически непригодны для решения
поставленной задачи методы

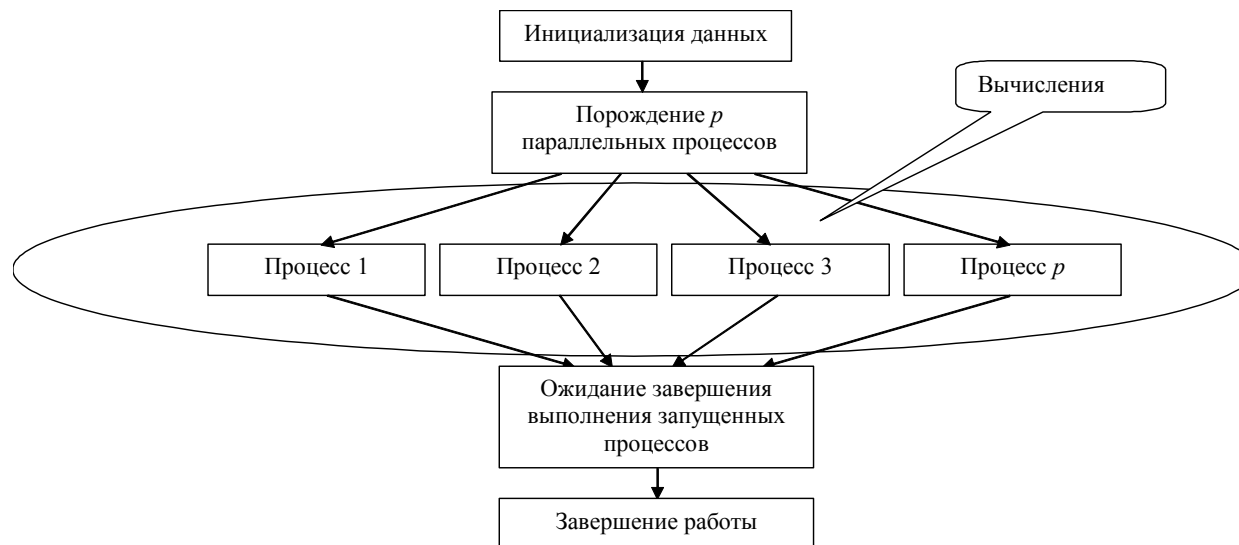
геометрического параллелизма
(статическая балансировка)

и

коллективного решения
(динамическая балансировка)

Метод глобального стека

- ❑ Вычислительные системы с общей памятью
- ❑ Динамическая балансировка загрузки
- ❑ Отсутствие централизованного управления

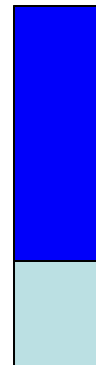


Стеки алгоритма

□ Глобальный стек



□ Локальные стеки



IntTrap04p

IntTrap04p

IntTrap04p

IntTrap04p

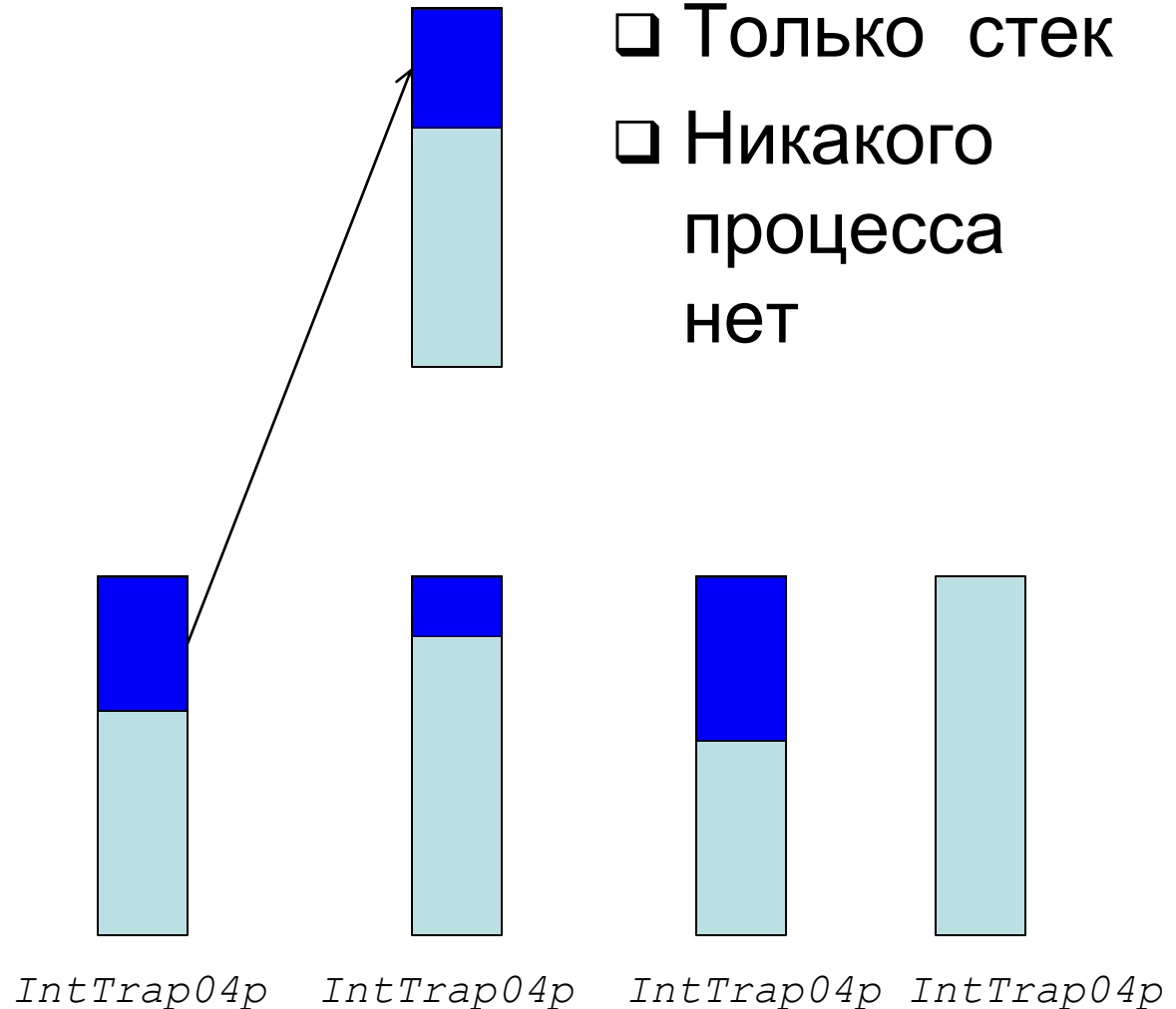
Стеки алгоритма

□ Глобальный стек

□ Только стек

□ Никакого
процесса
нет

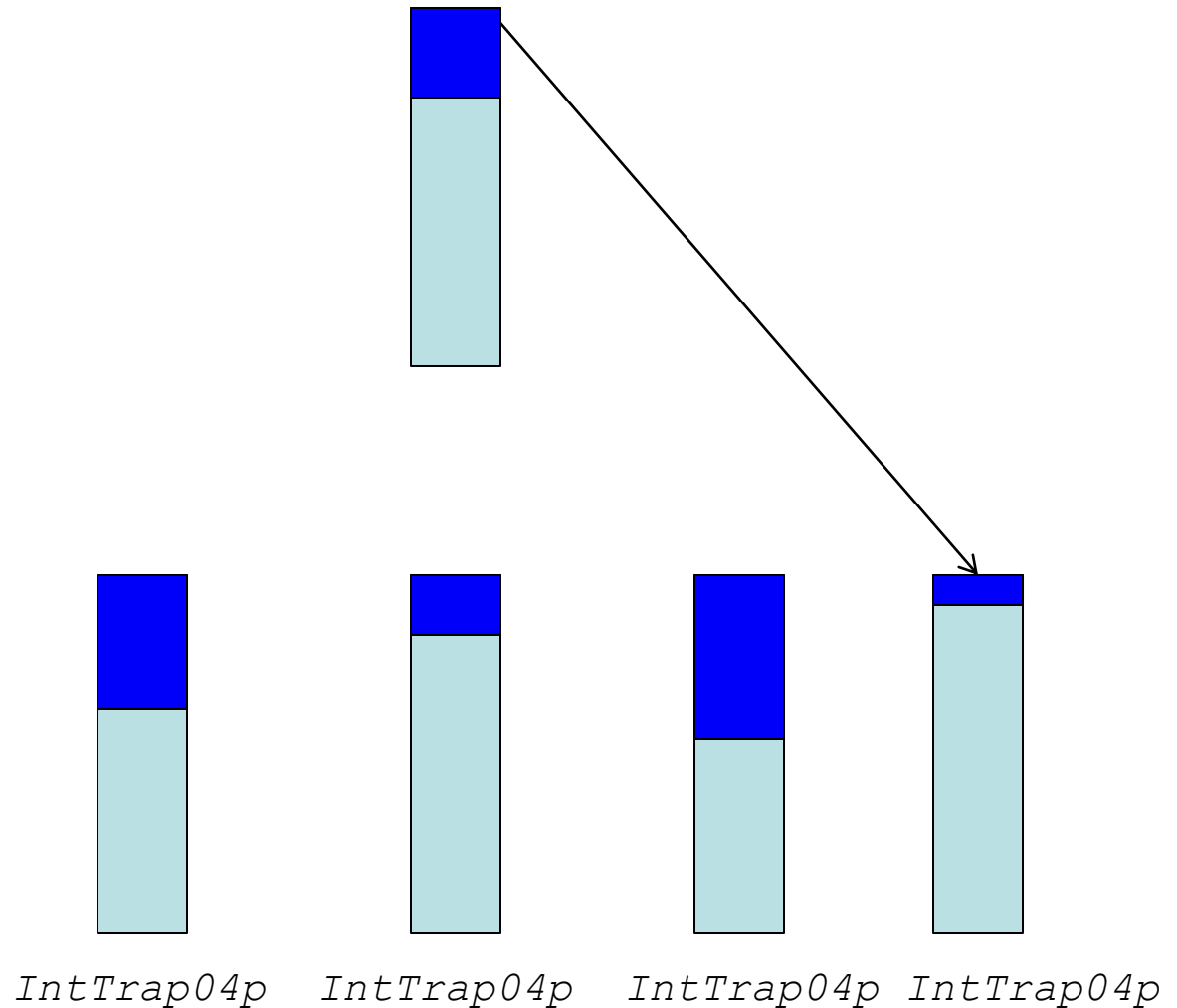
□ Локальные стеки



Стеки алгоритма

□ Глобальный стек

□ Локальные стеки



Идея алгоритма

- ❑ Пусть есть доступный всем параллельным процессам список отрезков интегрирования, организованный в виде стека. Назовем его **глобальным стеком**.
- ❑ Пусть у каждого процесса есть свой, доступный только этому процессу локальный стек
- ❑ Перед запуском параллельных процессов в глобальный стек помещается единственная запись (в дальнейшем "**отрезок**"):
 - координаты концов отрезка интегрирования,
 - значения функции на концах,
 - приближенное значение интеграла на этом отрезке.
- ❑ 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, но,

если при записи в **локальный стек** в нем есть несколько отрезков, а в глобальном стеке **отрезки** отсутствуют, то:

- переместить часть **отрезков** из **локального стека** в **глобальный стек**.

Вопросы

- какую часть отрезков следует перемещать из локального стека в глобальный стек?
- в какой момент интеграл вычислен?
- что должен делать процесс у которого пуст локальный стек, если глобальный стек тоже пуст?
 - должен ли процесс закончить работу, если в его локальном и в глобальном стеке отрезков нет?

Схема Интегрирующего процесса

IntTrap04p()

```
{  
    // цикла обработки стека отрезков  
  
    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)  
  
        ИНТЕГРИРОВАНИЕ ОДНОГО ОТРЕЗКА  
    }  
  
    sdat.s_all = sdat.s_all + s  
}
```

Правильное определение общей суммы

```
main()  
{  
.  
Sem_init(sdat.sem_sum,1) //доступ к глобальной сумме открыт  
.  
}  
  
IntTrap04p()  
{  
...  
// Начало критической секции сложения частичных сумм  
sem_wait(sdat.sem_sum)  
sdat.s_all = sdat.s_all + s  
sem_post(sdat.sem_sum)  
// Конец критической секции сложения частичных сумм  
}
```

Схема Интегрирующего процесса

IntTrap04p()

```
{  
    // цикла обработки стека отрезков  
  
    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)  
  
        ИНТЕГРИРОВАНИЕ ОДНОГО ОТРЕЗКА  
    }  
    sem_wait(sdat.sem_sum)  
    sdat.s_all = sdat.s_all + s  
    sem_post(sdat.sem_sum)  
}
```

Схема интегрирования отрезка

```
while(1) // интегрирование одного отрезка
{
    c=(a+b)/2;    fc=f(c)

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

    if(!BreakCond(sacb,sab)) // Точность на части отрезка достигнута
    {
        st=sacb
        if(sp==0) break; // локальный стек пуст, выход

        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
    }

    if((sp>SPK) && (!sdat.ntask)) // перемещение части локального стека в общий список интервалов
    {
        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++
        }
    }
}
```

Схема интегрирования отрезка

// интегрирование одного отрезка

while(1)

{

Инициализация

Точность на части отрезка достигнута?

Добавлять отрезки в глобальный стек?

}

c=(a+b)/2;

fc=f(c)

sac=(fa+fc)(c-a)/2*

scb=(fc+fb)(b-c)/2*

sacb=sac+scb

Схема интегрирования отрезка

```
// интегрирование одного отрезка
while(1)
{
    Инициализация
    Точность на части отрезка достигнута?
    Добавлять отрезки в глобальный стек?
}
```

```
if(!BreakCond(sacb,sab))

{ // Точность на части отрезка достигнута
  s+=sacb
  if(sp==0) break; // локальный стек пуст, выход

  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
}
```

Схема интегрирования отрезка

// интегрирование одного отрезка

while(1)

{

Инициализация

Точность на части отрезка достигнута?

Добавлять отрезки в глобальный стек?

}

*if((sp>SPK) && (!sdat.ntask)) // перемещение части
локального стека в общий список интервалов*

{

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++

}

}

Преждевременное окончание работы процесса

```
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
    ...
}
```

Преждевременный выход

- ❑ Плохое условие выхода из *цикла обработки стека интервалов*
- ❑ Интегрирующие процессы не должны заканчивать работу до тех пор, пока все отрезки интервала интегрирования не будут полностью обработаны
- ❑ Преждевременное завершение работы приведет к получению верного ответа, но за большее время

Если глобальный и локальный стеки пусты

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.

Необходимые данные

	<i>семафоры доступа:</i>
<i>sdat.sem_list</i>	семафор доступа к глобальному стеку отрезков
<i>sdat.s_all</i>	семафор доступа к значению интеграла

	<i>семафоры состояния:</i>
<i>sdat.sem_task_present</i>	семафор наличия записей в глобальном стеке отрезков

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

Параллельный алгоритм: метод глобального стека

```
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

                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
            }
        }

        // одного интервала

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

Инициализация

```
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
```

Доступ процесса *IntTrap04p* к глобальному стеку

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

    sem_wait(sdat.sem_task_present)

    // чтение одного интервала из списка интервалов
    sem_wait(sdat.sem_list)

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

    if(sdat.ntask) // в глобальном стеке остались задания
        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)

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

Запись терминальных отрезков

```
// Начало критической секции заполнения глобального
// стека терминальными отрезками (a>b)

sem_wait(&sdат.sem_list)
    sdат.nactive--

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

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

sem_post(sdат.sem_list)
//
// Конец критической секции заполнения глобального
// стека терминальными отрезками
```

Результаты тестирования

$$J = \int_{10^{-5}}^1 \frac{1}{x^2} \sin^2\left(\frac{1}{x}\right)$$

Время выполнения

<i>Np</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>tiger.jscs.ru</i>	<i>31.39</i>	<i>15.61</i>	<i>10.29</i>	<i>7.83</i>
<i>ga03.imamod.ru</i>	<i>37.48</i>	<i>19.00</i>	-	-

Ускорение

<i>Np</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>tiger.jscs.ru</i>	<i>1</i>	<i>2.01</i>	<i>3.05</i>	<i>4.01</i>
<i>ga03.imamod.ru</i>	<i>1</i>	<i>1.97</i>		

Эффективность

<i>np</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>tiger.jscs.ru</i>	<i>100%</i>	<i>101%</i>	<i>102%</i>	<i>100%</i>
<i>ga03.imamod.ru</i>	<i>100%</i>	<i>99%</i>		

Вопросы

- какую часть отрезков следует перемещать из локального стека в глобальный стек?
- в какой момент интеграл вычислен?
- что должен делать процесс у которого пуст локальный стек, если глобальный стек тоже пуст?
 - должен ли процесс закончить работу, если в его локальном и в глобальном стеке отрезков нет?
- На сколько отрезков изначально разбить отрезок?

Заключение

- ❑ Рассмотрен ряд методов вычисления интегралов на многопроцессорных системах, проанализированы их преимущества и недостатки
- ❑ Показано, что методы геометрического параллелизма и коллективного решения неприменимы для эффективного численного интегрирования функций общего вида
- ❑ Показано, что идея децентрализованного управления расчетом эффективно решает задачу

Литература

- Якововский М.В. Введение в параллельные методы решения задач: Учебное пособие / Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2013. – 328 с., илл. – (Серия «Суперкомпьютерное образование») ISBN 978-5-211-06382-2

- *Якововский М.В., Кулькова Е.Ю.* Решение задач на многопроцессорных вычислительных системах с разделяемой памятью. - М.: СТАНКИН, 2004. – 30 с.
http://www.imamod.ru/~serge/arc/stud/Jakob_2.pdf

Якобовский М.В.

проф., д.ф.-м.н.,

зав. сектором

«Программного обеспечения многопроцессорных
систем и вычислительных сетей»

Института прикладной математики им.
М.В.Келдыша Российской академии наук

[mail: lira@imamod.ru](mailto:lira@imamod.ru)

[web: http://lira.imamod.ru](http://lira.imamod.ru)