**Лабораторная 4. Решение задачи распараллеливания в MATLAB**

Задание:

1. Выполнить решение задачи с помощью интерактивного режима для любого числа процессов;

2. Выполнить решение задачи с помощью m-файла и функции для любого числа процессов;

3. Выполнить решение с помощью spmd и обмена сообщениями для любого числа процессов;

4. Выполнить решение с помощью parfor.

5. Выполнить решение с помощью GPU.

В каждом задании наряду с параллельным решением привести последовательное.

Исходные данные для заданий 3-5 генерируются с помощью датчика случайных чисел, где **100000<n<1000000, 100<An<100000**. Результаты сравниваются по времени выполнения при разном числе процессов и оформляются в виде таблицы вида

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Последовательное решение, время** | **Число процессов** | **Spmd, время** | | **Parfor, время** | | **Параллельная задача, время** |
| **CPU** | **GPU** | **CPU** | **GPU** |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Варианты для задания 1

## 1. 2. . 3.

4.  . 5. . 6.  .

7.  . 8.  . 9. .

10. . 11. . 12. .

13.  . 14. . 15. 

16. . 17. . 18. 

Варианты для задания 2

|  |  |  |
| --- | --- | --- |
| 1. | 2. | 3. |
| 4. | 5. | 6. |
| 7. | 8. | 9. |
| 10. |  |  |

Варианты для заданий 3-5

1. Даны последовательности чисел А = {а0…аn–1} и С = {с0…сn–1}. Создать приложение, определяющее, совпадают ли поэлементно элементы массивов А и С.

2. Дана последовательность чисел С = {с0…сn–1}. Дан набор из N пар кодирующих чисел (ai,bi), т.е. все ai заменяются на bi. Создать приложение, кодирующее последовательность следующим образом: массив разделяется на подмассивы и каждый процесс осуществляет кодирование своего подмассива. Для задания 5 допускается ограничиться фиксированным набором кодирующих чисел.

3. Дана последовательность чисел С = {с0…сn–1} и число b. Создать приложение для определения количества вхождений числа b в массив C.

4. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска произведения чисел a0\*а1\*…\*an–1.

5. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска максимального ai.

6. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска минимального ai.

7. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска всех ai, являющихся простыми числами.

8. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска всех ai, являющихся квадратами любого натурального числа.

9. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для вычисления выражения a0-а1+a2-а3+a4-а5+...

10. Дана последовательность натуральных чисел {a0…an–1}. Создать приложение для поиска суммы ∑ai, где ai– четные числа.

**Средства распараллеливания в пакете MATLAB**

MATLAB Parallel Computing Toolbox (PCT) – это набор специальных средств и функций для написания параллельных алгоритмов и организации распределенных вычислений, который позволяет использовать как локальные многопроцессорные, так и распределенные вычислительные ресурсы. PCT предназначен для разработки параллельных алгоритмов, для формирования распределенной задачи, передачи её на сервер и для приема результатов вычислений. Основная задача формулируется на языке MATLAB и может использовать функции любых пакетов расширения. Также пакет поддерживает интерфейс передачи сообщений MPI, что позволяет разрабатывать на MATLAB эффективные приложения с параллельными вычислениями. Серверная часть пакета - [MATLAB Distributed Computing Server](http://sl-matlab.ru/services/products/detail.php?ID=417&list=c) (DST) обеспечивает исполнение основной задачи на удаленных сессиях MATLAB. Планировщик MathWorks Job Manager распределяет ежду исполнителями подзадачи. Он координирует выполнение заданий и асинхронно распределяет задачи рабочим станциям – исполнителям. Планировщик устанавливается на любой машине находящейся в сети и может обрабатывать задания различных пользователей различных платформ. Принцип динамического лицензирования позволяет использовать при разработке распределенных приложений любые пакеты расширений MATLAB имеющиеся на клиентской стороне.

**Работа (job)** – это некоторая сложная операция, которую нужно выполнить в сеансе MATLAB. Работа разбивается на сегменты, называемые **заданиями (tasks)**. Пользователь решает, как лучше разделить работу на задания. Работа делится на идентичные задания, но они выполняются не одинаково.

Сессия MATLAB , в которой определены работа и задания, называется **клиентской сессией (client)**. Клиент использует **PCT** для определения работы и заданий. MATLAB DST выполняет работу путем вычисления заданий и возврата результата в клиентскую сессию (рис. 1).

**Jobmanager** - это часть программного обеспечения, которая координирует выполнение работ и вычисление их заданий*.* **Jobmanager** распределяет задания для выполнения индивидуальным сессиям MATLAB, называемым **рабочими (workers)**.

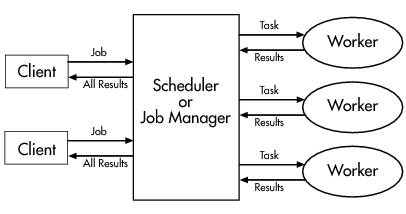


Рис. 1. Взаимодействие объектов при параллельном решении задач

Можно конфигурировать один компьютер как клиента, рабочего или планировщика (**jobmanager**).

При запуске сеанса MATLAB ему доступна только конфигурация **'local'**, позволяющая производить вычисления на локальном компьютере.

Ее можно настраивать с помощью пункта меню **Parallel, Manage Configuration.**

Двойной щелчок по конфигурации открывает окно настройки (рис. 2)

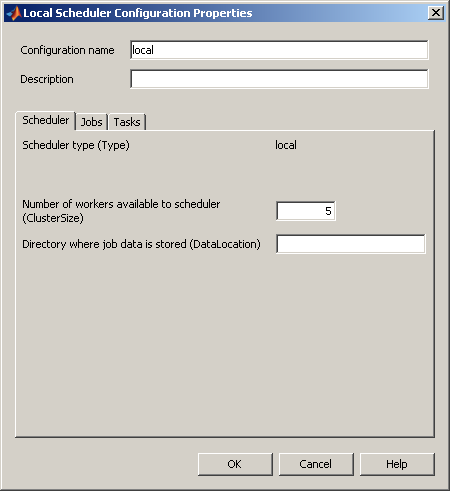


Рис. 2. Окно настройки конфигурации

В нем следует задать число рабочих сеансов, доступное планировщику

**Способы распараллеливания процессов в пакете MATLAB**

**Интерактивное выполнение рабочих процессов**

Это режим **pmode**, с помощью которого непосредственно из командного окна MATLAB становится возможным обращение к процессам **workers**, просмотр их локальных переменных, обмен данными между ними. В режиме **pmode** команды, вводимые в рабочем окне MATLAB, будут исполняться всеми рабочими процессами (потоками), называемыми также **labs**, ассоциированными с соответствующим **jobmanager**.

При окончании сессии **pmode**, ее работа разрушается, и вся информация и данные из **labs** теряются. Начало любой сессии **pmode** всегда начинается с начального состояния.

Режим **pmode** запускается командой вида:

pmode start local 4

При этом начинается выполнение 4 параллельных процессов, создается параллельная работа для запуска на этих потоках и открывается **Parallel Command Window** (рис.3).

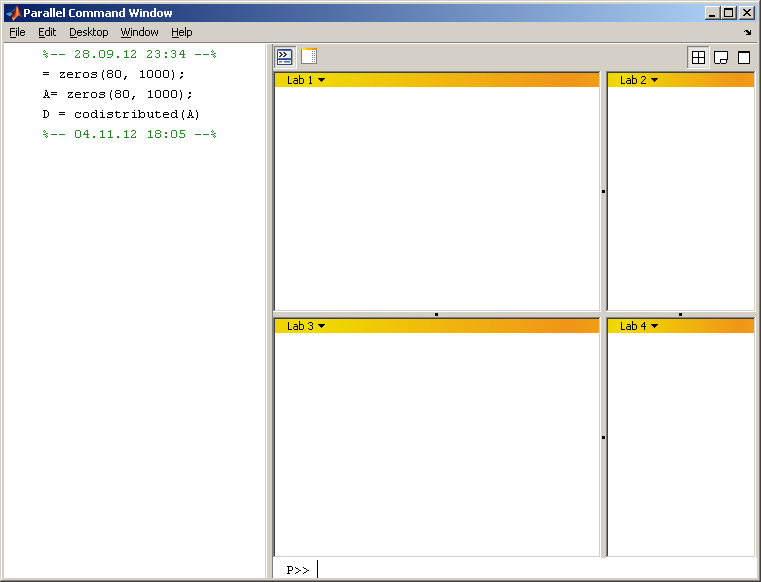
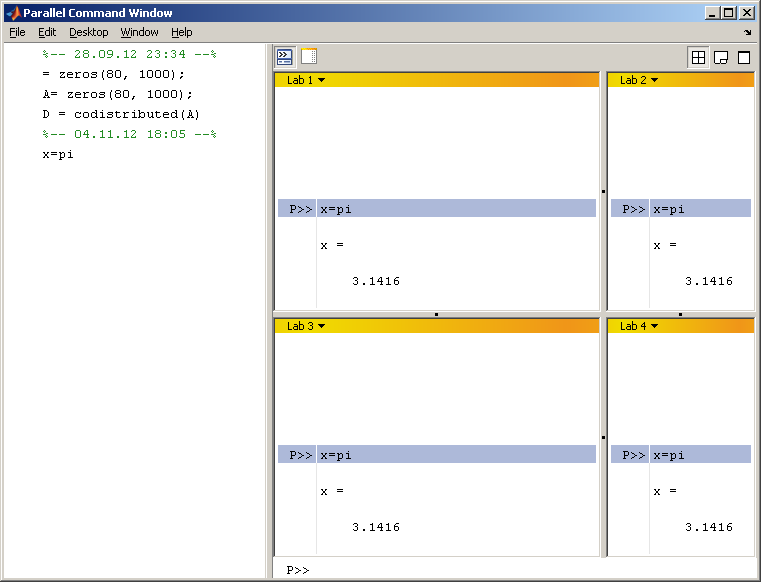


Рис. 3. Окно параллельных задач

Присвоим некоторой переменной значение в строке приглашения интерактивного режима выполнения параллельных процессов:

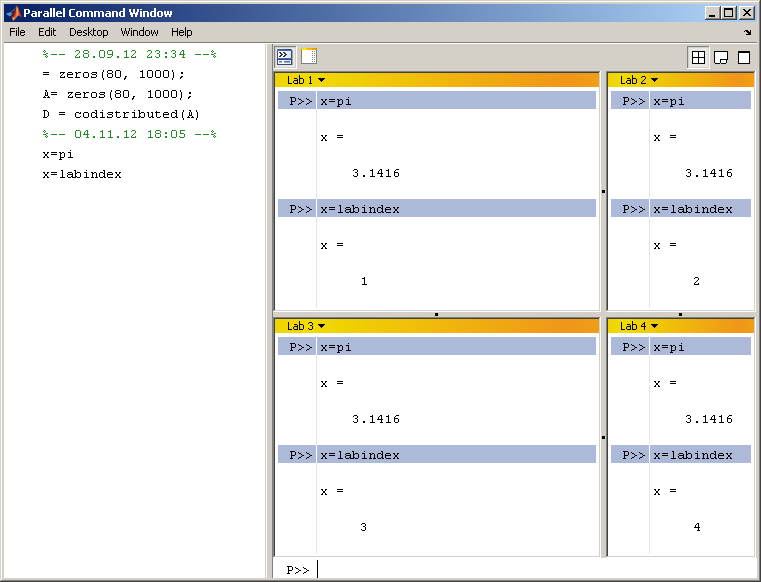
x = pi

Значение присвоится во всех параллельных процессах.



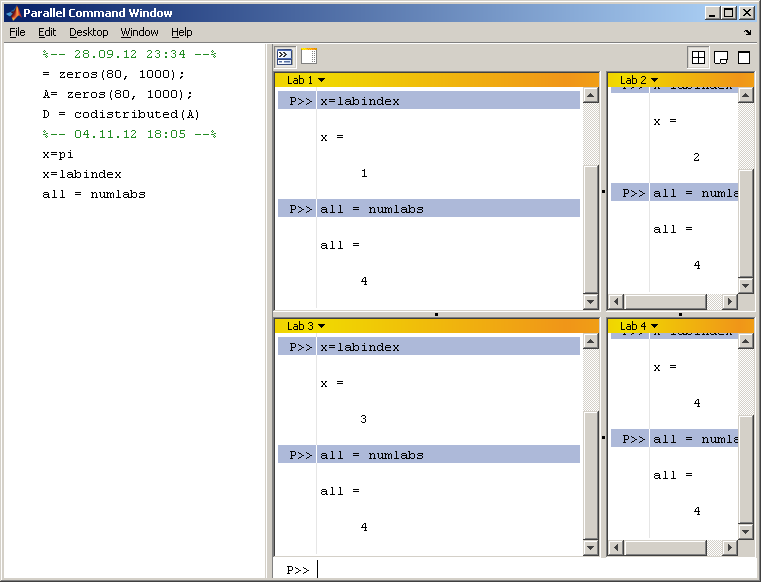
Переменная не всегда имеет одно и то же значение в каждом параллельном процессе. Функция [**labindex**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/labindex.html) возвращает номер каждого процесса, работающего в параллельном режиме. В примере переменная x получает различные значения в рабочем пространстве каждого потока.

x = labindex



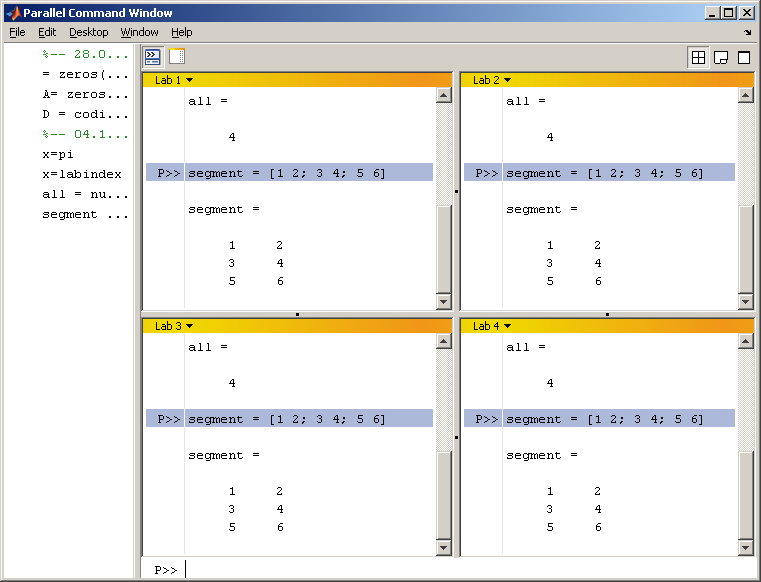
Получим значение общего числа потоков, работающих в текущей параллельной работе с помощью функции [**numlabs**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/numlabs.html).

all = numlabs



Создадим массив на всех параллельных процессах:

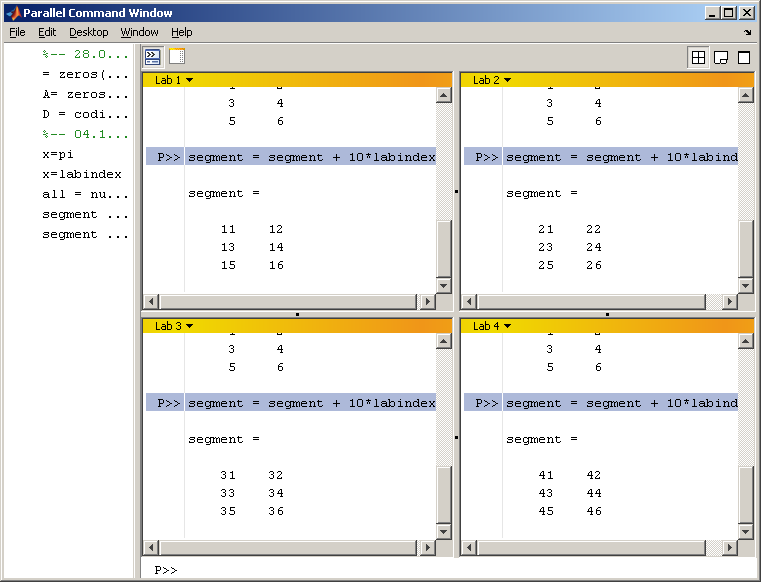
segment = [1 2; 3 4; 5 6]



Назначим уникальное значение массиву в зависимости от номера потока:

segment = segment + 10\*labindex

Так как он имеет различные значения в каждом потоке, это массив типа - **variant**.



До этого момента в примере массивы variant независимы.

Используем функцию codistributed.build для объединения отдельных массивов в общий синхронизированный массив, распределённый между исполнителями.

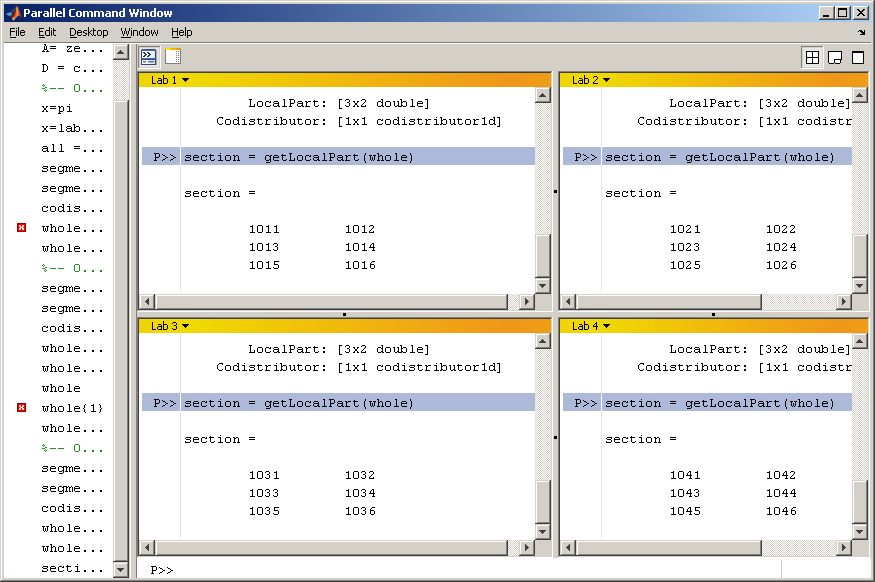
codist = codistributor1d(2, [2 2 2 2], [3 8])

whole = codistributed.build(segment, codist)

Тем самым мы объединили массивы размерности 3x2 в один распределённый массив размерности 3 на 8. Применение объекта codistributor1d указывает на то, что массив распределён по своему второму измерению (столбцам), по 2 столбца на каждого исполнителя. На каждом исполнителе находятся данные относящейся к нему локальной части общего массива (порции).

Можно использовать функцию [**getLocalPart**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/getlocalpart.html)**()** для доступа к порции такого массива в конкретном потоке:

section = getLocalPart(whole)



Переменная **section** - массив типа **variant**, так как он различен в каждом потоке.

Если есть необходимость использовать весь массив в одном рабочем пространстве, следует использовать функцию [**gather**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/gather.html)**()**.

combined = gather(whole)

Заметим, что это объединяет весь массив в рабочем пространстве всех потоков. Можно собрать массив в рабочем пространстве только одного параллельного процесса.

Конвертация **codistributed** массива C в **variant** массив X так, что все данные содержатся в потоке с указанным номером, а X размера 0 строк и 0 столбцов содержится во всех остальных потоках:

X = gather(C, номер потока)

Так как вывод во всех параллельных процессах обычно невозможен, для графического использования данных следует получить данные в рабочем пространстве клиента. Копирование массива в клиентское пространство выполняется командой:

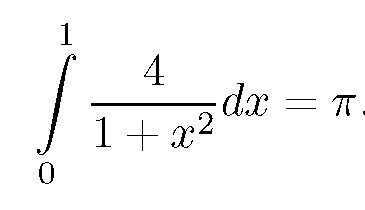
pmode lab2client combined 1

Если требуется, переменную, хранящуюся в сессии рабочего процесса, можно экспортировать в текущую (пользовательскую) сессию МАТLАВ. Команда **lab2client** позволяет копировать определенную пере­менную из любого процесса в пользовательскую сессию.

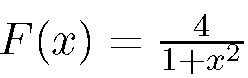
Выход из режима **pmode** и возврат в сеанс клиента происходит по команде:

pmode exit

Покажем решение задачи вычисления числа π как значения определенного интеграла:

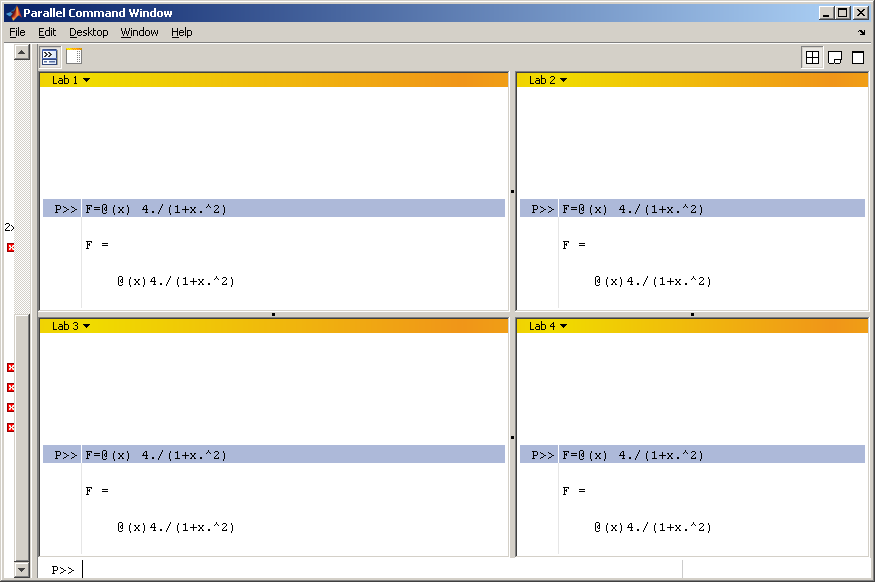


Чтобы проинтегрировать функцию , следует воспользоваться тем свойством, что каждый из доступных рабочих процессов может интегрировать функцию *Р(х)* на своем участке интервала.



Очевидно, что первым шагом решения задачи должно быть распространение функции F*(х)* на все рабочие процессы. Вводим команду:

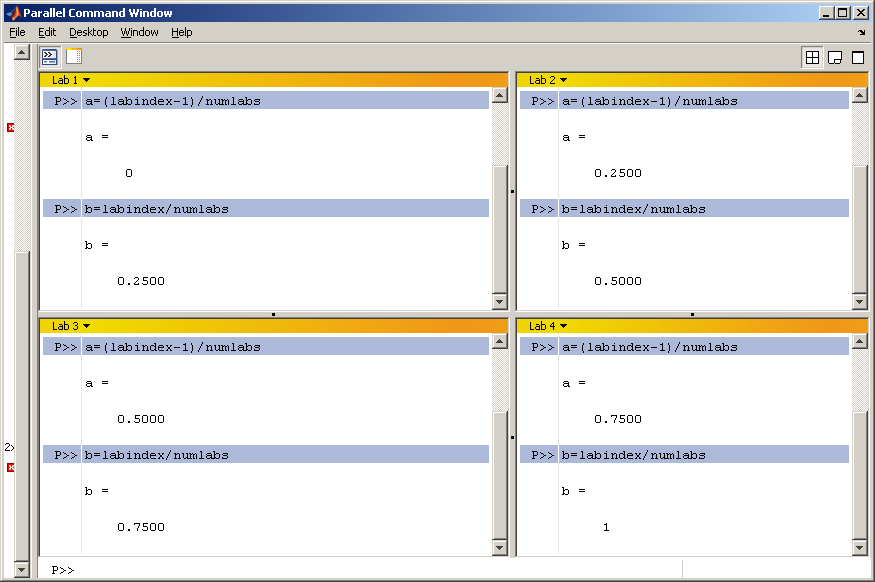
F=@(x) 4./(1+x.^2);



и команды, определяющие интервал интегрирования для каждого процесса

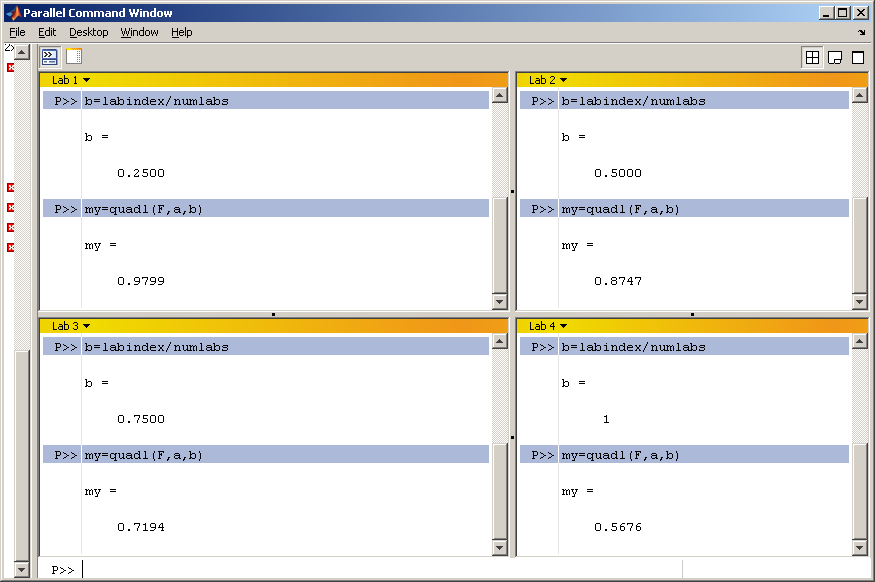
a = (labindex-1)/numlabs;

b = labindex/numlabs;



Теперь, воспользовавшись функцией **quadl(F,a,b)**, в качестве параметров которой передаются подынтегральная функция и границы ин­тегрирования, вычислим значение определенного интеграла в каждом из рабочих процессов:

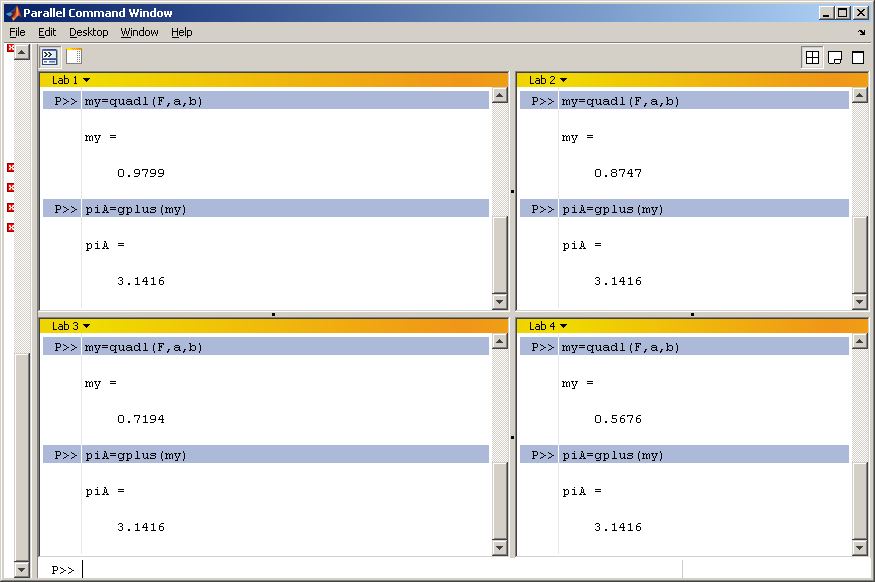
my=quadl(F,a,b)



После того как в каждой сессии определена переменная my, необходимо воспользоваться функцией **gplus()** и произвести глобальное суммирование переменной my по всем процессам.

**gplus(х)** возвращает сумму значений переменной х по всем процессам. Результат тиражируется на всех сессиях процессов:

piA=gplus(my)



Если требуется, переменную, хранящуюся в сессии рабочего процесса, можно экспортировать в текущую (пользовательскую) сессию MATLAB. Команда **lab2client** позволяет копировать определенную пере­менную из любого процесса в пользовательскую сессию.

pmode lab2client piA 1

Итогом рассмотренной задачи является переменная piA, определенная в локальной сессии MATLAB. Для сравнения полу­ченного значения со значением константы рi, которое хранится в системе МАТLАВ, рассмотрим разность piА - pi:

аns =2.4866е-010

pmode lab2client piA 1

piA

piA = 3.1416

**Режим создания параллельных задач**

В этом режиме с помощью команд создаются работа, задача и запускается некоторая функция.

Вначале отыскивается имеющийся в текущей конфигурации планировщик (jobmanager) с помощью команды вида:

jm = findResource('scheduler',

'type','jobmanager','configuration','local')

Одним из методов объекта **jobmanager** является **createParallelJob()**.

pjob=jm.createParallelJob();

Переменная pjob является ссылочной переменной и ссылается на определенный участок оперативной памяти, выделенной для системного процесса планировщика.

В силу того, что задача будет отправляться на вычисление че­рез планировщик jm (у которого в свойствах указано, сколькими занятыми и сколькими свободными рабочими процессами он об­ладает в данный момент времени), уже в самой задаче должно быть указано, сколько свободных процессов требу­ется планировщику для начала вычислений, описанных в задаче pjob.

Значение свойств MaximumNumberOfWorkers и

MinimumNumberOfWorkers есть положительное целое число. Установим его равным двум, то есть вычисления в задаче pjob начнутся, как только у планировщика jm появятся два свободных рабочих процесса.

Запись значений этим свойствам можно выполнить с помощью команд

set(pjob,'MinimumNumberOfWorkers',2);

set(pjob,'MaximumNumberOfWorkers',2);

Параллельное задание есть не что иное, как **m-файл**, который будет исполняться одновременно всеми рабочими процессами. Передать m-файл напрямую рабочему процессу нельзя, для этой цели необходимо использовать объект Task (параллельное задание). Объект Task является дочерним объектом по отношению к объекту paralleljob и может быть создан посредством метода createTask.

Создадим m-файл и опишем в нем функцию, ко­торая будет выполняться каждым процессом. Пусть это будет вычисление суммы сходящегося ряда , общий член которого имеет вид .



function y=par\_sum(x)

ibeg=(labindex-1)\*20+1;

iend=labindex\*20;

if labindex==numlabs

iend=40;

end

z=0;

for n=ibeg:iend

xn=x;

for i=0:n

xn=xn\*x;

end

f=1;

for i=1:n

f=f\*i;

end

z=z+xn/f;

end

y=gplus(z);

end

После того как написали m-файл, следует указать в свойстве FileDependencies объекта pjob имя файла, который будет выполняться рабочими процессами

set(pjob, 'FileDependencies', {'par\_sum.m'});

После этого создаем переменную, которая будет являться подза­дачей. Во входных параметрах требуется указать переменную - параллельную задачу, количество выходных аргументов (в рассматриваемом примере - 1) и входные аргументы {x}:

obj = createTask(pjob,'par\_sum',1,{x})

Теперь все, что осталось сделать - это выполнить следующий участок кода:

submit(pjob); waitForState(pjob);

Команда submit отправляет задачу pjob планировщику jobmanager. Задача начинает решаться рабочими процессами не сразу, а только тогда, когда свободными окажутся столько процессов, сколько указано в свойстве MinimumNumberOfWorkers. До этого момента задача находится в очереди. Как только количества свободных процессов оказалось достаточно, рабочие процессы начинают выполнять один и тот же m-файл. В этот момент свойство State объекта pjob принимает значение running. Свойство поменяет свое значение на finished только после того, как задача, описанная в m-файле, выполнится. Это произойдет после выполнения команды waitForState.

Как только задача поменяла статус на finished, можно воспользоваться методом getAllOutputArguments() и получить данные в локальной сессии MATLAB.

results = getAllOutputArguments(pjob)

Для завершения параллельной работы выполняется команда:

destroy(pjob);

Передача сообщений между рабочими сеансами

В приложении PCT систе­мы MATLABимеется встроенный набор функций для передачи сообщений между процессами, основанный на интерфей­се передачи сообщений MessagePassingInterface (MPI).

Посылка данных другому параллельному процессу:

labSend(данные любого типа, номер процесса - получателя)

Получение данных от другого параллельного процесса:

данные = labReceive

данные = labReceive(номер lab)

Пример - суммирование элементов массива с подсчетом подсумм в отдельных рабочих процессах. Запускающий m-файл:

%ссылка на планирощик%

jm = findResource('scheduler','type','jobmanager','configuration','local');

%создаем пар. задачу,pjob-сслылочная переменнная , кот ссылается на определенный

%участок оперативной памяти,выделенной для системного процесса планировщика)

pjob=jm.createParallelJob();

set(pjob,'MinimumNumberOfWorkers',2);

set(pjob,'MaximumNumberOfWorkers',2);

%создание связи между файлами

set(pjob, 'FileDependencies', {'par\_sum.m'});

x=[1 2 3 4 5 6 7 8 9 10];

%создаем переменную которая будет являться подзадачей

obj = createTask(pjob,'par\_sum',1,{x});

%запускаем параллельную задачу

submit(pjob);

waitForState(pjob);

%получения данных локальной сессии

results = getAllOutputArguments(pjob);

%завершение паралельной работы

destroy(pjob);

s=0;

for i=1:length(x)

s=s+x(i);

end

s

Функция par\_sum в файле par\_sum.m:

function z=par\_sum(x)

z=0;

% Вычисление частичной суммы на каждом из процессов

% на каждом процессе суммируются элементы вектора x от i1 до i2

k =fix(m / numlabs);

i1 = k \* (labindex-1)+1;

i2 = k \* ( labindex );

if ( labindex == numlabs )

i2 = m;

end

ProcSum=0;

for i = i1:i2

ProcSum = ProcSum + x(i);

end

% Сборка частичных сумм на процессе с номером 1

if ( labindex == 1 )

TotalSum = ProcSum;

for i=2:numlabs

ProcSum=labReceive(i);

TotalSum = TotalSum + ProcSum;

end

z=TotalSum;

else

% Все процессы отсылают свои частичные суммы

labSend(ProcSum, 1);

end

**Запуск одной программы во всех рабочих процессах (блок spmd)**

**SPMD (single program - multiple data, одна программа – много данных)** позволяет незаметно чередовать обычное и параллельное программирование. Оператор **spmd** позволяет определить блок кода для запуска на многих рабочих процессах.

"Одна программа" означает, что один и тот же программный код запускается на различных рабочих процессах. Когда блок **spmd** завершается (по оператору **end**) программа продолжает свою работу на клиенте.

"Много данных" означает, что хотя код блока **spmd** идентичен на всех lab, каждый рабочий процесс может использовать уникальные данные.

Типичные случаи применения **spmd** следующие:

* Программы, имеющие длительное время выполнения — **spmd** позволяет нескольким lab выполнять алгоритм одновременно.
* Программы, оперирующие с большими наборами данных — **spmd** позволяет разделить данные между многими рабочими процессами.

Перед запуском **spmd** нужно выполнить команду [**matlabpool**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/matlabpool.html) для создания некоторого числа рабочих процессов MATLAB (lab) для выполнения блоков **spmd**. lab могут выполняться удаленно на кластере или локально на клиентской машине. Команда **matlabpool** запускает столько рабочих процессов, сколько их определено в конфигурации **local**.

Можно запустить команду вида:

matlabpool 3

для запуска конкретного числа рабочих процессов. Их число на локальном компьютере не более 8.

Когда заканчивается использование пула, он закрывается по команде:

matlabpool close

Определение блока spmd имеет вид:

spmd

операторы

end

Например, создадим случайную матрицу:

matlabpool 3

spmd

R = rand(4,4);

end

matlabpool close

Каждый поток, использующий **spmd**, имеет уникальное значение [**labindex**](jar:file:///C:/Program%20Files/MATLAB/R2009b/help/toolbox/distcomp/help.jar%21/labindex.html). Это позволяет определить уникальный фрагмент кода для некоторых потоков.

Например, создадим массивы различного размера в зависимости от **labindex**:

spmd

if labindex==1

R = rand(9,9);

else

R = rand(4,4);

end

end

**Цикл parfor**

Фундаментальное понятие цикла **parfor** в MATLAB такое же, как и у стандартного цикла **for**: MATLAB выполняет ряд операций (тело цикла) в диапазоне значений. Часть тела цикла **parfor** выполняется на клиенте MATLAB (где **parfor** запущен), и часть выполняется параллельно на рабочих процессах MATLAB.

Необходимые данные, которые обрабатывает **parfor**, отправляются от клиента рабочим процессам, где и происходит большая часть вычислений, затем результаты отсылаются назад клиенту и объединяются.

Поскольку несколько рабочих MATLAB могут обрабатывать данные одновременно в теле одного цикла, **parfor** может обеспечить значительно лучшую производительность, чем его аналог цикл **for**. Каждое выполнение тела **parfor**­цикла ­ итерация. Рабочие MATLAB выполняют итерации беспорядочно и независимо друг от друга. Поскольку все итерации независимы, нет никакой гарантии синхронизации итераций, так же как и нет потребности в этом. Если число рабочих равно числу итераций цикла, каждый рабочий выполняет одну итерацию. Если итераций больше, чем рабочих, некоторые рабочие выполняют более одной итерации; в этом случае, рабочий может получить несколько итерации сразу, чтобы уменьшить время коммуникации.

Цикл **parfor** полезен в ситуациях, где есть необходимость выполнить много простых итераций.

**Parfor** делит итерации цикла на группы так, чтобы каждый рабочий выполнял некоторую часть общего количества итераций. Циклы **parfor** также полезны, когда выполнение итерации цикла занимает много времени, ведь рабочие могут выполнять итерации одновременно. Цикл **parfor** нельзя использовать, когда итерации цикла зависят от результатов других. Каждая итерация должна быть независимой от всех других. Поскольку некоторое количество времени тратится на коммуникации этот цикл не целесообразно использовать для небольшого количества простых вычислений.

**Создание цикла parfor**

Цикл **for**, в котором все итерации полностью независимы друг от друга - хороший кандидат на **parfor**­цикл.

В основном, если одна итерация зависит от результатов другой, не представляется возможным преобразовать такой цикл в **parfor**.

Этот пример показывает эквивалентные, последовательный (сверху) и параллельный (снизу) циклы:

clear A

for i = 1:8

A(i) = i;

end

A

clear A

parfor i = 1:8

A(i) = i;

end

A

В этом примере видно, что каждая итерация зависит только от своего номера, и не зависит от результатов других итераций.

**Различия между циклами for и parfor**

Поскольку циклы **parfor** ­ не совсем то же самое, что и циклы **for**, есть некоторые особенности их использования. Из предыдущего примера можно заметить, что массив, проиндексированный в теле цикла, доступны после его завершения. Однако предположим, что в цикле используется неиндексированная переменная. Рассмотрим следующий пример:

clear A

d = 0; i = 0;

for i = 1:4

d = i\*2;

A(i) = d;

end

A

d

i

clear A

d = 0; i = 0;

parfor i = 1:4

d = i\*2;

A(i) = d;

end

A

d

i

Хотя элементы массива А будут одинаковыми в обоих случаях, значения переменной d будут различными. В цикле слева итерации выполняются последовательно, таким образом, d будет иметь значение, принятое в последней итерации цикла. В **parfor**­цикле справа, итерации выполняются параллельно, а не последовательности, таким образом невозможно определить значение d до завершения цикла. Это также относится переменной цикла, i. Поэтому, поведение цикла **parfor** определяется так, что оно не затронуло значения d и i вне цикла вообще. Их значения остаются неизменными до и после цикла. Итак, цикл **parfor** требует, чтобы весь код, который следует за циклом не, зависел от последовательности итераций цикла. Результат выполнения:

Рассмотрим следующий пример - попытаемся посчитать последовательность Фибоначчи:

f = zeros(1,50);

f(1) = 1;

f(2) = 2;

parfor n = 3:50

f(n) = f(n­1) + f(n­2);

end

Этот цикл написан некорректно, поскольку значение f в каждой итерации зависит от результата выполнения других итераций.

**Функции для работы с параллельными задачами**

Xs = gcat(X) сцепляет variant массивы X из каждого lab по второму измерению (по столбцам). Результат распространяется во все lab.

Xs = gcat(X, номер измерения) сцепляет variant массивы X из каждого lab по указанному измерению.

## Пример с 4 потоками:

Xs = gcat(labindex)

Возвращает Xs = [1 2 3 4] во все четыре потока.

res = gop(@F, x) вычисляет функцию F по значению x из каждого потока. Результат дублируется во все потоки.

Функция F(x,y) должна принимать два аргумента одного типа и получать один результат того же типа, так что он может использоваться итеративно, то есть:

F(F(x1,x2),F(x3,x4))

Функция F должна быть ассоциативна, то есть:

F(F(x1, x2), x3) = F(x1, F(x2, x3))

res = gop(@F, x, targetlab) выполняет вычисления и помещает результат в переменную res в поток, указываемый параметром targetlab. Во всех остальных потоках переменная res устанавливается в значение[].

## Аргументы:

|  |  |
| --- | --- |
| F | Функция, которая действует во всех lab. |
| X | Аргумент для функции F, должен быть одной и той же переменной во всех lab, но может иметь различные значения. |
| res | Переменная для сохранения результата. |
| targetlab | Рабочий процесс, которому возвращается результат. |

## Примеры:

Вычисление суммы всех значений x во всех lab:

res = gop(@plus,x)

Нахождение максимального значения x по всем lab:

res = gop(@max,x)

Можно создать собственную функцию для запуска функцией **gop()**:

matlabpool(4)

spmd

x=labindex

res=gop(@pr,x)

end

res{1}

matlabpool close

Файл pr.m

function z = pr(x,y)

z=x.\*y;

end

**Измерение времени выполнения фрагмента кода**

tic; любые операторы; toc;

измеряет время выполнения операторов. Команда **tic** начинает отсчет времени, затем выполняются операторы и **toc** останавливает таймер, выводя на экран время выполнения в секундах.

tic; любые операторы; tElapsed=toc; делает то же самое, но запоминает время выполнения в переменной **tElapsed**.

tStart=tic; любые операторы; toc(tStart); делает то же самое, но позволяет запустить несколько счетчиков времени. Результат **tic** запоминается в переменной **tStart** и затем и используем ту же переменную при вызове **toc**. Измерение времени происходит от точки вызова **tic** и связанной с ней **toc** и выводит время в секундах.

tStart=tic; любые операторы; tElapsed=toc(tStart); делает то же самое, что и предыдущий вариант, но дополнительно запоминает время выполнения в переменной **tElapsed**.

Пример. Изменение минимального и среднего времени вычисления суммы функций Бесселя:

REPS = 1000; minTime = Inf; nsum = 10;

tic;

for i=1:REPS

tStart = tic; total = 0;

for j=1:nsum,

total = total + besselj(j,REPS);

end

tElapsed = toc(tStart);

minTime = min(tElapsed, minTime);

end

averageTime = toc/REPS;

**Реализация расчетов на GPU**

Графический ускоритель (GPU) является векторным сопроцессором, позволяющим с высокой эффективностью выполнять одинаковые действия над множеством однородных данных. Изначально видеоускорители применялись только для обработки графической информации, однако со времён появления возможности выполнения произвольного кода (т.н. шейдеров) возникла идея использовать их и для выполнения различных научных, физических и т.п. вычислений. В настоящее время производители видеокарт предлагают решения для применения графических ускорителей в целях выполнения вычислений общего назначения на GPU (GPGPU – general purpose GPU computations). В качестве таковых выступают технология CUDA от nVidia и OpenCL от ATi/AMD.

Вычисления GPU в сущности состоят в применении одного и того же программного кода (т.н. ядра задачи – kernel) к большому массиву данных. Над каждым элементом массива операция производятся параллельно по одному и тому же алгоритму. Далеко не каждая задача может быть оптимально реализована с использованием GPU (что связано со спецификой их архитектуры), однако на ряде задач производительность может превосходить центральный процессор на порядок (т.к. современные GPU содержат сотни процессорных ядер).

Matlab начиная с версии R2010b содержит некоторую функциональность для выполнения математических операций с задействованием GPU. Возможно как выполнение кода для CUDA, так и применением “родного” Matlab-кода с выполнением команд на GPU. При этом часть встроенных функций допускает их выполнение на GPU при вызове в отношении находящихся в памяти GPU данных.

Для работы основанной на технологии CUDA GPGPU функциональности Matlab требуется видеопроцессор производства nVidia, установленные актуальные версии драйверов и CUDA Toolkit. Проверить функционирование системы можно с помощью утилиты CUDA-Z и аналогичных, а также вызвав команду gpuDevice() в Matlab. При наличии нескольких видеокарт в системе необходимо настроить видеодрайвер так, чтобы для Matlab применялось видеоядро nVidia. Также можно запускать Matlab через пункт “Запустить с графическим процессором” в контекстном меню ярлыка Matlab.

Следует отметить, что возможности вычислений на GPU в Matlab ранних версий весьма ограничены. Для более гибкого использования таких вычислений рекомендуется применять Matlab версий новее, чем R2010b.

Получить список функций, для которых реализована поддержка вычислений на GPU в данной версии Matlab можно с помощью команды

methods('parallel.gpu.GPUArray').

В качестве примера рассмотрим следующую задачу.

Требуется реализовать программу, выполняющую суммирование всех элементов массива *ai*, таких, что *ai\*b≥c*. Реализацию выполнить с использованием GPU.

Поскольку GPU – векторный вычислитель, то наиболее эффективно его применение в векторизованных программах. Базовой функцией для таких вычислений в Matlab является *arrayfun(@F,a,b)*. Данная функция выполняет некоторую заданную функцию *F* в отношении каждого из элементов массивов a и b, возвращая результат выполнения. Важно понять, что функция выполняется поэлементно (т.н. операция map), т.е. из массивов *a* и *b* берутся очередные элементы, передаются как аргумент в функцию, результат помещается в массив, возвращаемый функцией. На *N* элементов входных массивов будет *N* элементов в результирующем. Необходимо учесть, что тело функции не должно содержать циклов, ветвлений, не должно изменять одну и ту же переменную из разных экземпляров и т.п.

Напишем функцию gpuFunction, оставляющую в результирующем массиве только те элементы, которые удовлетворяют условию *ai\*b≥c*. Остальные элементы будут занулены. Параметры *b* и *c* являются скалярами.

function res=gpuFunction(a,b,c)

res = a \* (a\*b >= c);

end

Примечание: для логических вычислений можно было бы заменить соответствующие элементы нулями и единицами.

Далее, выполним суммирование таких элементов с помощью встроенной функции sum. Данная функция поддерживает вычисления на GPU. Для того, чтобы вычисления происходили на GPU, передаваемый ей массив должен быть размещён в памяти GPU. Для этого он создаётся как gpuArray. Результаты выполнения операций над ним также будут представлять собой массивы, расположенные в памяти GPU. Для извлечения данных из памяти графического ускорителя используется команда gather.

В результате, текст программы имеет следующий вид:

tic

x = randi([100,100000],1000000,1,'single');

b = 10;

c = 20000;

gx = gpuArray(x);

R = arrayfun(@gpuFunction,gx,b,c);

s = sum(R);

r = gather(s);

toc

r

Примечание: см. также функции *min, max, prod* и т.п.