Supercomputer Software Department

Institute of Computational Mathematics and Mathematical Geophysics SB RAS

- ENGLISH
- РУССКИЙ

Программирование многопоточных приложений. POSIX Threads.

Цель работы

Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами.

Формулировка задачи

Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Как формируется задание, см. ниже. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как *все* задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров. Программа не должна зависеть от числа компьютеров.
Понятно, что для распараллеливания задачи задания из списка нужно распределять между компьютерами. Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие -- еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Как минимум, потоков должно быть 2:

- поток, который обрабатывает задания и, когда задания закончились, обращается к другим компьютерам за добавкой к работе,
- поток, ожидающий запросов о работе от других компьютеров

Полезно может быть завести третий поток, который возьмет на себя задачу подкачки работ на

компьютер, при этом первый поток будет только обрабатывать задания. В таком случае третий поток, до того как кончатся задания (соответствующий момент времени определить самостоятельно), на фоне счета будет отсылать запросы о работе и добавлять к локальному списку пришедшие задания.

Сложность задачи заключается в

- разработке правильной политики взаимодействия между процессами, когда все посылки (send) запросов и данных и ожидания (receive) приема запросов и данных будут согласованы.
- организации корректной работы многих потоков с общими структурами данных. Необходимо обеспечивать взаимное исключение потоков при добавлении заданий в список, удалении задач, выборке заданий для выполнения. Кроме того, надо помнить, что могут быть некторые неявно используемые обшие данные, в частности, сокрытые в реализации подключаемых библиотек и в том числе MPI, см. о требованиях MPI к многопоточным программам. Существует понятие "потокобезопасный" ("thread-safe"). Этот термин может относиться к библиотеке, процедуре и т.п. Он означает, что если потоки одной программы будут одновременно пользоваться функциями этой библиотеки или процедурой, то корректность поведения программы не нарушится. Очевидно, в реализации потокобезопасного кода должно быть предусмотрено возможное параллельное использование этого кода несколькими потоками.

Использование потоков позволяет производить перераспределение заданий на фоне счета. Благодаря этому можно добиться гораздо более эффективного использования ресурсов, чем если бы процесс должен был прерывать обработку заданий на время принятия или отсылки части работы.

Инструменты

Для организации взаимодействия между компьютерами нужно использовать MPI. См. о требованиях MPI к многопоточным программам. Для организации потоков использовать POSIX Threads. См. описание API и пример программы с потоками .

Замечание
Для посылки запросов и данных к конкретным адресатам можно использовать, как обычно, `MPI_Send` с указанием номера (`rank`) процесса-адресата. Для приема запроса от *любого* процесса можно использовать функцию `MPI_Recv`, где в качестве параметра "источник сообщения" указать константу `MPI_ANY_SOURCE`.

Формирование списка заданий

Задание в данном случае пусть будет иметь совершенно модельный характер. Например, оно может быть таким: выполнить некоторые тратящие время процессора действия `repeatNum` раз. Различие в вычислительном весе заданий будет заключаться в том, что у каждого задания количество повторений `repeatNum` свое:

```
TaskList tl;
double globalRes = 0;
int iterCounter = 0;
```

```
...
while(true) //итерации обработки списков
{
    iterCounter++; // счетчик глобальных итераций
    for(всех task = номер задания из списка) // выборка заданий из списка
         for(int i = 0; i<tl[task].repeatNum; i++)
                   globalRes += sqrt(i);

    ...
}
```

Вес задачи можно назначить случайным образом с использованием функции `rand()`, однако для экспериментов лучше задать некоторые осмысленные правила изменения загрузки на процессорах. Например, на каждой глобальной итерации `iterCounter` веса заданий на процессоре с номером `rank` установить пропорционально (или в некоторой другой зависимости от) `abs(rank-(iterCounter%size))`, где `size` -- количество процессоров. Это создаст "волну" загрузки, смещающуюся с ходом итераций от процессора с меньшим номером к процессору с большим номером (с перескакиванием на начало). Задания генерируются для каждой глобальной итерации заново!

Синхронизация процессоров между итерациями должна заключаться в том, что никто не начинает новую итерацию, пока не обработаны все задания с предыдущей.

Пример многопоточной программы. Параллельное умножение векторов.

В программе порождаются 4 потока, каждый из которых вычисляет часть скалярного произведения, обрабатывая свои четверти от векторов, а затем добавляет полученный результат к глобальному результату.

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

//переменная для сборки окончательного результата
int globalRes = 0;
//векторы, которые предстоит умножить
int v1[100], v2[100];
//
int ids[4] = {0,1,2,3};
//четрыре объекта типа "описатель потока"
pthread_t thrs[4];
//мьютекс
pthread_mutex_t mutex;

//функция потока
void* prod(void* me)
{
//узнали номер потока
int offset = *((int*)me);
```

```
//вычислили смещение в векторе до "своей" четверти
offset *= 25;
//вычисление частичного результата
int res = 0;
for(int i = offset; i<offset+25; i++)
    res += v1[i]*v2[i];
//захват мьютекса
pthread_mutex_lock(&mutex);
//добавление к глобальному результату при исключительном владении глобальной
переменной
globalRes += res;
//освобождение мьютекса
pthread_mutex_unlock(&mutex);
}

int main()
{
     //тут где-то должна быть инициализация массивов

     //атрибуты потока
     pthread_attr_t attrs;

     //инициализация атрибутов потока
     if(0!=pthread_attr_init(&attrs))
     {
     perror("Cannot initialize attributes");
     abort();
     };
     //установка атрибута "присоединенный"
     if(0!=pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_JOINABLE))
     {
     perror("Error in setting attributes");
     abort();
     }
     //порождение четырех потоков
     for(int i = 0; i<4; i++)
     if(0!=pthread_create(&thrs[i], &attrs, prod, &ids[i]))
     {
     perror("Cannot create a thread");
     abort();
     }
     //освобождение ресурсов, занимаемых описателем атрибутов
     pthread_attr_destroy(&attrs);
     //ожидание завершения порожденных потоков
     for(int i = 0; i<4; i++)
     if(0!=pthread_join(thrs[i], NULL)
     {
     perror("Cannot join a thread");
     abort();
```

```
    }

    return 0;
}
```

Описание POSIX Threads API

POSIX Threads API состоит из большого набора функции, и лишь необходимые для выполнения задания описаны здесь. Для более полного освоения материала см. **литературу (http://ssdonline.sscc.ru/korneev/pthreads.html#biblio)** .

Замечание
В описании параметров слово "IN" означает, что параметр используется для передачи значения в функцию, а "OUT" -- что через параметр функция возвращает результаты. Часто, параметрами функций будут указатели. В таком случае, "IN" говорит о том, что в области памяти, на которую указывает параметр, размещены потребляемые функцией данные, а "OUT" говорит о том, что в этой области будут размещены данные в результате работы функции. Нужно заметить, что в обоих случаях указатель инициализируется до вызова функции, т.е. функция не отвечает за размещение данных в памяти -- она либо читает оттуда, откуда указано, либо пишет туда, куда указано. Если в качестве фактического параметра типа "IN" передается NULL, то это означает, что функция должна поступать в соответствии с политикой по умолчанию. Если NULL указан вместо параметра типа "OUT", то это означает, что соответствующий результат программисту не нужен.

Порождение потока

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine)(void*), void *arg);
```

`thread`	OUT	В результате успешного срабатывания функции по указанному адресу будет размещен описатель порожденного потока.
`attr`	IN	Атрибуты потока. Задают свойства потока. Может быть NULL. Описание атрибутов см. ниже.
`start_routine`	IN	Указатель на функцию потока. Выполнение потока состоит в выполнении этой функции.
`arg`	IN/OUT	Указатель, который будет передан функции потока в качестве параметра. OUT -- в том смысле, что функция потока может менять содержимое памяти с использованием этого указателя. `pthread_create` сожержимого не меняет, просто передает указатель функции потока. Функция потока сама интерпретирует содержание памяти по этому адресу.

Атрибуты

Объект задающий атрибуты потока имеет тип `pthread_attr_t`. Такой объект должен быть инициализирован с помощью функции

```
int pthread_attr_init(pthread_attr_t *attr);
```

В результате объект будет содержать набор свойств потока по умолчанию для данной реализации потоков. А ресурсы, которые могут использоваться в системе для хранения этих атрибутов освобождаются вызовом функции (после того, как объект был использован в вызове pthread_create и больше не нужен)

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Поток может быть "присоединяемым" (joinable) или "оторванным" (detached). Для установки этого свойства в атрибутах используется функция

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

где `detachstate` можно установить в `PTHREAD_CREATE_JOINABLE` или в `PTHREAD_CREATE_DETACHED` соответственно.

Присоединяемые и оторванные потоки

Для каждого присоединяемого потока, один из других потоков явно должнен вызвать функцию

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Поток, вызвавший эту функцию, останавливается, пока не окончится выполнение потока `thread`. Если никто не вызывает `pthread_join` для присоединяемого потока, то завершившись, он не освобождает свои ресурсы, а это может служить причиной утечки памяти в программе. `value_ptr` (OUT) -- это указатель на указатель, возвращенный функцией завершившегося потока.

Взаимное исключение потоков

Для организации взаимного исключения потоков при доступе к разделяемым данным используются мьютексы (mutex = mutual exclusion), объекты типа `pthread_mutex_t`. Мьютекс должен быть инициализирован перед использованием с помощью функкции

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*attr);
```

Параметр `attr` (IN) задает атрибуты мьютекса. Можно передать NULL для принятия атрибутов по умолчанию. Ресурсы, занимаемые мьютексом, могут быть освобождены функцией

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Для захвата мьютекса поток использует (см. пример) функцию

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

а для освобождения

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Условные переменные

Поясним использование условных переменных на примере. Например, поток номер 1 должен выполнить некоторые действия, когда значение некоторого глобального счетчика `counter` достигнет критического значения `criticalVal`, причем значение счетчика меняет поток номер 2. Тогда, чтобы первый поток не крутился в цикле, все время проверяя значение `counter`, можно использовать условную переменную, с помощью которой поток номер 1 устанавливается в состояние ожидания до тех пор, пока поток номер 2 не просигналирует о наступлении нужного события:

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

pthread_mutex_t mutex; //условная переменная pthread_cond_t cond; //счетчик int counter = 0;
ПОТОК НОМЕР 1

//захват мьютекса pthread_mutex_lock(&mutex); //если значение не равно критическому if(counter!=criticalValue) //останавливаемся в ожидании этого события. //При этом мьютекс будет разблокирован. Как только событие наступит, //и мьютекс будет отпущен вторым потоком, //мьютекс снова захватывается и выполнение потока 1 продолжается pthread_cond_wait(&cond, &mutex); //обработка критического значения processCriticalValue(); // освобождение мьютекса pthread_mutex_unlock();
ПОТОК НОМЕР 2

do { ... //захват мьютекса pthread_mutex_lock(&mutex); //изменение значения счетчика doSomethingWith(&counter); //если событие наступило if(counter==criticalValue) // просигналировать об этом ждущим на условной переменной cond pthread_cond_signal(&cond); //отпустить мьютекс pthread_mutex_unlock(&mutex); ... }while(...);
Условная переменная должна быть инициализирована функцией

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

А ресурсы, занятые ей, могут быть освобождены функцией

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

MPI и потоки

Эта тема раскрывается в главе **8.7. MPI and Threads (http://www.mpi-forum.org/docs/mpi-20-html/node162.htm#Node162)** стандарта MPI-2.0. В ней указываются минимальные требования к реализации MPI, поддерживающей программы с многими потоками, и говорится о том, как должна инициализироваться (в смысле того, для чего существует MPI_Init) такая реализация. Здесь заметим следующее:

- Каждый поток в процессе может посылать и принимать сообщения. Адресуемой единицей является процесс, поэтому любой поток может принять любое сообщение адресованное его процессу.
- Для того, чтобы сообщения не путались между потоками, каждый поток, например, может использовать свой MPI-коммуникатор для обмена данными с другими процессами.

- Стандарт говорит, что из утверждения "реализация MPI поддерживает многопоточность" (она "thread-compliant") должно следовать, что все ее процедуры (функции) потокобезопасны, и любая блокирующая операция MPI блокирует только вызвавший эту операцию поток.
- Функция `MPI_Finalize()` должна вызываться только тем потоком, который инициализировал MPI, и только после того, как все потоки завершили выполнение MPI-операций. Поэтому все потоки должны быть присоединяемыми (joinable) и присоединены к этому главному потоку перед вызовом `MPI_Finalize()`.
- Вместо `MPI_Init` в многопоточной программе может быть использована `int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int *provided)`

`argc`	IN	То же самое, что и в `MPI_Init`
`argv`	IN	То же самое, что и в `MPI_Init`
`required`	IN	Запрашиваемый уровень "поддержки потоков": ◦ `MPI_THREAD_SINGLE` -- программист обещает, что программа будет иметь лишь один поток. ◦ `MPI_THREAD_FUNNELED` -- программист обещает, что только тот поток, который инициализировал MPI будет в дальнейшем вызывать функции MPI. ◦ `MPI_THREAD_SERIALIZED` -- программист обещает, что потоки не будут вызывать функции MPI одновременно. ◦ `MPI_THREAD_MULTIPLE` -- программист ничего не обещает. Любой поток может вызывать MPI-функции независимо от других потоков. Если реализация MPI поддерживает этот уровень, то это и есть thread-compliant реализация MPI.
`provided`	OUT	Фактически предоставленный уровень

`MPI_Init` означает `MPI_Init_thread` с уровнем `MPI_THREAD_SINGLE`.

Литература

1. International Organization for Standardization, Geneva. Information technology --- Portable Operating System Interface (POSIX) --- Part 1: System Application Program Interface (API) [C Language], December 1996.
2. Потоки в UNIX. **http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html (http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html)**
3. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition. Follow the links. **http://www.opengroup.org/onlinepubs/009695399/mindex.html (http://www.opengroup.org/onlinepubs/009695399/mindex.html)**
4. MPI Documents. **http://www.mpi-forum.org/docs/ (http://www.mpi-forum.org/docs/)**
5. Manual pages. В командной строке UNIX набрать: man <имя фунции>.