**21. Критичні секції, замки (lock).**

**Критическая секция** — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения. При нахождении в критической секции двух (или более) процессов возникает состояние «гонки» («состязания»). Для избежания данной ситуации необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

**Критическая секция** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *critical section*) — объект синхронизации потоков, позволяющий предотвратить одновременное выполнение некоторого набора операций (обычно связанных с доступом к данным) несколькими потоками. Критическая секция выполняет те же задачи, что и [мьютекс](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81).

Между мьютексом и критической секцией есть терминологические различия, так процедура, аналогичная захвату мьютекса, называется входом в критическую секцию ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *enter*), снятию блокировки мьютекса — выходом из критической секции ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *leave*).

Процедура входа и выхода из критических секций обычно занимает меньшее время, нежели аналогичные операции мьютекса, что связано с отсутствием необходимости обращаться к ядру ОС.

**Переменная-замок**

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в *критическую секцию* только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из *критической секции* процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " *Критическая секция* ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {

while(lock); lock = 1;

critical section

lock = 0;

remainder section

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию *взаимоисключения*, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессуP1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в *критический участок*. Мы получаем два процесса, одновременно выполняющих свои *критические секции*.

std:: lock, функция которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку.

**22. Семафори.**

**Семафо́р** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *semaphore*) — объект, ограничивающий количество [потоков](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F), которые могут войти в заданный участок кода. Определение введено [Эдсгером Дейкстрой](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5). Семафоры используются для синхронизации и защиты передачи данных через [разделяемую память](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), а также для синхронизации работы процессов и потоков.

### Типы семафоров

В зависимости от значений, которые может приниматъ семафор он делится на:

* Двоичный : принимает значения 0 и 1.
* Троичный : принимает значения 0,1 и 2 и тд

Некоторые из проблем, которые могут решать семафоры:

* запрет одновременного выполнения заданных участков кода ([критические секции](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F));
* поочерёдный доступ к критическому ресурсу (важному [ресурсу](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B), для которого невозможен (или нежелателен) одновременный доступ);
* синхронизация процессов и потоков (например, можно инициировать обработку события отпусканием семафора).

## Проблемы семафоров

Во-первых, можно написать программу с «утечкой семафора», вызвав enter() и забыв вызвать leave(). Реже встречаются ошибки, когда дважды вызывается leave().

Во-вторых, семафоры чреваты [взаимной блокировкой](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) потоков.

Предложены Дейкстрой еще в середине 1960-х годов.

Под семафором S понимается переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций P(S) и V(S), реализуемых в соответствии со следующими алгоритмами:

операция P(S)

если S > 0

то S = S – 1

иначе < ожидать S >

операция V(S)

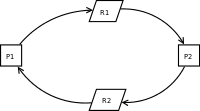
если < один или несколько потоков ожидают S >

то < снять ожидание у одного из ожидающих потоков >

иначе S = S + 1

Принципиальным в понимании семафоров является то, что операции P(S) и V(S) предполагаются неделимыми (атомарными), что гарантирует взаимоисключение при использовании общих семафоров.

Семафоры широко используются для синхронизации и взаимоисключения потоков. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

[](https://commons.wikimedia.org/wiki/File:Process_deadlock.svg?uselang=ru) **23. Дедлоки, умови виникнення та алгоритми запобігання**.

**Взаи́мная блокиро́вка** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *deadlock*) — ситуация в многозадачной среде или [СУБД](https://ru.wikipedia.org/wiki/%D0%A1%D0%A3%D0%91%D0%94), при которой несколько [процессов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) находятся в состоянии бесконечного ожидания [ресурсов](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B), занятых самими этими процессами.

Взаимная блокировка двух процессов P1 и P2 нуждающихся в двух ресурсах.

## Простейший пример взаимной блокировки

|  |  |  |
| --- | --- | --- |
| **Шаг** | **Процесс 1** | **Процесс 2** |
| 0 | Хочет захватить A и B, начинает с A | Хочет захватить A и B, начинает с B |
| 1 | Захватывает ресурс A | Захватывает ресурс B |
| 2 | Ожидает освобождения ресурса B | Ожидает освобождения ресурса A |
| 3 | Взаимная блокировка | |

## Livelock

Это слово означает такую ситуацию: система не «застревает» (как в обычной взаимной блокировке), а занимается бесполезной работой, её состояние постоянно меняется — но, тем не менее, она «[зациклилась](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%86%D0%B8%D0%BA%D0%BB%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)», не производит никакой полезной работы.

Жизненный пример такой ситуации: двое встречаются лицом к лицу. Каждый из них пытается посторониться, но они не расходятся, а несколько секунд сдвигаются в одну и ту же сторону.

**Алгоритмы и методы предотвращения взаимной блокировки**

***1)Алгоритм Банкира***— это алгоритм [распределения ресурсов](https://en.wikipedia.org/wiki/Resource_allocation) (англ.) и обхода взаимоблокировок,

 Він досліджує можливий розвиток подій шляхом відтворення розподілу заздалегідь визначеної кількості ресурсів, і тоді робить перевірку на безпечність стану з метою дослідження на можливі умови взаємних блокувань для всіх інших очікуючих активностей, перед прийняттям рішення чи можна дозволити подальший розподіл.

## Алгоритм

Алгоритм банкіра виконується операційною системою щоразу, коли [процес](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) запитує ресурси.[[2]](https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%B1%D0%B0%D0%BD%D0%BA%D1%96%D1%80%D0%B0#cite_note-2) Алгоритм запобігає взаємним блокуванням шляхом відмови або відкладання запитів, якщо він визначає, що виконання цих запитів переведе систему до небезпечного стану (у якому можливе взаємне блокування). Коли в системі з'являється новий процес, він має заявити максимально потрібну кількість ресурсів кожного типу, але не більше, ніж загальна кількість ресурсів у системі. Також, коли процес отримує в своє розпорядження ресурси, він має повернути їх системі за скінченний проміжок часу.

***2)Предотвращение рекурсивных блокировок***. Это предотвращает поток от входа в одну и ту же блокировку несколько раз.

## Предотвращение взаимной блокировки

Классический способ борьбы с проблемой — разработка иерархии блокировок, установление правила, что некоторые блокировки никогда не могут захватываться в состоянии, в котором уже захвачены какие-то другие блокировки. Говоря точно, речь о разработке отношения сравнения между блокировками, и о запрете захвата «большей» блокировки в состоянии, когда уже захвачена «меньшая».

**24. Особливості паралельного середовища OpenMP.**

**25. Базові поняття OpenMP: паралельні блоки, патерн SPMD, паралельні цикли, редукція.**

В большинстве MPI-программ используется шаблон "Одна программа, разные данные" (Single Program Multiple Data, или SPMD) [mattson05]. В его основе лежит простой принцип: каждый элемент обработки (processing element, PE) выполняет одну и ту же программу. Каждому элементу обработки присваивается уникальный целочисленный идентификатор, который определяет его ранг в наборе элементов обработки. Программа использует этот ранг, чтобы распределить работу и определить, какой элемент PE какую работу выполняет. Иными словами, программа всего одна, но благодаря выбору, сделанному в соответствии с идентификатором, данные для каждого элемента обработки могут быть разными. Это и есть шаблон "Одна программа, разные данные".

Значительная часть параллельного программирования состоит именно в том, чтобы поручить всем потокам выполнение одних и тех же инструкций. Но чтобы использовать OpenMP в полной мере, потребуется что-то большее. Необходимо разделить между потоками работу по выполнению набора инструкций. Такой тип поведения называется "совместное выполнение работы". Самая типичная конструкция для совместной работы - это конструкция цикла (в C это цикл for).

|  |  |
| --- | --- |
| 1 | #pragma omp for |

Это работает только для простых циклов стандартного вида:

|  |  |
| --- | --- |
| 1 | **for**(i=lower\_limit; i<upper\_limit; inc\_exp) |

Конструкция for распределяет итерации цикла между потоками группы, созданными ранее с помощью конструкции parallel. Начальное и конечное значения счетчика цикла, а также выражение для шага счетчика (inc\_exp) должны быть полностью определены во время компиляции, а все константы, которые используются в этих выражениях, должны быть одинаковы для всех потоков группы. Если вдуматься, это не лишено смысла. Система должна вычислить, сколько итераций цикла должно быть выполнено, чтобы разделить их на наборы, которые будут обрабатывать группы потоков. Это можно сделать только согласованно и точно, если все потоки используют одни и те же наборы счетчиков.

Необходимо отметить, что сама по себе конструкция for потоки не создает. Это можно сделать только с помощью конструкции parallel. Для простоты можно поместить конструкции parallel и for в одну и ту же прагму.

|  |  |
| --- | --- |
| 1 | #pragma omp parallel for |

При этом будет создана группа потоков для выполнения итераций цикла, который следует непосредственно за прагмой.

Итерации цикла должны быть независимыми, чтобы результат выполнения цикла оставался неизменным независимо от того, в каком порядке и какими потоками выполняются эти итерации. Если один поток записывает переменную, которую затем считывает другой поток, то наблюдается кольцевая зависимость, и результат работы программы будет неверным. Программист должен тщательно проанализировать тело цикла, чтобы убедиться в отсутствии кольцевых зависимостей. В большинстве случаев такая зависимость возникает, если в переменную записываются промежуточные результаты, которые используются в данной итерации цикла.

Кроме того, часто встречается ситуация, когда переменная внутри цикла используется для сложения значений, полученных в каждой итерации. Например, это происходит в цикле, который суммирует результаты вычислений, чтобы получить одно итоговое значение. Такая ситуация часто возникает в параллельном программировании. Она называется "редукция". В OpenMP имеется оператор reduction:

|  |  |
| --- | --- |
| 1 | reduction(+:sum) |

Как и оператор private, он добавляется в конструкцию OpenMP, чтобы сообщить компилятору, что следует ожидать редукции. После этого создается временная закрытая переменная, которая используется для получения промежуточного результата операции суммирования для каждого потока. В конце выполнения конструкции значения этой переменной для каждого потока суммируются, чтобы получить конечный результат. Операция, которая используется при редукции, также указывается в операторе. В данном случае это операция "+". OpenMP определяет значение для закрытой переменной, которая используется в редукции, на основе соответствующей математической операции. Например, для "+" это значение равно нулю.

**26. Базові поняття OpenMP: секції, бар’єри, приватні та спільні змінні, статичне та динамічне розбиття циклів.**

Программа состоит из последовательных и параллельных секций.

•В начальный момент времени создается главная нить, выполняющая последовательные секции программы.

•При входе в параллельную секцию выполняется операция *fork*, порождающая семейство нитей. Каждая нить имеет свой уникальный числовой идентификатор (главной нити соответствует 0). При распараллеливании циклов все параллельные нити исполняют один код. В общем случае нити могут исполнять различные фрагменты кода.

•При выходе из параллельной секции выполняется операция *join*. Завершается выполнение всех нитей, кроме главной.

Директивы OpenMP

***#pragma omp*** задает границы параллельной секции программы. С данной директивой могут

использоваться следующие операторы:

• private;

• shared;

• default;

• firstprivate;

• reduction;

• if;

• copyin;

• num\_threads.

## Директивы pragma для синхронизации

При одновременном выполнении нескольких потоков часто возникает необходимость их синхронизации. OpenMP поддерживает несколько типов синхронизации, помогающих во многих ситуациях.

Один из типов — неявная барьерная синхронизация, которая выполняется в конце каждого параллельного региона для всех сопоставленных с ним потоков. Механизм барьерной синхронизации таков, что, пока все потоки не достигнут конца параллельного региона, ни один поток не сможет перейти его границу.

Неявная барьерная синхронизация выполняется также в конце каждого блока #pragma omp for, #pragma omp single и #pragma omp sections. Чтобы отключить неявную барьерную синхронизацию в каком-либо из этих трех блоков разделения работы, укажите раздел nowait:

#pragma omp parallel

{

#pragma omp for nowait

for(int i = 1; i < size; ++i)

x[i] = (y[i-1] + y[i+1])/2;

}

Как видите, этот раздел директивы распараллеливания говорит о том, что синхронизировать потоки в конце цикла for не надо, хотя в конце параллельного региона они все же будут синхронизированы.

Второй тип — явная барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Для этого включите в код директиву #pragma omp barrier.

В качестве барьеров можно использовать критические секции. В Win32 API для входа в критическую секцию и выхода из нее служат функции EnterCriticalSection и LeaveCriticalSection. В OpenMP для этого применяется директива #pragma omp critical [имя]. Она имеет такую же семантику, что и критическая секция Win32, и опирается на EnterCriticalSection. Вы можете использовать именованную критическую секцию, и тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем неименованным критическим секциям является взаимоисключающим.

***#pragma omp for***

Задает границы цикла, исполняемого в параллельном режиме.

***#pragma omp parallel*** {

for(int i = 1; i < size; ++i)

x[i] = (y[i-1] + y[i+1])/2;

} // каждый поток выполнит полный цикл for, // проделав много лишней работы

//сокращенный способ записи

***#pragma omp parallel for***

for(int i = 1; i < size; ++i)

x[i] = (y[i-1] + y[i+1])/2;

**Общие и частные данные**

•Разрабатывая параллельные программы, вы должны понимать, какие данные являются общими (shared), а какие частными (private), — от этого зависит не только производительность, но и корректная работа программы. В OpenMP это различие очевидно, к тому же вы можете настроить его вручную.

•Общие переменные доступны всем потокам из группы, поэтому изменения таких переменных в одном потоке видимы другим потокам в параллельном регионе. Что касается частных переменных, то каждый поток из группы располагает их отдельными экземплярами, поэтому изменения таких переменных в одном потоке никак не сказываются на их экземплярах, принадлежащих другим потокам.

•По умолчанию все переменные в параллельном регионе — общие, но из этого правила есть три исключения.

•Во-первых, частными являются индексы параллельных циклов for.

•Во-вторых, частными являются локальные переменные блоков параллельных регионов.

•В-третьих, частными будут любые переменные, указанные в разделах private, firstprivate, lastprivate и reduction.

Раздел **private** говорит о том, что для каждого потока должна быть создана частная копия каждой переменной из списка. Частные копии будут инициализироваться значением по умолчанию (с применением конструктора по умолчанию, если это уместно). Например, переменные типа int имеют по умолчанию значение 0.

•У раздела **firstprivate** такая же семантика, но перед выполнением параллельного региона он указывает копировать значение общей переменной в каждый поток, используя конструктор копирования, если это уместно.

•Семантика раздела **lastprivate** тоже совпадает с семантикой раздела private, но при выполнении последней итерации цикла или раздела конструкции распараллеливания значения переменных, указанных в разделе lastprivate, присваиваются переменным основного потока. Если это уместно, для копирования объектов применяется оператор присваивания копий (copy assignment operator).

раздел reduction, но он принимает переменную и оператор. Поддерживаемые этим разделом операторы перечислены в таблице , а у переменной должен быть скалярный тип (например, float, int или long, но не std::vector, int [] и т. д.).

•Переменная раздела reduction инициализируется в каждом потоке значением, указанным в таблице. В конце блока кода оператор раздела reduction применяется к каждой частной копии переменной, а также к исходному значению переменной.

**nowait** - Отменяет барьерную синхронизацию при завершении выполнения параллельной секции.

**schedule -** По умолчанию в OpenMP для планирования параллельного выполнения циклов for применяется алгоритм, называемый статическим планированием (static scheduling). Это означает, что все потоки из группы выполняют одинаковое число итераций цикла. Если n — число итераций цикла, а T — число потоков в группе, каждый поток выполнит n/T итераций (если n не делится на T без остатка, ничего страшного). Однако OpenMP поддерживает и другие механизмы планирования, оптимальные в разных ситуациях: динамическое планирование (dynamic scheduling), планирование в период выполнения (runtime scheduling) и управляемое планирование (guided scheduling).

**ordered** - обеспечивает сохранение того порядка выполнения итераций цикла, который соответствует последовательному выполнению программы.

Параллельная обработка в конструкциях, отличных от циклов

•OpenMP поддерживает параллелизм и на уровне функций. Этот механизм называется секциями OpenMP (OpenMP sections).

Синтаксис:

#pragma omp parallel sections

{

#pragma omp section

//Вызов первой функции

#pragma omp section

//Вызов второй функции

}

Барьерная синхронизация. В некоторых ситуациях ее целесообразно выполнять наряду с неявной. Используется **#pragma omp barrier.**

|  |  |  |
| --- | --- | --- |
| **Простая блокировка OpenMP** | **Вложенная блокировка OpenMP** | **Win32-функция** |
| omp\_lock\_t | omp\_nest\_lock\_t | CRITICAL\_SECTION |
| omp\_init\_lock | omp\_init\_nest\_lock | InitializeCriticalSection |
| omp\_destroy\_lock | omp\_destroy\_nest\_lock | DeleteCriticalSection |
| omp\_set\_lock | omp\_set\_nest\_lock | EnterCriticalSection |
| omp\_unset\_lock | omp\_unset\_nest\_lock | LeaveCriticalSection |
| omp\_test\_lock | omp\_test\_nest\_lock | TryEnterCriticalSection |

Для синхронизации кода можно использовать и подпрограммы исполняющей среды, и директивы синхронизации. Преимущество директив в том, что они прекрасно структурированы. Это делает их более понятными и облегчает поиск мест входа в синхронизированные регионы и выхода из них.

•Критическая секция - директива **#pragma omp critical [имя].** Если используется именованная критическая секцию, тогда доступ к блоку кода является взаимоисключающим только для других критических секций с тем же именем (это справедливо для всего процесса). Если имя не указано, директива ставится в соответствие некоему имени, выбираемому системой. Доступ ко всем неименованным критическим секциям является взаимоисключающим.

В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, — например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива **#pragma omp single.**

•Иногда возможностей директивы single недостаточно. В ряде случаев требуется, чтобы блок кода был выполнен основным потоком, — например, если этот поток отвечает за обработку GUI и вам нужно, чтобы какую-то задачу выполнил именно он. Тогда применяется директива **#pragma omp master.** В отличие от директивы single при входе в блок master и выходе из него нет никакого неявного барьера.

•Чтобы завершить все незавершенные операции над памятью перед началом следующей операции, используйте директиву **#pragma omp flush**

Поддержка динамического создания потоков определяется значением булевого свойства, которое по умолчанию равно false. Если при входе потока в параллельный регион это свойство имеет значение false, исполняющая среда OpenMP создает группу, число потоков в которой равно значению, возвращаемому функцией omp\_get\_max\_threads. По умолчанию omp\_get\_max\_threads возвращает число потоков, поддерживаемых аппаратно, или значение переменной OMP\_NUM\_THREADS. Если поддержка динамического создания потоков включена, исполняющая среда OpenMP создаст группу, которая может содержать переменное число потоков, не превышающее значение, которое возвращается функцией omp\_get\_max\_threads.

<https://msdn.microsoft.com/ru-ru/library/dd335940.aspx>

* 1. **Базові поняття MPI: процес, ранг, комунікатор, особливості написання MPI програми.**

**Процесс** - совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

**Коммуникатор** представляет собой структуру, содержащую либо все процессы,

исполняющиеся в рамках данного приложения, либо их подмножество.

Процессы, принадлежащие одному и тому же коммуникатору, наделяются

общим контекстом обмена. Операции обмена возможны только между

процессами, связанными с общим контекстом, то есть, принадлежащие одному

и тому же коммуникатору (рис. 13). Каждому коммуникатору присваивается

идентификатор. В MPI есть несколько стандартных коммуникаторов:

* MPI\_COMM\_WORLD–включаетвсепроцессыпараллельнойпрограммы;
* MPI\_COMM\_SELF–включаеттолькоданныйпроцесс;
* MPI\_COMM\_NULL– пустой коммуникатор, не содержит ни одного

процесса.

В MPI имеются процедуры, позволяющие создавать новые коммуникаторы,

содержащие подмножества процессов.

**Ранг** процесса представляет собой уникальный числовой идентификатор, назначаемый процессу в том или ином коммуникаторе. Ранги в разных коммуникаторах назначаются независимо и имеют целое значение от 0 до число процессов – 1.

* 1. **Базові поняття MPI: блокуючий та неблокуючий обмін повідомленнями.**

Сообщение содержит пересылаемые данные и служебную информацию. Для того, чтобы передать сообщение, необходимо указать:

* ранг процесса-отправителя сообщения;
* адрес, по которому размещаются пересылаемые данные процесса-отправителя;
* тип пересылаемых данных;
* количество данных;
* ранг процесса, который должен получить сообщение;
* адрес, по которому должны быть размещены данные процессом-получателем.
* тег сообщения;
* идентификатор коммуникатора, описывающего область взаимодействия, внутри которой происходит обмен.

Тег это задаваемое пользователем целое число от 0 до 32767, которое играет роль идентификатора сообщения и позволяет различать сообщения, приходящие от одного процесса. Теги могут использоваться и для соблюдения определенного порядка приема

сообщений.

Прием сообщения начинается с подготовки буфера достаточного размера. В этот буфер записываются принимаемые данные. Операция отправки или приема сообщения считается завершенной, если программа может вновь использовать буферы сообщений.

Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.

При выполнении глобальных операций используются коллективные обмены. Асинхронные коммуникации реализуются с помощью запросов о получении сообщений. Имеется несколько разновидностей двухточечного обмена.

* Блокирующие прием*/*передача приостанавливают выполнение процесса на время приема сообщения.
* Неблокирующие прием*/*передача выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения.
  1. **Базові поняття MPI: редукція, групові операції.**

В рассматриваемой программе после входной стадии каждый процессор выполняет по существу те же самые команды до заключительной стадии суммирования. Поэтому, если функция f(x) дополнительно не усложнена (т. е. не требует значительной работы для оценки интеграла по некоторым частям отрезка [a;b], то эта часть программы распределяет среди процессоров одинаковую нагрузку. В заключительной стадии суммирования процесс 0 еще раз получает непропорциональное количество работы.

Возможны также варианты организации передач, как и в случае рассылки входных данных. Поэтому следует использовать другие механизмы, более оптимизированные для этой цели.

"Общая сумма", которую нужно вычислить представляет собой пример общего класса коллективных операций коммуникации, называемых операциями редукции. В глобальной операции редукции, все процессы (в коммуникаторе) передают данные, которые объединяются с использованием бинарных операций. Типичные бинарные операции - суммирование, максимум и минимум, логические и т.д. MPI содержит специальную функцию для выполнения операции редукции:

Операция приведения, результат которой передается одному процессу:

int MPI\_Reduce(void \*buf, void \*result, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm)

Входные параметры:

* buf - адрес буфера передачи;

• count - количество элементов в буфере передачи;

• datatype - тип данных в буфере передачи;

• op - операция приведения;

• root - ранг главного процесса;

• comm - коммуникатор.

MPI\_Reduce применяет операцию приведения к операндам из buf, а результат каждой операции помещается в буфер результата result. MPI\_Reduce должна вызываться всеми процессами в коммуникаторе comm, а аргументы count, datatype и op в этих вызовах должны совпадать.

|  |  |
| --- | --- |
| Название операции | Смысл |
| MPI\_MAX | Максимум |
| MPI\_MIN | Минимум |
| MPI\_SUM | Сумма |
| MPI\_PROD | Произведение |
| MPI\_LAND | Логическое И |
| MPI\_BAND | Битовое И |
| MPI\_LOR | Логическое ИЛИ |
| MPI\_BOR | Битовое ИЛИ |
| MPI\_LXOR | Логическое исключающее ИЛИ |
| MPI\_BXOR | Битовое исключающее ИЛИ |
| MPI\_MAXLOC | Максимум и его расположение |
| MPI\_MINLOC | Минимум и его расположение |

Существует также возможность определения собственных операций.

Таким образом, завершение программы вычисления интеграла будет следующим:

/\* Суммирование результатов от каждого процесса \*/

MPI\_Reduce(&integral, &total, 1, MPI\_FLOAT,

    MPI\_SUM, 0, MPI\_COMM\_WORLD);

/\* Вывод результата \*/

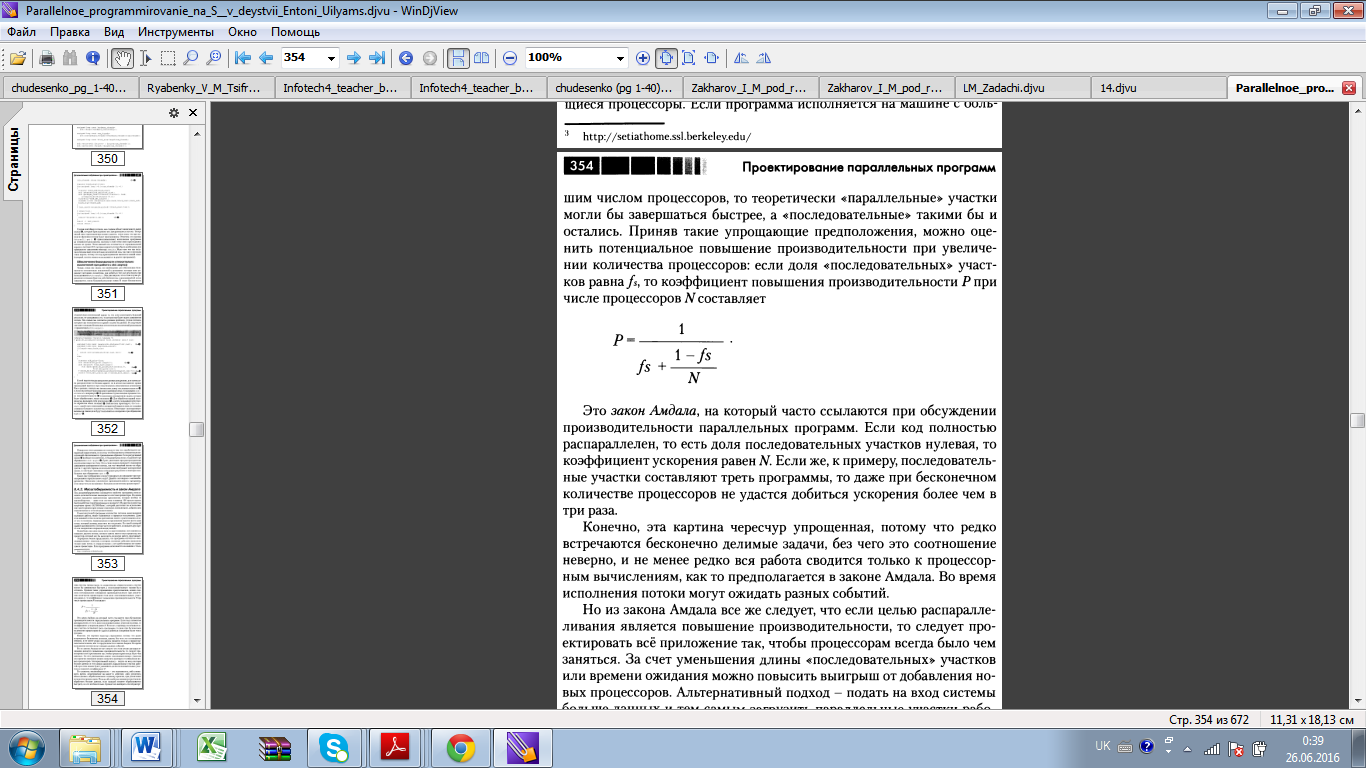
Следует отметить, что каждый процессор вызывает MPI\_Reduce() с одинаковыми аргументами. Например, если total имеет значение только для процесса 0, каждый процесс тем не менее использует этот аргумент.

Глобальные операции редукции

* Сумма
* Произведение
* Минимум
* Максимум
* Битовые операции

«Собираемая» переменная может быть простой или массивом.

30. Закон Амдела.



В многопоточной программе количество потоков, выполняющих полезную работу, может изменяться в процессе исполнения. Даже если каждый поток на всем протяжении своего существования делает что-то полезное, первоначально в приложении имеется всего один поток, который должен запустить все остальные. Но такой сценарий крайне маловероятен. Потоки часто не работают, а ожидают друг друга или завершения операций ввода/вывода.

Всякий раз, как один поток чего-то ждет (неважно, чего именно), а никакого другого потока, готового занять вместо него процессор, нет, процессор, который мог бы выполнять полезную работу, простаивает.

Упрощенно можно представлять, что программа состоит из «последовательных» участков, в которых полезные действия выполняет только один поток, и «параллельных», где задействованы все имеющиеся процессоры. Если программа исполняется на машине с большим числом процессоров, то теоретически «параллельные» участки могли бы завершаться быстрее, а «последовательные» такими бы и остались. Приняв такие упрощающие предположения, можно оценить потенциальное повышение производительности при увеличении количества процессоров: если доля «последовательных» участков равна /х, то коэффициент повышения производительности Р при числе процессоров N составляет

**!!!Тут должна бытъ формула которая сверху**

Это закон Амдала, на который часто ссылаются при обсуждении производительности параллельных программ. Если код полностью распараллелен, то есть доля последовательных участков нулевая, то коэффициент ускорения равен N. Если же, к примеру, последовательные участки составляют треть программы, то даже при бесконечном количестве процессоров не удастся добиться ускорения более чем в три раза.

Конечно, эта картина чересчур упрощенная, потому что редко встречаются бесконечно делимые задачи, без чего это соотношение неверно, и не менее редко вся работа сводится только к процессорным вычислениям, как то предполагается в законе Амдала. Во время исполнения потоки могут ожидать разных событий.

Но из закона Амдала все же следует, что если целью распараллеливания является повышение производительности, то следует проектировать всё приложение так, чтобы процессорам всегда было чем заняться. За счет уменьшения длины «последовательных» участков или времени ожидания можно повысить выигрыш от добавления новых процессоров. Альтернативный подход - подать на вход системы больше данных и тем самым загрузить параллельные участки работой; при этом можно будет уменьшить долю последовательных участков и повысить коэффициент Р.

По существу, масштабируемость - это возможность либо уменьшить время, затрачиваемое на какое-то действие, либо увеличить объем данных, обрабатываемых в единицу времени, при увеличении количества процессоров. Иногда оба свойства эквивалентны (можно обработать больше данных, если каждый элемент обрабатывается быстрее), но это необязательно. Прежде чем выбирать способ распределения работы между потоками, важно определить, какие аспекты масштабируемости представляют для вас наибольший интерес.

В начале этого раздела я уже говорил, что у потоков не всегда есть чем заняться. Иногда они вынуждены ждать другие потоки, завершения ввода/вывода или еще чего-то. Если на время этого ожидания загрузить систему какой-нибудь полезной работой, такое простаивание можно «скрыть».