Министерство образования Московской области
Государственный университет «Дубна»
Институт системного анализа и управления
Кафедра распределенных информационно-вычислительных систем

**М. В. Башашин, Е. В. Земляная,
О. И. Стрельцова**

Основы технологии OpenMP на кластере HybriLIT

УЧЕБНОЕ ПОСОБИЕ

Рекомендовано учебно-методическим советом университета «Дубна» в качестве учебного пособия для студентов, обучающихся по направлениям подготовки «Информатика и вычислительная техника», «Программная инженерия», «Фундаментальная информатика и информационные технологии», «Информационные системы и технологии», «Прикладная математика и информатика», «Автоматизация технологических процессов и производств»
(бакалавриат)

Дубна
2020

УДК 004.42
ББК 32.973я73
Б 33

Р е ц е н з е н т :
доктор физико-математических наук, доцент *Д. С. Кулябов*

Башашин, М. В.

Б 33 Основы технологии OpenMP на кластере HybriLIT : учебное пособие / М. В. Башашин, Е. В. Земляная, О. И. Стрельцова. — Дубна : Гос. ун-т «Дубна», 2019. — 50 [2] с.

ISBN 978-5-89847-598-7

Пособие предназначено для практического освоения технологии *OpenMP* (*Open specifications for Multi-Processing*) с использованием ресурсов гетерогенной вычислительной платформы *HybriLIT* многофункционального информационно-вычислительного центра Лаборатории информационных технологий ОИЯИ в рамках практикума по введению в основы параллельного программирования. Излагаются базовые принципы и основные конструкции технологии *OpenMP*, описан порядок создания и запуска *OpenMP*-приложений на кластере *HybriLIT*, рассматриваются вопросы создания комбинированных приложений на основе совместного использования технологий *MPI* и *OpenMP*.

Данное пособие ориентировано на студентов государственного университета «Дубна», изучающих курс «Архитектура вычислительных систем» (бакалавриат), а также может использоваться для самостоятельного освоения технологии *OpenMP*.

УДК
ББК

ISBN 978-5-89847-598-7

Оглавление

Глава 1. Общие сведения о технологии OpenMP и подготовке OpenMP-приложений на кластере HybriLIT

1.1. OpenMP: общая характеристика

OpenMP (*Open specifications for Multi-Processing*) – один из популярных стандартов для создания параллельных компьютерных программ. В модели OpenMP программа представляется как набор параллельных потоков (нитей, *threads*), объединенных общей памятью. Технология OpenMP включает набор специальных директив компилятору, библиотечных функций и переменных окружения.

Директивы компилятору (прагмы) используются для обозначения областей в программе с возможностью параллельного выполнения. Компилятор, поддерживающий OpenMP, преобразует исходный код и вставляет соответствующие вызовы функций для параллельного выполнения этих областей. Компилятором, не поддерживающим OpenMP, директивы игнорируются. Тем самым, открывается возможность создания единого кода для параллельного и последовательного выполнения одной и той же задачи, а также возможность относительно простой «пошаговой» параллельной оптимизации уже существующих последовательных программ.

В отличие, например, от другой популярной технологии параллельного программирования MPI, реализующей модель распределенной памяти, особенностью технологии OpenMP является ориентация на компьютерные системы с общей памятью, в которых доступ к любой ячейке памяти имеют все имеющиеся параллельные узлы (рис. 1).

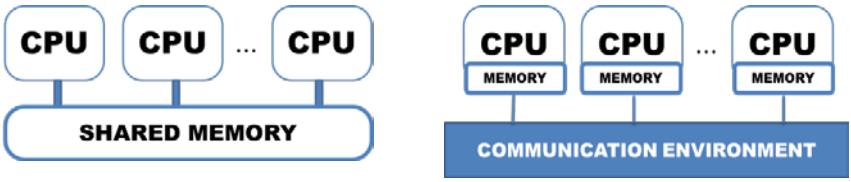

Рис. 1. Иллюстрация систем с общей (слева) и распределенной (справа) памятью

Рассмотрим особенности OpenMP в сравнении с технологией MPI, уже описанной нами в пособии «Практическое введение в технологию MPI на кластере *HybriLIT*» [2].

- В отличие от MPI, где каждый параллельный процесс имеет доступ только к своей локальной памяти, в технологии OpenMP взаимодействие между параллельными потоками (нитями, *threads*) осуществляется за счет общих переменных (*shared*).
- В отличие от MPI, где все операции компьютерного кода выполняют все задействованные параллельные процессы, в технологии OpenMP работу начинает одна нить (нить-мастер). Параллельные области организуются с помощью специальных OpenMP-директив. При входе в параллельную область (*fork*) порождаются дополнительные нити, каждая из которых получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области (*join*) происходит неявная синхронизация нитей, и дальнейшее выполнение программы продолжает только нить-мастер. Таким образом, программа разбивается на последовательные и параллельные области (рис. 2).

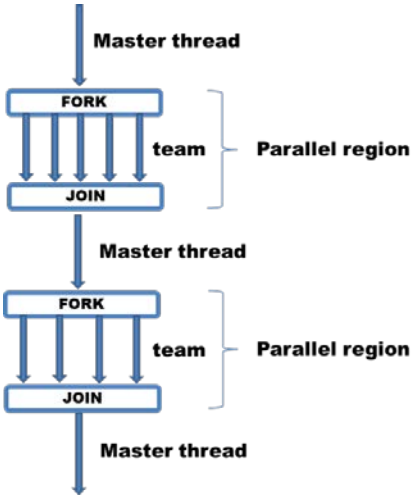

Рис. 2. Структура OpenMP-программы, состоящей из параллельных и последовательных областей

- В отличие от MPI, реализующей только явное распараллеливание, в OpenMP существуют также возможности высокоуровневого параллелизма за счет использования конструкций для автоматического распараллеливания циклов и независимых фрагментов.
- Синтаксис OpenMP главным образом основан на директивах. В C\C++ все директивы начинаются с `#pragma omp`.

Отметим, что такие особенности OpenMP, как наличие общих переменных и возможности для высокоуровнего параллелизма, делают технологию OpenMP удобной для создания параллельных приложений. Однако в данной технологии, как и в других, существуют свои «подводные камни», требующие специального внимания разработчика параллельных программ. В частности, при создании OpenMP-приложений следует учитывать затраты компьютерного времени на организацию параллельных секций и на работу с общими переменными.

1.2. Компиляция и запуск OpenMP-приложений на кластере HybriLIT

Общие сведения об особенностях гетерогенной платформы HybriLIT, о порядке работы на кластере, а также базовые команды *OS LINUX*, популярных редакторов *NANO* и *VIM* и пакетов *SLURM* и *MODULE* для запуска задач представлены в Приложениях 1, 2, 3.

Для компиляции OpenMP-программ используются стандартные компиляторы с ключами поддержки OpenMP. На кластере HybriLIT доступны компиляторы *GNU* и *Intel*. *Intel*-компилятор позволяет достичь высокой степени оптимизации кода для процессоров *Intel*, которые и лежат в основе архитектуры кластера HybriLIT. Данный компилятор полностью совместим с компилятором *GNU C* на уровне исходного кода и объектных модулей, что без дополнительных затрат позволяет осуществлять миграцию на него приложений, созданных с помощью *GNU C*.

В настоящее время (январь 2020) на кластере HybriLIT доступны компиляторы *GNU* версии 4.8.5 (установлен по умолчанию), 4.9.3, 5.3.0, 6.2.0, 7.2.0, 8.2.0, 9.1.0, а также компилятор из

состава *Intel Cluster Studio* с поддержкой OpenMP. Перед компиляцией, так же, как и при работе с технологией MPI, необходимо загрузить соответствующий модуль (Приложение 3).

В табл. 1 представлены команды для компиляции программ, написанных на C, C++ для компиляторов *Intel* и *GNU*.

Таблица 1. **Команды компиляции, используемые в компиляторах**

	Intel	**GNU**
C	`icc -qopenmp hello.c`	`gcc -fopenmp hello.c`
C++	`icpc -qopenmp hello.cpp`	`g++ -fopenmp hello.cpp`

Запуск OpenMP-приложений осуществляется с помощью скрипт-файла, содержащего следующую информацию:

```
#!/bin/sh
#SBATCH -p cpu # выбор типа используемой очереди
#SBATCH -c 5   # установка числа вычислительных
               # потоков (нитей)
#SBATCH –t 60  # установка времени расчета
export OMP_NUM_THREADS=5 # установка значения
       # переменной окружения OMP_NUM_THREADS,
       # определяющей число OMP-нитей
export OMP_PLACES=cores  # оптимизация
            # распределения потоков
            # по вычислительным ядрам
./a.out     # запуск приложения
```

Для запуска приложения используется команда `sbatch`. Более подробную информацию см. в Приложениях 1 и 3.

1.3. Создание параллельных областей в OpenMP-программе

1.3.1. Прагма `parallel`

Директива `parallel` создаёт группу из `N` потоков (нитей). Количество нитей `N` определяется во время выполнения, обычно это число ядер процессора, но также можно задать `N` вручную. Каждый из потоков в группе выполняет следующую за директивой команду или блок команд, заключённый в `{}` скобках.

После выполнения параллельного фрагмента потоки «сливаются» в один. Прагма `parallel` может иметь ряд опций, которые записываются в строку через пробел. Также при необходимости в этой же строке объявляются локальные (`private`) и общие (`shared`) переменные. Таким образом, директива `parallel` выглядит следующим образом:

```
#pragma omp parallel [необязательные опции]
{ параллельная область }
```

Общая структура OpenMP-программы имеет вид:

```
#include <omp.h>
...
int main(){
//последовательная область
#pragma omp parallel
{
//Первая параллельная область
}
//последовательная область
#pragma omp parallel
{
//Вторая параллельная область
}
//последовательная область
...
return 0;
}
```

После выполнения параллельного региона все потоки, кроме мастер-нити, прекращают свое существование. По умолчанию синхронизация не предусмотрена, поэтому определить последовательность выполнения конкретного оператора разными нитями не представляется возможным.

1.3.2. Регулирование количества нитей в параллельной области

Существуют следующие способы задания числа потоков в параллельных областях.

- Количество нитей в параллельных областях запускаемых OpenMP-задач регулируется переменной окружения `OMP_NUM_THREADS`, уже описанной выше в скрипт-файле и определяющей количество нитей во всех запускаемых OpenMP-программах. Для изменения установленного по умолчанию значения в командном режиме используется команда `export`.

  Пример:

  ```
  > export OMP_NUM_THREADS=4
  ```

- Функция `omp_set_num_threads(int)` позволяет указать желаемое число нитей во всех параллельных блоках данной программы, в то время как при запуске других программ число нитей будет соответствовать установленному значению `OMP_NUM_THREADS`.

  Пример:

  ```
  ...
  omp_set_num_threads(3);
  #pragma omp parallel
  {…}
  #pragma omp parallel
  {…}
  ...
  ```

В данном фрагменте программы устанавливается количество нитей, равное 3, во всех параллельных блоках программы.

- Параметр `num_threads` прагмы `parallel` позволяет указать число нитей в конкретном параллельном блоке, в то время как число нитей в остальных параллельных областях данной программы и в других запускаемых OpenMP-программах определяется согласно двум вышеприведенным вариантам.

  Пример:

```
...
omp_set_num_threads(3);
   ...
#pragma omp parallel num_threads(2)
{printf("Hello\n";}
...
#pragma omp parallel
{printf("Goodbye\n";}
...
```

В данном фрагменте программы устанавливается количество нитей, равное 3, во всех параллельных блоках программы. Однако первый блок, печатающий "Hello", будет выполнен 2 раза.

1.3.3. Функции для определения количества нитей и нумерации нитей в параллельной области

Потоки в параллельной области нумеруются, начиная с 0 до `OMP_NUM_THREADS`-1. Основному потоку, инициировавшему организацию параллельного региона, всегда присваивается номер 0. Он называется *master-thread* (нить-мастер). Получить свой номер каждая нить может с помощью функции `omp_get_thread_num()`.

Для определения числа нитей в параллельной области используется функция `omp_get_num_threads()`.

В нижеследующем примере представлен фрагмент OpenMP-программы, иллюстрирующий применение указанных функций.

```
#pragma omp parallel
{
int rank, size;
rank = omp_get_thread_num();
size = omp_get_num_threads();
if(rank==0) printf(“hello\n”);
if(rank==size-1) printf(“goodbye\n”);
}
```

Здесь целочисленные переменные `rank`, `size` объявлены внутри параллельной области. Это означает, во-первых, что они являются локальными, т.е. каждая нить имеет свой экземпляр переменной с этим именем, и, во-вторых, что вне данной параллельной области указанные переменные не существуют.

В вышеприведенном примере в результате вызова функций `omp_get_thread_num()` и `omp_get_num_threads()` каждая нить записывает свой номер в переменную `rank` и количество нитей в переменную `size`. Тем самым, открывается возможность распределения работы между нитями с явным указанием, что должна делать нить с конкретным номером. Так, в нашем примере нить с номером 0 выводит на дисплей слово “hello”, а нить с максимальным номером `size`-1 печатает слово “goodbye”.

1.4. Примеры самостоятельных заданий

1. Организовать с помощью прагмы `parallel` и функции `omp_set_num_threads` параллельную область с количеством нитей, равным 5. Организовать вызов каждой нитью в параллельном блоке функций `omp_get_num_threads` и `omp_get_thread_num` и вывод на дисплей каждой нитью ее номера и числа нитей в этом параллельном блоке.
2. Организовать с помощью прагмы `parallel` и функции `omp_set_num_threads` параллельную область с количеством нитей, равным 6. Организовать вызов каждой нитью в параллельном блоке функций `omp_get_num_threads` и

`omp_get_thread_num` и вывод на экран каждой нитью, кроме мастер-нити, своего номера. Нить с номером 0 должна вывести число нитей в параллельном блоке.

3. Организовать с помощью прагмы `parallel` и опции `num_threads` параллельную область с количеством нитей, равным 4. Организовать вызов функций `omp_get_num_threads` и `omp_get_thread_num` каждой нитью в параллельном блоке. Каждая нить в параллельной области должна вывести на экран свой номер. Нить с максимальным номером должна также напечатать число нитей в параллельном блоке.
4. Организовать с помощью прагмы `parallel` и опции `num_threads` параллельную область с количеством нитей, равным 8. Каждая нить параллельного блока с четным номером должна напечатать слово “Hello”, каждая нить с нечетным номером – слово “Goodbye”.
5. Организовать с помощью прагмы `parallel` и опции `num_threads` параллельную область с количеством нитей, равным 10. Нить с максимальным номером должна вызвать функцию `omp_get_num_threads` и напечатать количество нитей в параллельном блоке. Нить-мастер должна напечатать слово “Hello” и свой номер.

Глава 2. Основные конструкции технологии OpenMP для организации параллельных вычислений

2.1. Классификация переменных OpenMP

2.1.1. Общие и локальные данные

Использование системы с общей памятью накладывает отпечаток на переменные, которые будут использоваться в параллельном регионе. В OpenMP существуют как общие, так и локальные переменные. Необходимо учитывать особенности работы с каждым типом, в том числе тот факт, что доступ у каждой нити к общей переменной и, соответственно, возможность изменить ее значение в случае некорректного использования общих переменных может привести к неправильной работе программы.

В OpenMP определены следующие типы переменных:

- `shared` – такие переменные являются общими для всех нитей, т.е. все нити будут обращаться к одной области памяти;
- `private` – каждый поток имеет свою копию переменной, значение которой обновляется независимо каждой нитью и не сохраняется за пределами параллельной области;
- `firstprivate` – аналогична `private`, но переменные данного типа при входе в параллельный регион инициализируются значением, которое переменная имела перед открытием этого региона;
- `lastprivate` – аналогична `private`, но после закрытия параллельной области переменные сохраняют значение, полученное в ходе вычислений внутри параллельного региона. Используется в конструкциях `for` и `section`.

На кластере HybriLIT типом переменной по умолчанию является `shared`. Для отмены этой установки можно воспользоваться опцией `default (none)`.

Нижеприведенные фрагменты иллюстрируют особенности использования переменных типа `shared`, `private` и `firstprivate`.

Пример 1. Здесь переменная c объявлена локальной, в то время как a и b по умолчанию являются общими. Внутри параллельного блока каждая нить вычисляет значение своей локальной переменной c как сумму a+b и выводит это значение на экран. Однако после завершения выполнения параллельной области значение переменной c не сохраняется, так что мастер-нить напечатает c = 0.

```
int a=5, b=3, c=0;
#pragma omp parallel private(c)
{
c=a+b;
printf("in parallel: a=%d, b=%d, c=%d\n");
}
printf("in master-thread: a=%d, b=%d, c=%d\n");
```

Пример 2. Здесь отменена установка типа переменных по умолчанию, переменные a и b объявлены общими, а переменная c объявлена локальной, так что ее значение, как и в предыдущем примере, не передается из последовательной области в параллельную и наоборот. Поэтому результаты выполнения данного примера будут такими же, как в примере 1.

```
int a=5, b=3, c=0;
#pragma omp parallel default(none) private(c)
shared (a,b)
{
printf("in parallel: a=%d, b=%d, c=%d\n");
c=a+b;
printf("in parallel after sum: a=%d, b=%d,
c=%d\n");
}
printf("in master-thread: a=%d, b=%d, c=%d\n");
```

Пример 3. Здесь все переменные имеют тип firstprivate, т.е. каждая нить имеет свой экземпляр каждой из этих переменных, причем во всех нитях при открытии параллельного блока имеем a = 5, b = 3, c = 0. После операции c = a + b значение переменной c в каждой нити обновляется. Однако это новое значение не пере-

дается за пределы параллельной области. После завершения параллельного блока мастер-нить имеет значение `c = 0`.

```
int a=5, b=3, c=0;
#pragma omp parallel firstprivate(a,b,c)
{
printf("in parallel: a=%d, b=%d, c=%d\n");
c=a+b;
printf("in parallel after sum: a=%d, b=%d,
c=%d\n");
}
printf("in master-thread: a=%d, b=%d, c=%d\n");
```

Пример 4. Здесь переменные `a` и `b` имеют тип `firstprivate`, в то время как переменная `c` является общей для всех нитей. В результате суммирования каждая нить обновляет значение общей переменной `c`. При этом, поскольку `a` и `b` в каждой нити имеют одинаковые значения, все нити вычисляют одно и то же значение `c`, т.е. результат не зависит от порядка доступа нитей к общей переменной. Поскольку переменная `c` является общей, ее новое значение сохраняется в мастер-нити за пределами параллельной области, т.е. после завершения параллельного блока мастер-нить выведет на дисплей значение `c = 8`.

```
int a=5, b=3, c=0;
#pragma omp parallel firstprivate(a,b)
shared(c)
{
printf("in parallel: a=%d, b=%d, c=%d\n");
c=a+b;
printf("in parallel after sum: a=%d, b=%d,
c=%d\n");
}
printf("in master-thread: a=%d, b=%d, c=%d\n");
```

2.1.2. Переменные окружения OpenMP

Переменные окружения необходимы для конфигурирования запуска параллельного OpenMP-приложения средствами операци-

онной системы. Для операционных систем семейства *Linux* изменение данных переменных осуществляется через команду `export`, для операционных систем *Windows* – через команду `set`.

- `OMP_NUM_THREADS` – устанавливает количество потоков в параллельном блоке. По умолчанию количество нитей равно количеству виртуальных процессоров.
- `OMP_SCHEDULE` – устанавливает тип распределения работ в параллельных циклах с типом `runtime`.
- `OMP_DYNAMIC` – разрешает или запрещает динамическое изменение количества нитей, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации на конкретной вычислительной системе.
- `OMP_NESTED` – разрешает или запрещает вложенный параллелизм, т.е. создание вложенной параллельной области одной из нитей уже созданного параллельного блока. По умолчанию – запрещено.

2.2. Функции OpenMP

Для рационального использования ресурсов вычислительной системы и написания гибких OpenMP-программ пользователю предоставляется возможность управлять ходом выполнения программы посредством библиотечных функций. Перед использованием функций необходимо подключить заголовочный файл `omp.h`.

Три наиболее часто используемые функции уже рассмотрены в предыдущих разделах:

- `void omp_set_num_threads(int num_threads)` – устанавливает количество нитей, которое может быть запрошено для параллельного блока;
- `int omp_get_num_threads()` – возвращает количество нитей, доступных в данный момент;
- `int omp_get_thread_num()` – возвращает номер нити (целое число от 0 до `OMP_NUM_THREADS` – 1).

Перечислим здесь некоторые другие полезные функции OpenMP:

- `int omp_get_max_threads()` – возвращает максимальное количество нитей, которое может быть установлено

с помощью функции `omp_set_num_threads` на данной вычислительной системе;

- `int omp_get_num_procs()` – возвращает количество физических процессоров, доступных программе на данной вычислительной системе;
- `int omp_in_parallel()` – возвращает ненулевое значение, если функция вызвана внутри параллельного блока, в противном случае возвращается 0; функция полезна при отладке сложных комплексов программ, состоящих из множества модулей;
- `double omp_get_wtime()` – функция для работы с системным таймером, позволяющая организовать замеры времени выполнения фрагментов кода;
- `double omp_get_wtick(void)` – функция для измерения разрешения таймера. Позволяет оценить точность измерения времени выполнения программ с помощью функции `omp_get_wtime`.

2.3. Конструкции для параллельного выполнения циклов

Как уже отмечалось в главе 1, наряду с возможностями организации явного параллелизма с использованием нумерации нитей в параллельных блоках технология OpenMP предоставляет возможности для автоматического распараллеливания на основе указаний компилятору с помощью соответствующих директив. Наиболее успешно распараллеливанию подвергаются итерационные процессы и алгоритмы, реализуемые в виде циклов. Для их распараллеливания в технологии OpenMP используется директива (прагма) `for`.

2.3.1. Прагма `for`

Директива `for` является указанием компилятору, что итерации данного цикла можно выполнить параллельно. Распределение итераций цикла между нитями возлагается на компилятор и зависит от количества нитей, выполняющих данный фрагмент кода, а также от количества итераций в цикле.

После применения директивы `for` обязательным условием является нахождение в следующей строке соответствующего оператора цикла. С точки зрения написания кода директива может быть применена следующими способами.

- Подразумевается, что параллельный регион создается только для распараллеливания цикла. В этом случае можно совместить директивы `parallel` и `for` в одной строке.

```
……….
#pragma omp parallel for
      for(i=0; i<N;i++){
……….
}
```

- Подразумевается, что в параллельной области, помимо распараллеливаемого цикла, присутствуют какие-либо еще вычисления до и\или после цикла. В этом случае директива `for` применяется внутри параллельной области непосредственно перед оператором цикла.

```
#pragma omp parallel
{
……….
#pragma omp for
      for(i=0; i<N;i++){
……….
}
……….
}
```

Отметим, что применение директивы `for` в области, выполняемой одной нитью, не является ошибкой, однако в этом случае цикл будет выполнен в последовательном режиме.

Ниже приведен фрагмент программы, демонстрирующий параллельное поэлементное перемножение элементов двух массивов.

```
int N=500;

  double *a,*b,*c;
  int i;
  double time1,time2;
```

```
a=(double*)malloc(N*sizeof(double));
b=(double*)malloc(N*sizeof(double));
c=(double*)malloc(N*sizeof(double));

memset(a,(double)rand(),N*N*sizeof(double));
memset(b,(double)rand(),N*N*sizeof(double));
memset(c,0,N*N*sizeof(double));
omp_set_num_threads(5);
#pragma omp parallel shared (a,b,c) private (i)
{
       #pragma omp for
       for(i=0;i<N;i++){
             c[i] = a[i] * b[i];
       }
}
```

Отметим, что компилятор не проверяет зависимость по данным (рекурсию) между итерациями цикла и, соответственно, корректность параллельной реализации. Эта ответственность полностью лежит на разработчике параллельного приложения.

2.3.2. Опции для распределения итераций между нитями

В общем случае, поскольку ответственность за распараллеливание цикла возлагается на компилятор, пользователь не знает, каким образом итерации будут распределяться между нитями. Однако существуют возможности регулирования распределения итераций цикла между нитями в параллельном регионе с помощью опций директивы `schedule`. Предусматриваются следующие варианты:

- `static` – статическое распределение, является распределением по умолчанию. Итерации распределяются по нитям равными (или почти равными) блоками. В этом можно убедиться, если в предыдущем примере добавить в цикл оператор вывода на экран номера нити и индекса цикла. Статическое распределение вполне эффективно при приблизительно равном времени выполнения итераций.

```
.....................
#pragma omp parallel for schedule(static)
for(i=0; i<N;i++){
```

```
……
}
```

- `dynamic` – итерации распределяются блоками заданного размера (по умолчанию размер равен 1) между нитями. Как только какая-либо из нитей заканчивает обработку своей порции данных, она захватывает следующую. Стоит отметить, что при этом подходе несколько большие накладные расходы, но при больших объемах данных можно добиться лучшей балансировки загрузки между потоками.

```
……
#pragma omp parallel for schedule(dynamic,3)
for(i=0; i<N;i++){
……
}
```

В данном примере задается размер блока итераций, равный 3.

- `guided` – данный тип распределения итераций аналогичен предыдущему, за тем исключением, что размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения. При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.

```
……
#pragma omp parallel for schedule(guided,10)
for(i=0; i<N;i++){
……
}
```

Здесь начальный размер блока итераций задан равным 10.

Отметим, что при завершении цикла происходит неявная синхронизация всех нитей, задействованных в выполнении цикла, т.е. все нити ждут друг друга и только после завершения обработки

всего цикла начинают дальнейшие вычисления. В некоторых случаях это может привести к существенным потерям времени. Избежать этого помогает опция `nowait`.

2.4. Параллельное выполнение независимых фрагментов

Для случаев, когда исходная программа имеет независимые модули, которые могут быть выполнены параллельно, OpenMP предоставляет еще одну возможность автоматического распараллеливания – с помощью директивы `sections`. Эта директива используется для распределения независимых фрагментов (секций) программного кода между потоками, при этом каждый поток будет выполнять свою секцию. Выбор конкретной нити, которая будет выполнять данную секцию, осуществляется компилятором. Если секций больше, чем потоков, то несколько секций будут выполняться одной нитью по очереди.

Нижеприведенный фрагмент демонстрирует схему использования директивы `sections`. В параллельной области, организованной с помощью прагмы `parallel`, вся совокупность независимых фрагментов выделяется прагмой `sections`, а каждый из независимых фрагментов этой совокупности выделяется с помощью прагмы `section`.

```
#pragma omp parallel
{
...
#pragma omp sections
{
#pragma omp section
{
...
}
#pragma omp section
{
...
}
}
...
}
```

Ниже в качестве примера приведен фрагмент программы, производящей на основе формулы трапеций численное интегрирование функций, заданных в Sqr1 и Sqr2.

```
……….
double Sqr1(double x){
double c = 11;
return(
0.1*x*x*x+0.2*x*x+1.0+x/(x+c+0.5)/(x+0.5);
}
double Sqr2(double x){
double c = 12;
return(
0.1*x*x*x-0.2*x*x*1.0/(x+c+1)/(x+c*c+2);
}
……….
      int N=100000001;

      int i;
      double h, a, b, x, f;
      double  S1=0;
      double  S2=0;
      double  S3=0;

      a = 1.0;
      b = 11.0;
      h = (b-a)/(N-1);

      omp_set_num_threads(2);

      #pragma omp parallel shared (a,h,N,S1,S2)
private (i,x)
      {
      #pragma omp sections
                  {
                  #pragma omp section
            {
```

```
        // first integral

                    for(i=1;i<N-1;i++){
                          x = a+h*i;
                          S1 += Sqr1(x);
                    }
        S1 += (Sqr1(a)+Sqr1(b))*0.5;
        S1 *= h;
        }
              #pragma omp section
        {
        // second integral

                    for(i=1;i<N-1;i++){
                          x = a+h*i;
                          S2 += Sqr2(x);
                    }
        S2 += (Sqr2(a)+Sqr2(b))*0.5;
        S2 *= h;
        }
}
……..……
```

Здесь в параллельном режиме вычисляются два интеграла с пределами интегрирования `a` и `b` и подынтегральными выражениями, определяемыми функциями `Sqrt1` и `Sqrt2`; `h` – шаг равномерной дискретной сетки по аргументу x с количеством узлов `N`, используемый в квадратурной формуле трапеций.

Отметим, что совместное использование директив `sections` и `for` запрещено: нельзя организовывать параллельное выполнение независимых фрагментов внутри параллельного цикла; нельзя организовывать параллельное выполнение цикла в области `sections`.

2.5. Примеры самостоятельных заданий

1. Написать OpenMP-программу, выполняющую следующее. Объявляются целочисленные переменные `x`, `y`, `z`, имеющие

значения x = 1, y = 10, z = 20. Организуется параллельная область, в которой переменная x является shared, переменная y является private, переменная z является firstprivate. Внутри параллельной области осуществляются операции x+ = 1, y+ = 1, z+ = 1. После завершения параллельной области необходимо вывести на экран значения переменных x, y, z и объяснить полученный результат.

2. Написать OpenMP-программу, выполняющую следующее. Задается массив X из 20 целых чисел, каждое из которых равно 15. Устанавливается число нитей в параллельных фрагментах, равное 5. Внутри параллельного блока с помощью прагмы for организуется параллельное выполнение цикла, в котором каждый элемент массива X умножается на 2 и выводится на печать номер этого элемента и номер нити. После завершения параллельного блока выводится на экран результирующие значения массива X.
3. Написать OpenMP-программу, выполняющую следующее. Задается массив B из 30 целых чисел, каждое из которых равно 10. Внутри параллельного блока с помощью прагмы for организуется параллельное выполнение цикла, в котором к каждому элементу массива B прибавляется число, равное количеству параллельных нитей, и выводится на печать номер этого элемента и номер нити. После завершения параллельного блока выводятся на экран результирующие значения массива B.
4. Написать OpenMP-программу, выполняющую следующее. Задаются целые переменные a = 10 и b = 20, которые являются общими для параллельной области, имеющей не менее трех нитей, организованной с помощью прагмы parallel. Внутри параллельной области с помощью прагмы sections организуются три назывисимых блока, в первом из которых вычисляется c = a + 1, во втором d = b * 2, в третьем f = a + b. После завершения параллельной области выводятся на экран результирующие значения c, d, f.
5. Написать OpenMP-программу, выполняющую следующее. Задаются целые переменные, a = 20 и b = 2, которые являются общими для параллельной области, имеющей не менее четырех нитей, организованной с помощью прагмы parallel. Внутри параллельной области с помощью прагмы sections

организуются четыре независимых блока, в первом из которых вычисляется `c = a * b + 10`, во втором `d = (a + b) * 2`, в третьем `f = (a + b)/2`, в четвертом `x = a – b + 1`. После завершения параллельного блока выводятся на экран результирующие значения `c, d, f, x`.

Глава 3. Конструкции OpenMP для синхронизации нитей и редукции

3.1. Синхронизация нитей

Директивы синхронизации обеспечивают возможность организации явной синхронизации работы нитей в параллельной области, а также синхронизации путем назначения специального режима для выполнения фрагмента, выделенного внутри параллельной секции. Например, такой фрагмент может быть выполнен только одной нитью или всеми нитями, но по очереди. Директивы данного класса позволяют обеспечить корректную работу с общими переменными, а также организовать однократное выполнение выделенного фрагмента без закрытия параллельной области, что, как уже упоминалось в главе 1, требует существенных затрат компьютерного времени.

3.1.1. Прагмы `single` и `master` для однократного выполнения фрагментов

Директива `single` указывает на однократное выполнение выделенного внутри параллельной области блока произвольной нитью. Ситуации применения данной директивы различны: от однократного выполнения какой-либо арифметической операции до ручного открытия/закрытия синхронизирующего семафора.

В нижеследующем примере внутри представлен фрагмент программы, иллюстрирующий применение директивы `single` для однократного выполнения какой-либо нитью операции увеличения общей целочисленной переменной `count`. Остальные нити пропускают выделенный фрагмент, как это показано на рис. 3.

```
int count=0;
#pragma omp parallel
{...
#pragma omp single
{
count++;
}
...}
```

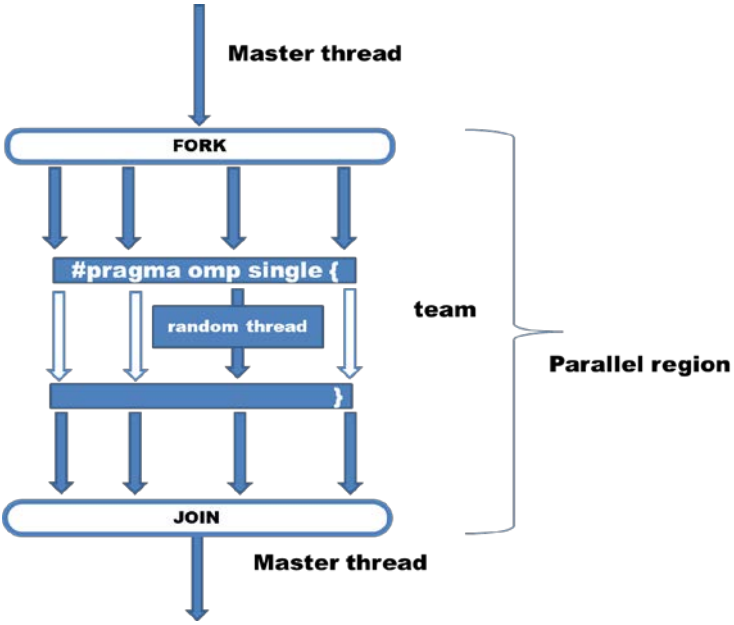

Рис. 3. Иллюстрация применения директивы `single`

Директива `master`, подобно `single`, также указывает на однократное выполнение блока, однако выделенный блок будет выполнен не произвольной нитью, как в случае применения директивы `single`, а мастер-нитью, имеющей внутри параллельной области номер 0 (рис. 4). Синтаксис оформления `master`-фрагмента аналогичен применению прагмы `single`:

```
int count=0;
#pragma omp parallel
{...
#pragma omp master
{
count++;
}
...}
```

В результате выполнения двух последних фрагментов значение переменной `count` будет равно 1.

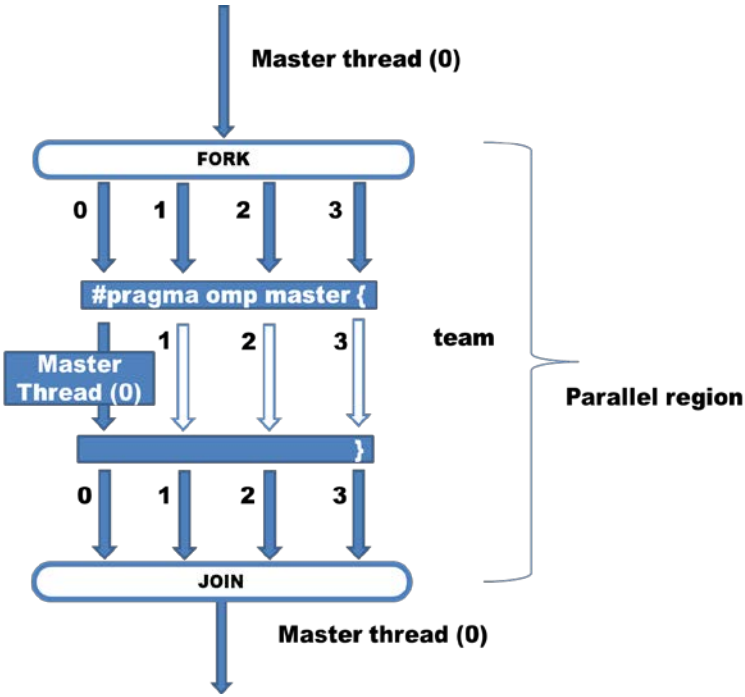

Рис. 4. Иллюстрация применения директивы `master`

3.1.2. Прагмы `critical`, `atomic`, `ordered` для организации критических секций

Критическая секция, организуемая в OpenMP с помощью директив `critical` и `atomic`, выполняется всеми нитями, задействованными в данной параллельной области, однако в каждый конкретный момент критический фрагмент выполняется только одной нитью, т.е. когда одна нить выполняет содержимое критической секции, остальные нити, готовые к выполнению критической секции, находятся в режиме ожидания (рис. 5). Очередность выполнения нитями критической секции не определена. Если несколько нитей ожидают выполнения критического фрагмента, порядок входа нитей в критическую секцию осуществляется случайным образом.

В приведенном ниже фрагменте каждая из нитей в организованной критической секции увеличивает значение общей переменной `count` на 1. В результате ее значение становится равным количеству нитей, выполняющих данную параллельную область, т.е. `count = 3`.

```
int count=0;
#pragma omp parallel num_threads(3)
{...
#pragma omp critical
{
count++;
}
...}
```

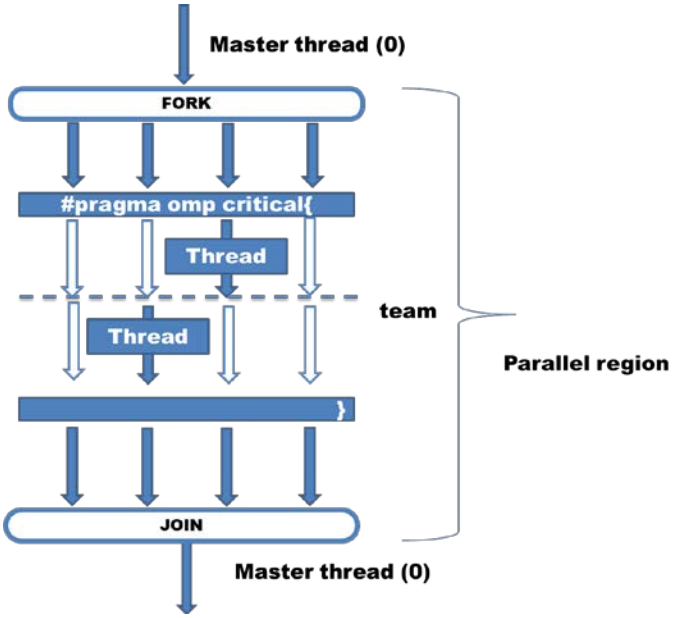

Рис. 5. Иллюстрация применения директивы `critical`

Директива `atomic` действует аналогично директиве `critical`, но относится только к идущему непосредственно оператору:

```
#pragma omp atomic
   count++;
```

Отметим, что в языках C и C++ имеются ограничения на операторы, которые могут применяться в `atomic`-секции.

Еще одна директива для выделения критических секций – директива `ordered`, она используется только в параллельных циклах. Эта директива обеспечивает упорядоченное выполнение указанного блока всеми нитями по очереди, по порядку их номеров. Внутри `ordered`-секции в каждый момент времени может находиться только один поток.

```
#pragma omp parallel private(myid)
{
 myid = omp_get_thred_num();
 #pragma omp for private(i)
 for(i=0; i<8; i++)
       #pragma omp ordered
       printf("myid=%d; i=%d\n", myid, i);
}
```

В приведенном здесь фрагменте с помощью директивы `ordered` обеспечивается печать каждой нитью выполняемых ею номеров итераций цикла, в порядке возрастания номеров нитей `myid`.

3.1.3. Прагмы `barrier` и `flush`

Барьерная синхронизация подразумевает создание некоторой «преграды», которая останавливает выполнение нити до тех пор, пока до данного барьера не доберутся все нити параллельного региона. После того как все нити дошли до барьера, выполнение кода всеми нитями продолжается. В OpenMP за барьерную синхронизацию отвечает директива `barrier`.

```
#pragma omp parallel
```

```
{
...
#pragma omp barrier
...
}
```

Синхронизация типа `flush` используется для обновления локальных переменных в оперативной памяти.

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени каждая нить будет «видеть» единый согласованный образ памяти. Именно для этих целей и предназначена данная директива:

```
#pragma omp flush  (var1, var2,…)
```

В результате выполнения директивы `flush` значения всех перечисленных в круглых скобках переменных, временно хранящиеся в регистрах, будут занесены в основную память, и все изменения переменных, сделанные нитями во время их работы, станут «видимыми» остальным нитям. Таким образом, после выполнения данной директивы гарантируется, что все указанные переменные будут иметь одно и то же значение для всех параллельных потоков. Если список переменных прагмы `flush` не указан, она применяется в отношении всех переменных, задействованных в данной программе. Применять данную директиву следует с осторожностью, поскольку ее выполнение в полном объеме может повлечь значительные накладные расходы.

3.2. Редукция в OpenMP

Порядок выполнения операторов над общими данными внутри параллельных областей не определён, поэтому при некорректном написании программы результат может отличаться от запуска к запуску. Наряду с директивами синхронизации, обеспечить корректные действия с общими данными позволяет опция `reduction` прагмы `parallel`.

Механизм редукции на основе опции `reduction` задаёт оператор и список общих переменных, над которыми указанная глобальная операция производится. Для каждой переменной создаются локальные копии в каждой нити. Локальные копии инициализируются соответственно типу оператора (так, для операций сложения начальное значение – 0, для операций умножения – 1). После завершения всех операций параллельного блока над локальными копиями переменных и над исходным значением переменной редукции в мастер-нити перед параллельным блоком выполняется заданный глобальный оператор.

В нижеприведенном фрагменте объявлена переменная редукции `nCount`, над которой по завершении параллельной области будет выполнена операция сложения. При открытии параллельной секции в каждой нити появляется свой экземпляр целочисленной переменной с именем `nCount`. Несмотря на заданное в мастер-нити значение `nCount=10`, все локальные переменные `nCount` имеют начальное значение 0. Далее каждая нить увеличивает значение своей переменной `nCount` на 1. По завершении параллельной секции все эти значения суммируются, результат (равный количеству нитей в параллельной области) суммируется с исходным значением 10 и присваивается переменной `nCount` в нити-мастере. Таким образом, после завершения выполнения представленного фрагмента значение переменной `count` будет равно 15.

```
omp_set_num_threads(5);
int nCount=10;
#pragma omp parallel reduction (+ : nCount)
{
...
nCount+=1;
...
}
```

Доступные для использования в OpenMP операции редукции представлены в Приложении 4 с пояснением особенностей их применения.

Таким образом, операции редукции обеспечивают корректную работу с общей памятью в параллельных областях, когда над

общими данными осуществляют операции все параллельные нити и когда требуется передача результатов вычислений нити-мастеру после завершения параллельной секции.

3.3. Примеры самостоятельных заданий

1. Написать OpenMP-программу, выполняющую следующее. Устанавливается число нитей в параллельных фрагментах, равное 3, с помощью опции `num_threads` прагмы `parallel`. Внутри параллельного блока, организованного с помощью прагмы `omp parallel`, каждая нить печатает свой номер. Организуется `master`-секция с помощью прагмы `master`, внутри которой на экран выводится фраза “Hello master”, количество нитей и номер нити, которая выполнила `master`-секцию.
2. Написать OpenMP-программу, выполняющую следующее. Задается целая переменная `a = 11`, которая является общей для параллельного блока, организованного с помощью прагмы `parallel`. Устанавливается число нитей в параллельных фрагментах, равное 4, с помощью опции `num_threads` прагмы `parallel`. Внутри параллельного блока каждая нить печатает свой номер и организуется критическая секция с помощью прагмы `critical`, внутри которой к переменной `a` прибавляется 1. После завершения параллельного блока выводится на экран результирующее значение `a`.
3. Написать OpenMP-программу, выполняющую следующее. Устанавливается число нитей в параллельных фрагментах, равное 5, с помощью функции `omp_set_num_threads`. Внутри параллельного блока, организованного с помощью прагмы `parallel`, каждая нить печатает свой номер и слово “Hello”. Нить с номером 1 печатает количество нитей. Организуется `single`-секция с помощью прагмы `single`, внутри которой на экран выводится слово “Single”, количество нитей и номер нити, которая выполнила `single`-секцию.
4. Написать OpenMP-программу, выполняющую следующее. Задается целочисленная переменная `b = 50`. Организуется параллельная область с количеством нитей 3, с переменной редукции `b` и операцией редукции «+». Внутри параллельного блока каждая нить выполняет операцию `b = 10`. После завершения параллельного блока нужно вывести на экран результирующее

значение переменной `b` и объяснить полученный результат.

5. Написать OpenMP-программу, выполняющую следующее. Задается целочисленная переменная `b = 0`. Организуется параллельная область с количеством нитей 10, с переменной редукции `b` и операцией редукции «`max`». Внутри параллельного блока каждая нить присваивает переменной `b` свой номер. После завершения параллельного блока нужно вывести на экран результирующее значение переменной `b` и объяснить полученный результат.

Глава 4. Создание приложения на базе совместного использования технологий MPI и OpenMP

В предыдущих главах описаны основные конструкции технологии OpenMP. Этот инструментарий позволяет создать параллельное приложение с относительно небольшими трудозатратами. Однако существует ряд сдерживающих факторов, которые ограничивают возможности OpenMP для организации высокопроизводительных приложений.

Главным сдерживающим фактором является то, что технология ориентирована на системы с общей памятью. Если приложение изначально «затачивается» под использование на персональных компьютерах, этот фактор не является сдерживающим. Однако запуск такой программы на кластере, состоящем из множества вычислительных узлов с распределенной памятью, ограничен ресурсами только одного физического узла.

Еще одним сдерживающим фактором является баланс нагрузки на одно вычислительное ядро. Большое количество нитей (с учетом большой вычислительной нагрузки на нить), вычисляемых на одном ядре, также может привести к падению производительности приложения.

Одним из способов улучшить распределение нагрузки на ядра является создание так называемых гибридных приложений, ориентированных на современные вычислительные системы, архитектура которых по большей части является гетерогенной. Гибридным приложением называется параллельное приложение, использующее для распараллеливания несколько технологий параллельного программирования.

В нашем случае мы будем рассматривать пример создания гибридного приложения с использованием технологий MPI и OpenMP. Главенствующую роль в данной связке можно отдать технологии MPI с ее масштабируемостью и возможностью использовать вычислительные ядра нескольких вычислительных узлов. OpenMP в свою очередь будет создавать несколько нитей на каждый MPI-процесс, обеспечивая увеличение степени параллелизма.

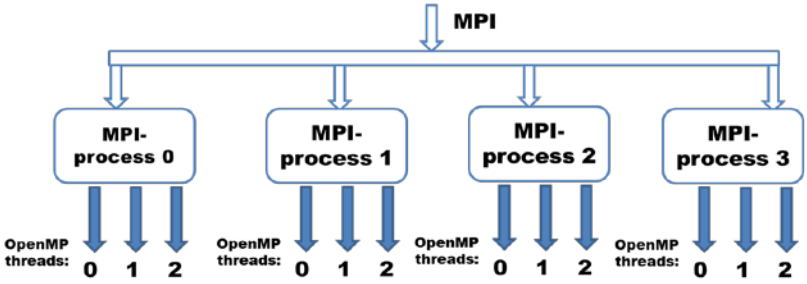

Рис. 6. Иллюстрация гибридного приложения MPI + OpenMP

4.1. Организация запуска гибридного приложения на основе MPI+OpenMP

Рассмотрим сначала порядок компиляции и запуска гибридного MPI+OpenMP-приложения на кластере HybriLIT.

Создание скрипта для запуска такого приложения позволяет системе выбрать ресурсы, характерные как для технологии MPI, так и для OpenMP. В основе скрипта лежит стандартный MPI-скрипт, используемый на кластере HybriLIT, с добавлением команд для выделения нитей OpenMP и распределения их по выделенным для MPI-ядрам. Число нитей OpenMP указано в два раза больше, чем число MPI-ядер. Таким образом, получается, что одному MPI-ядру соответствует две OpenMP-нити. Приведем пример скрипта для запуска MPI+OpenMP-приложения.

```
#!/bin/sh      # инициализация оболочки Linux
#SBATCH -p tut # выбор очереди для расчетов
#SBATCH -n 10  # выделение физических ядер (MPI)
#SBATCH -t 30  # максимальное время выполнения
               # программы
export OMP_NUM_THREADS=20 # выделение OpenMP-нитей
export OMP_PLACES=cores # распределение OpenMP-нитей
                  # по физическим ядрам
mpirun ./a.out   # Запуск программы на выполнение
```

Компиляция гибридного приложения на основе MPI+OpenMP осуществляется компилятором MPI с добавлением ключа библиотеки OpenMP.

Таблица 2. **Команды компиляции, используемые в компиляторах**

Язык	Компилятор	Команда компиляции
C	**Open MPI**	`mpicc -fopenmp hello.c`
C	**Intel**	`mpiicc -qopenmp hello.c`
C++	**Open MPI**	`mpicc++ -fopenmp hello.cpp`
C++	**Intel**	`mpiicpc -qopenmp hello.cpp`

4.2. Пример параллельной программы на основе комбинированного применения технологий MPI и OpenMP

Как уже сказано, основой параллельной реализации кода программы будет выступать MPI. В тех местах программы, где предполагается повышенная вычислительная нагрузка на MPI-процессы, вводится дополнительное распараллеливание с помощью OpenMP.

Ниже приведен пример гибридной программы по поэлементному перемножению двух массивов. OpenMP-реализация такой задачи рассмотрена в § 2.3.1 настоящего пособия, а MPI-версия такой программы обсуждается в § 4.1.2 учебного пособия «Практическое введение в технологию MPI». Здесь представлена полная версия комбинированной MPI+OpenMP-программы с использованием MPI-функций `MPI_Scatter` и `MPI_Gather` соответственно для начальной рассылки данных и финальной сборки результатов. Фактически OpenMP-оптимизация в данном случае сводится к дополнительному распараллеливанию каждого MPI-блока на основе использования директив `parallel` и `for` в основном цикле по перемножению элементов массива. Соответствующая OpenMP-прагма выделена жирным шрифтом.

```
/*
element-wise multiplication of two arrays
*/
```

```
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#define comm MPI_COMM_WORLD

#define N 50000001

int main(int argc, char* argv[])
{

      //Initialization of variables and arrays
      int imin,imax, i;

      double *a=NULL;
      double *b=NULL;
      double *c=NULL;

      double *a1=NULL;
      double *b1=NULL;
      double *c1=NULL;

      int *chunks, *disps;

// Initialization of time variables
            double t1, t2;

// Initialization and calculation of MPI variables
      int delta, np, rank, ost;

      MPI_Init(&argc,&argv);
      MPI_Comm_rank(comm,&rank);
      MPI_Comm_size(comm,&np);
      MPI_Status status;

      chunks=(int*)malloc(np*sizeof(int));
      disps=(int*)malloc (np*sizeof(int));
      if(rank==0){
```

```
a=(double*)malloc(N*sizeof(double));
b=(double*)malloc(N*sizeof(double));
c=(double*)malloc(N*sizeof(double));

// Start time
      t1= MPI_Wtime();

// Calculation of arrays a and b

      for (i=0;i<N;i++){
            a[i]=(i*0.7)+(i*1.2);
            b[i]=(i*0.4)+(i*0.2);

      }

// Filling of the array c by zeroes
      memset(c,0,N*sizeof(double));

}

//Region of division into blocks
delta=N/np;
ost=N%np;
imin= rank*delta;
imax = (rank+1)*delta;
if(rank== np-1){
      imax= N;
}

for(i= 0; i<np; i++){
      disps[i]= i*delta;
      if(i==(np-1))
            chunks[i]= N-((np-1)*delta);
      else
            chunks[i]= delta;
}

//memory allocation
```

```
a1=(double*)malloc((chunks[rank])*sizeof(double));
b1=(double*)malloc((chunks[rank])*sizeof(double));
c1=(double*)malloc((chunks[rank])*sizeof(double));

//partial sending of initial data to all MPI pro-
cesses

      MPI_Scatterv(a, chunks, disps, MPI_DOUBLE, a1,
chunks[rank], MPI_DOUBLE, 0, comm);
      MPI_Scatterv(b, chunks, disps, MPI_DOUBLE, b1,
chunks[rank], MPI_DOUBLE, 0, comm);

//---------------------------------
// Main loop
// OMP PARALLELISM

      #pragma omp parallel for private(i)
      for(i=0;i<chunks[rank];i++){

            c1[i]=a1[i]*b1[i];

      }

//Gathering of the result

      MPI_Gatherv(c1,  chunks[rank],  MPI_DOUBLE,  c,
chunks, disps, MPI_DOUBLE, 0, comm);


      if(rank==0){

      //Stop time
            t2 = MPI_Wtime();

      // Print execution time
      printf("Execution time: %f seconds \n",t2-t1);
      }
      // Free memory
```

```
        free (a);
        free (b);
        free (c);

        free (a1);
        free (b1);
        free (c1);


        MPI_Finalize();

        return 0;
}
```

Предоставляем читателю самостоятельно проверить путем тестовых расчетов эффективность данного кода в зависимости от размерности перемножаемых массивов, количества используемых MPI-процессов и числа OpenMP-нитей, приходящихся на каждый MPI-процесс.

Отметим в заключение, что при создании OpenMP-приложений, а также комбинированных приложений с использованием технологии OpenMP необходимо учитывать, что создание каждой параллельной OpenMP-области, как и ее закрытие, требует затрат компьютерного времени, эквивалентных затратам на выполнение не менее 1000 арифметических операций с плавающей точкой. Поэтому распараллеливание коротких циклов и фрагментов с небольшой вычислительной нагрузкой неэффективно. Применение директив `parallel`, `for`, `section` приведет к ускорению вычислений, только когда каждая нить в параллельной области будет выполнять объем вычислений, существенно превышающий 2000 операций.

Список рекомендуемой литературы

1. Гетерогенная вычислительная платформа HybriLIT: http://hybrilit.jinr.ru.
2. Башашин, М. В. Практическое введение в технологию MPI на кластере HybriLIT : учебное пособие / М.В. Башашин, Е.В. Земляная, О.И. Стрельцова. – Дубна : Гос. ун-т «Дубна», 2019. – 50 с.
3. Антонов, А. С. Технологии параллельного программирования MPI и OpenMP : учебное пособие. – Москва : Изд-во МГУ, 2012. – 344 с.
4. Воеводин, В. В. Параллельные вычисления : учебное пособие / В.В. Воеводин, В.В. Воеводин. – Санкт-Петербург : БХВ-Петербург, 2015. – 603 с.
5. Лупин, С. А. Технологии параллельного программирования : учебное пособие / С.А. Лупин, М.А. Посыпкин. – Москва : Форум, 2008. – 205 с.
6. Гергель, В. П. Современные языки и технологии параллельного программирования : учебник. – Москва : Изд-во Московского ун-та, 2012. – 408 с.
7. Левин, М. П. Программирование с использованием OpenMP: Современные языки и технологии параллельного программирования : учебное пособие. – Москва : БИНОМ, 2008. – 118 с.

Приложение 1. Общая характеристика учебно-тестового полигона HybriLIT

Гетерогенная вычислительная платформа HybriLIT является частью Многофункционального информационно-вычислительного комплекса (МИВК) Лаборатории информационных технологий Объединенного института ядерных исследований (Дубна) [1]. Платформа HybriLIT состоит из суперкомпьютера «ГОВОРУН» и учебно-тестового полигона HybriLIT, имеющих единую программно-информационную среду. Практические занятия по курсу «Архитектура вычислительных систем» проходят на учебно-тестовом полигоне HybriLIT (далее – кластер HybriLIT), но единая программная среда позволяет студентам получить опыт работы на суперкомпьютерах, таких как суперкомпьютер «ГОВОРУН».

Гетерогенная структура вычислительных узлов позволяет разрабатывать параллельные приложения для решения широкого круга математических ресурсоемких задач.

Для получения доступа на кластер необходимо зарегистрироваться. Для этого нужно заполнить форму на сайте кластера и получить временный пароль, который пользователь должен заменить на постоянный согласно инструкции.

Работа на кластере осуществляется в удаленном режиме по *SSH*-протоколу:

- для пользователей *Windows*: с использованием *SSH*-клиента, например, приложения *PuTTY*;
- для пользователей *Linux* и *MacOS*: с использованием команды

```
$ ssh USERNAME@hydra.jinr.ru
```

Кластер HybriLIT находится под управлением операционной системы *Scientific Linux* 7.4.

Для создания и редактирования файлов на кластере HybriLIT доступны несколько редакторов, включая **nano** и **vim**. Основные команды работы с этими редакторами приведены в Приложении 2.

На кластере HybriLIT установлены различные пакеты, библиотеки, инструменты для отладки и профилирования параллельных приложений и другое программное обеспечение (ПО). Для удобства использования только необходимого ПО на кластере установлен пакет `Modules`, предназначенный для динамического

изменения переменных окружения. Перед компиляцией приложения пользователю необходимо загрузить модули, необходимые для работы.

Для распределения ресурсов кластера между пользовательскими задачами на кластере в качестве планировщика заданий используется SLURM, а `sbatch` — команда для запуска приложения в режиме очереди. Для запуска приложения с помощью команды `sbatch` требуется использовать скрипт-файл. Наиболее часто используемые команды для работы с пакетом `Modules` и системой SLURM приведены в Приложении 3.

Детальная информация о структуре и возможностях кластера, решаемых задачах, порядке доступа к ресурсам, а также инструкция по компиляции и запуску различных приложений представлена на сайте [1]; подробная информация о модульной системе, доступных модулях, компиляторах и порядке работы на кластере представлена на сайте http://hybrilit.jinr.ru/how_to_work_on_cluster.

Приложение 2. Основные команды Linux, редакторы vim и nano

Базовые команды **Linux**

Команда	Описание
ls	выдать список файлов в текущем каталоге
ls -la	выдать подробный список, включая скрытые файлы
cd [каталог]	сменить текущий каталог, если имя каталога не указывается, то текущим становится домашний каталог пользователя
cp <*что_копировать*> <*куда_копировать*>	копировать файлы
mv <*что_перемещать*> <*куда_перемещать*>	переместить или переименовать файл
ln -s <*на_что_сделать_ссылку*> <*имя_ссылки*>	создать символическую ссылку
rm <*файл(ы)*>	удалить файл(ы)
cat <*имя_файла*>	вывод содержимого файла на стандартный вывод (по умолчанию – на экран)
find <*каталог*> -name *имя_файла*	найти файл *имя_файла* и отобразить результат на экране, поиск начинается с *<каталог>*
mc	запустить программу управления файлами *MidnightCommander*
pwd	вывести имя текущего каталога
ps -a	вывести список текущих процессов
<*имя_файла*> | grep <*фрагмент*>	поиск фрагмента текста в *<имя_файла>*
grep <*User_name*>	отобразить все процессы, запущенные от *User_name*
kill	принудительное завершение работы приложения (процесса)

killall <*имя_программы*>	принудительное завершение работы всех процессов по имени программы

Редактор **Vim**

Вызов:

vi <*имя_файла*> или **vim** <*имя_файла*>

:wq	записать файл и выйти из **vi**/**vim**
:w	сохранить изменения
:q!	выход без сохранения изменений
:x	выход с сохранением нового содержимого файла
ZZ	сохранить изменения и выйти
i	режим вставки
ESC	перейти в режим просмотра
I	добавление в начало строки
a	режим добавления
A	добавление в конец строки
dd	вырезать (удалить) строку
cc	удалить и начать редактирование
yy	копировать строку
:<№ строки>	перейти на строку №
:set number	включить нумерацию строк
:set nonumber	отключить нумерацию строк
u	отменить предыдущее изменение
U	восстановить первоначальный вид строки
p	вставить содержимое буфера или удаленные строки после текущей строки
P	поместить содержимое буфера или удаленные строки перед текущей строкой

Редактор **nano**

Вызов:

$ nano <*имя_файла*>

Команда	Описание
Ctrl-X	закрыть редактор
Ctrl-O	сохранить
Ctrl-C	номер строки\текущая позиция
Ctrl-W	поиск
Ctrl-W затем Ctrl-T	переход к строке №
Ctrl-K	вырезать строку
Ctrl-U	вставить из буфера
Alt-A	выделение текста: нажмите Alt+A (или Ctrl+6), затем установите курсор в конец текста, который нужно вырезать; отмеченный текст при этом выделяется
Alt-6	копировать в буфер

Приложение 3. Основные команды: Module и SLURM

Список основных команд для работы с пакетом `Module`:

Команды Module	Описание
module avail	просмотреть список доступных модулей
module add <name of the module from the list>	загрузить модуль (добавить модуль в список подключенных)
module list	просмотреть список подключенных модулей
module rm <name of the module>	выгрузить модуль из списка подключенных

Список основных команд планировщика SLURM:

Команды SLURM	Описание
sinfo	команда для просмотра состояния вычислительных узлов и очередей SLURM
squeue	команда для просмотра списка запущенных задач
sbatch < name of the submission script >	команда для запуска задачи в режиме очереди: отправляет задачу для последующего выполнения (скрипт-файл обычно содержит одну или несколько команд **srun** для запуска параллельных задач)
scancel <ID-job>	команда для удаления задачи из очереди

Приложение 4. Операции редукции в OpenMP

Для использования в опции `reduction` доступны операции сложения, вычитания, умножения, логического «и», побитового исключающего «или», логического «или», операции «не», вычисления минимума и максимума, обозначаемые соответственно: `+`, `-`, `*`, `&`, `^`, `|`, `&&`, `||`, `min`, `max`.

Для каждой из перечисленных операций при входе в параллельную область инициализируется свое начальное значение переменной редукции `omp_red=omp_priv` (табл. П4). Это значение зависит от типа операции редукции и может отличаться от значения `omp_red=omp_in`, которое переменная редукции имеет перед входом в параллельную область.

Значение результата редукции формируется как результат `omp_out` указанной в опции `reduction` операции над значениями переменной редукции в разных нитях на выходе из параллельной области и над исходным значением `omp_in`, которое переменная имела до открытия параллельной области.

Таблица П4. **Доступные для редукции операции, их инициализирующие значения и результат выполняемых глобальных операций**

Операция редукции	Инициализация значения переменной редукции в параллельной области	Формирование результата редукции
`+`	`omp_priv=0`	`omp_out += omp_in`
`*`	`omp_priv=1`	`omp_out *= omp_in`
`-`	`omp_priv=0`	`omp_out -= omp_in`
`&`	`omp_priv=0`	`omp_out &= omp_in`
`|`	`omp_priv=0`	`omp_out |= omp_in`
`^`	`omp_priv=0`	`omp_out ^= omp_in`
`&&`	`omp_priv=1`	`omp_out && omp_in`
`||`	`omp_priv=0`	`omp_out || omp_in`
`max`	`omp_priv=` наименьшее значение редуцируемого типа данных	`if (omp_in>omp_out) omp_out = omp_in;`
`min`	`omp_priv=` наибольшее значение редуцируемого типа данных	`if (omp_in<omp_out) omp_out = omp_in;`

Учебное издание

Башашин Максим Викторович
Земляная Елена Валериевна
Стрельцова Оксана Ивановна

Основы технологии OpenMP на кластере HybriLIT

УЧЕБНОЕ ПОСОБИЕ

Редактор Ю. С. Цепилова
Технический редактор Ю. С. Цепилова
Компьютерная верстка Ю. С. Цепилова
Корректор Ю. С. Цепилова

Подписано в печать. Формат 60×84/16. Усл. печ. л. 2,90.
Тираж 24 экз. Заказ № 7.

ГБОУ ВО МО «Университет «Дубна»
141980, г. Дубна Московской обл., ул. Университетская, 19