Комментарии к лабораторной работе №5

Файл построен в виде нумерованных римскими цифрами заметок.

I В тексте описания данной лабораторной работы (http://ssd.sscc.ru/ru/content/opplabs/loadbalancing) под словами «компьютер» и «процессор» вам следует подразумевать MPI-процесс.

II Общий алгоритм работы параллельной программы можно представить так. Весь коллектив запущенных MPI-процессов:

1) изначально присваивает iterCounter=0;
2) берёт список заданий номер iterCounter;
3) для каждого задания i из списка вычисляется вес Tl[i].repeatNum;
4) выполняет задания данного списка;
5) синхронизируется (после синхронизации у всех процессов все задания из данного списка должны быть завершены);
6) iterCounter := iterCounter+1;
7) возврат к п.2.

III Обратите внимание на фразу «для экспериментов лучше задать некоторые осмысленные правила изменения загрузки на процессорах». И я рекомендую вам оставить это в программе не только для экспериментов, а так и сдавать. То есть Tl[i].repeatNum, который можно интерпретировать как вес задания i, сделать не рандомным, а, например, приблизительно следующим (почему «приблизительно» - поймете позже ϑ):

Tl[i].repeatNum = |50-i%100| * |rank-(iterCounter%size)| * L, i=0..size*100-1,
параметр L подберите сами.

IV После того, как процесс заканчивает обработку своей порции заданий из списка, он обращается к другим процессам (другому процессу) за новыми для себя заданиями. По какому протоколу процессы будут забирать задания – дело ваше. Можно брать несколько заданий от одного процесса, можно брать по одному заданию от нескольких процессов, можно по несколько заданий от нескольких процессов… Какие MPI-функции использовать для коммуникаций – опять-таки оставляю вам на откуп. В тексте описания сказано про операции точка-точка. Но можно, например, добавить коллективные операции. Допустим, освобождающийся процесс при помощи MPI_Bcast сообщает всем о том, что ему нужно передать работу (часть заданий). Затем происходит сверка количества

оставшихся заданий у процессов (при помощи Allgather или Allreduce) и тот процесс / те процессы, у кого оставшихся заданий больше всего, отправляет / отправляют освободившемуся процессу несколько своих.

V Обратите внимание на фразу «Считается, что этот вес нельзя узнать, пока задание не выполнено». То есть в Вашем протоколе перераспределения работы нельзя учитывать вес заданий (Tl[i].repeatNum), а можно только их оставшееся количество.

VI Вместо квадратного корня в сумме «globalRes += sqrt(i);» можно использовать sin(i), чтобы накопленная сумма не была слишком большой.

VII После каждой итерации iterCounter (после каждого списка задач) каждый MPI-процесс должен выводить:

В последней разности $T^p_k(1)$ – засечка времени, сделанная процессом *p* в самом начале выполнения итерации *k* . $T^p_k(2)$ – засечка времени, сделанная процессом *p* в конце выполнения итерации *k* сразу после того, как он закончил работать над заданиями и перестал брать задания у других. То есть после засечки $T^p_k(2)$ процесс только ждет, когда остальные закончат свои операции, чтобы вместе перейти к следующей итерации. Таким образом, T^p_k – время активной работы процесса *p* на данной итерации *k*.

Кроме того, после каждой итерации нужно посчитать время дисбаланса:
$\Delta_k = \max_{m,n} |T^m_k - T^n_k|$

и долю дисбаланса

$(\Delta_k / \max_p (T^p_k)) * 100\%$

Доля дисбаланса, выраженная в процентах, покажет вам насколько был плох или хорош ваш алгоритм балансировки.

VIII Количество поочередно обрабатываемых списков (iterCounter) сделайте таким, чтобы программа выполнялась не менее 30 сек. и списков должно быть не менее 3. Помните, что кроме количества списков вы можете управлять параметром L.

(Опционально. За выполнение +20%) IX Напишите последовательную программу. Посчитайте ускорение и эффективность для 2, 3, 4 процессов. Помните, что на каждом узле по 12 ядер. В каждом процессе по 3 потока, поэтому при количестве процессов, большем 4, потоки могут мешать друг другу. Для чистоты вычислительного эксперимента рекомендую:

1) использовать только 1 узел, чтобы не было коммуникаций по сети;

2) вне зависимости от того, сколько процессов вам нужно для эксперимента, забирать все ядра узла в задании для системы пакетной обработки, чтобы больше никто данным узлом не пользовался во время вашего эксперимента.

Последовательная программа, сравниваясь с которой вы будете вычислять ускорение и эффективность, делает следующее:

- запускает только 1 поток (то есть никакой работы с POSIX Threads нет);

- считает все задания подряд.

Обратите внимание на то, что количество и вес заданий должны быть одинаковы вне зависимости от количества процессов для корректного расчета ускорения и эффективности. Именно поэтому количество size*100 заданий в списке не пойдет. А также не пойдет рандомное распределение нагрузки.

Ускорение и эффективность вам покажет насколько эффективно вы сделали перераспределение работы и подгрузку новой порции без простоя основного потока, обрабатывающего задания. Здесь вам пригодится третий поток, о котором идет речь в задании. Наличие этого третьего потока не будет являться обязательным для сдачи лабораторной.

О POSIX THREADS

I POSIX Threads – инструмент программирования многопоточных приложений более низкого уровня, чем OpenMP. OpenMP хорошо подходит для распараллеливания тяжелых циклов – ситуаций, когда у потоков работа однородна. POSIX Threads чаще используют, когда нужно в пределах одного процесса запустить несколько потоков, выполняющих принципиально разную работу. В данной лабораторной – как раз такой случай.

II Объект mutex может принадлежать только одному потоку в одно время. При помощи объекта mutex можно реализовать критическую секцию. Делается это очень простым способом:

```
pthread_mutex_lock(&mutex);  //Захватываем объект mutex. Если он уже занят, то ждем.
work_in_critical_section();
pthread_mutex_unlock(&mutex);  //Освобождаем объект mutex.
```

Есть еще функция pthread_mutex_trylock, которая работает так же, как pthread_mutex_lock, только если мьютекс занят, то она тут же выходит, не дожидаясь освобождения.

В данной лабе вам может пригодиться мьютекс, например, для доступа к массиву заданий Tl. Ведь к этому массиву обращается поток, обрабатывающий задания и поток, отправляющий/принимающий задания от других.

III В нашем случае, естественно, при помощи pthread_create нужно порождать потоки с разными функциями, поскольку, в отличие от программы в примере, каждый

поток в пределах процесса занят своей работой, принципиально отличающейся от работы других потоков того же процесса.

ТРЕБОВАНИЯ

1) Программа не должна зависеть от числа процессов.
2) Использование коммуникаций MPI между процессами.
3) Использование POSIX Threads для порождения нитей в рамках процессов. Минимум в каждом процессе по 2 нити.
4) Вес заданий считать заранее неизвестным для процессов.