Лекция 7. Разработка параллельных программ. Проблемы балансировки загрузки.

1. Языки и средства параллельного программирования.

А) Языки программирования делятся на 3 группы:

1) стандартные последовательные языки, поддерживающие специальные библиотеки функций для организации параллельных вычислений (Fortran 77, Pascal, C/C++/C#);

2) специализированные языки, включающие операторы или директивы скалярной, векторной, матричной и параллельной обработки (Occam, Fortran 90/95, HPF, MPF, MPC, MC#, TC#);

3) шейдерные языки программирования для спецвычислителей (например, графических процессоров и ПЛИС'ов, CUDA, OpenCL и др.).

Б) Среды разработки параллельных программ: Intel Parallel Studio, CUDA Toolkit и NVIDIA Nsight, ROCm (AMD), MS Visual Studio, Eclipse, Qt Designer, Boost и др.

В) Системные средства ОС – каналы межпроцессорного обмена, сокеты, средства порождения конкурентных или параллельных процессов, средства создания и использования разделяемых ресурсов.

Г) Официальные стандарты организации параллельных программ: POSIX, PVM, MPI, OpenMP.

Модели, библиотеки и средства – ShMem (Cray, глобальная адресация памяти), NUMA (англ. Non-Uniform Memory Access – неравномерный доступ к памяти) и ccNUMA (когерентный кэш памяти), CUDA (Nvidia, программирование ГПУ), MapReduce (Google). ПО ИПМ для автоматизации распараллеливания – DVM (distributed virtual machine), DVMH (hybrid DVM), NORMA, SAPFOR.

Д) Элементы параллельного программирования – идеи, методы, примеры алгоритмов и программ, в том числе: контексты и нити, синхронные и асинхронные каналы обмена данными, виртуальные объекты (в том числе: виртуальные процессоры, память, коммуникации, топологии, ресурсы).

2. Общие направления разработки параллельных программ.

1) выбор оптимальной последовательности алгоритмов $\{A_k\}$, критерии:

а) макс. быстродействие (минимальное время решения для каждого реализуемого k),

б) макс. масштабируемость (чем больше данных, тем больше вычислителей можно взять),

в) универсальность и однородность кода ($\forall k, p: \quad A_k = A_p$).

2) определение объема данных: N ~ p или N >>p;

3) определение модели памяти МВС: распределенная, разделяемая, гибридная (удаленные и локальные процессы), иерархическая;

4) выбор платформы: аппаратные средства, язык программирования, ***коммуникационная*** среда;

5) определение способов ввода-вывода исходных, промежуточных и результирующих данных;

6) выбор каналов связи и топологии обменов;

7) применение виртуальных процессоров (вычислителей) и виртуальных топологий обменов;

8) отладочная информация (протоколирование) и способы тестирования.

9) решение проблемы балансировки загрузки процессоров;

3. Параллельный алгоритм и структура программы.

Каков параллельный алгоритм – одноэтапный (а) или многоэтапный (б):

а) $Q_{посл} = Q_0 + Q_1 + ... + Q_n \le Q_0 + n \cdot q, \quad Q_{пар} = Q_0 + \max\limits_{k=1..n} Q_k \le Q_0 + q;$

б) $Q_{посл} = \sum\limits_m Q^{(m)}_{посл}, \quad Q^{(m)}_{посл} = Q^{(m)}_0 + Q^{(m)}_1 + ... + Q^{(m)}_{n^{(m)}} \le Q^{(m)}_0 + n^{(m)} \cdot q^{(m)}, \quad Q_{пар} = \sum\limits_m Q^{(m)}_{пар}, \quad Q^{(m)}_{пар} \le Q^{(m)}_0 + q^{(m)};$

на каждом этапе своя степень параллелизма.

ПА должен быть наилучшим хотя бы по одному из указанных выше критериев.

Любая программа содержит три последовательные части:

1) ***инициализацию*** (настройка окружения, коммуникаций, ввода/вывода исходных данных);

2) ***расчетную часть*** (вычисления, обмены данными, ввод-вывод);

3) ***сохранение результатов и окончание работы***.

4. Выбор платформы.

Модель ВС – мультипроцессорная система, мультикомпьютерная система, смешанная система.

В каждом классе – однородная, неоднородная или гибридная архитектура.

Память: общая, распределенная, гибридная, иерархическая.

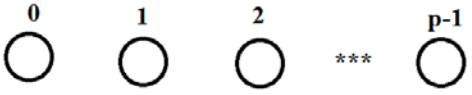
Проблема связи между различными типами памяти. Память ЦПУ, ГПУ. ПЛИСов.

Синхронизация и защита памяти.

Каналы связи - ***синхронные и асинхронные*** (последние бывают ***буферизованные и небуферизованные***). Топологии обменов бывают реальные и виртуальные. Пример: организация топологий в MPI.

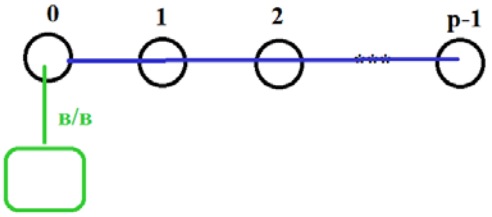
5. Ввод-вывод данных.

а) ввод-вывод с мастера (для малого числа процессоров);
б) ввод-вывод с каждого процессора (большие поля данных);
в) смешанная схема ввода-вывода (несколько мастеров).



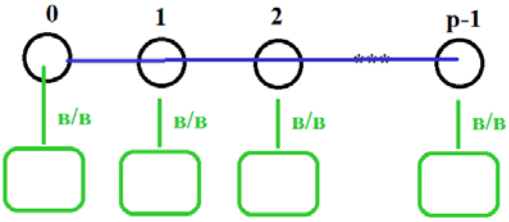
Данные: исходные
промежуточные
результирующие

а) ввод-вывод с мастера - универсальный, но неэффективный



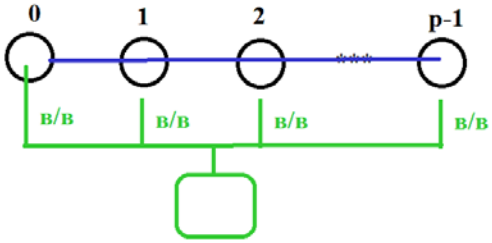
+ не нужно перепрограммировать посл. в/в
работа с FS не изменится
возможна реконфигурация вычислений
- недостаток ОЗУ на мастере
существенное замедление времени в/в

б) ввод-вывод с каждого



+ быстрый в/в
возможность отладки каждого процесса
- невозможноть доступа ко всем LFS
невозможность реконфигурации

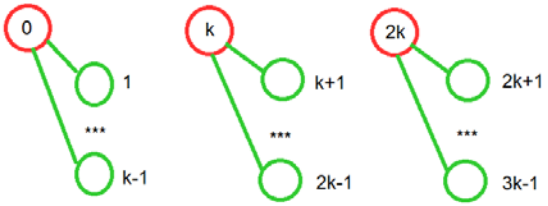
LFS - локальная
файловая система



+ есть доступ ко всем результатам
возможна реконфигурация
- необходимо перепрограммиров. в/в
ограничение на количество файлов
замедление в/в

NFS - сетевая
файловая система

в) смешанная схема - несколько мастеров



Системы хранения данных: LFS (локальные диски вычислителей), NFS (сетевая), PFS (параллельная).
Ввод-вывод с мастера – универсальный подход.
Применим даже там, где не все процессоры имеют доступ к диску.
Преимущество: единая мнемоника файловых имен (main.px, main.d, main.r, main.p, main.w, …), малое время записи данных на общий диск (например, из-за скорости сети).
Недостаток: если объем данных превышает размер ОЗУ одного процессора, то приходится организовывать последовательную схему записи и чтения.
Ввод-вывод с каждого процессора может осуществляться двумя способами:
1) запись-чтение всеми одного общего файла (конфликт по записи + дополнительное время определения своего фрагмента);
2) запись-чтение каждым своей группы файлов (более устойчивая схема, но нельзя изменить число процессоров при продолжении расчетов без доп. действий, сильно увеличивается нагрузка на файловую систему по количеству файлов и размеру директорий).

Смешанная схема ввода-вывода – несколько мастеров и несколько уровней специализации. Очень гибкая схема. В любом варианте необходимо обеспечивать масштабируемость.

6. Компиляция и запуск программ.

Компиляция:
1) прямое обращение к компилятору (cc, CC, gcc, icc, f77, ifc, …);
2) использование скриптов (mpicc, mpicxx, mpif77);
3) обращение через makefile: make –f main.mkc или nmake /f main.mak.
Компиляция на собственном или удаленном узле.
Запуск:
1) непосредственный: >main.px
2) с помощью среды: >mpirun –np 10 –machine hosts main.px

7. Отладка и протоколирование.

При отладке и в процессе массовых расчетов необходимо иметь контрольную информацию с каждого процессора, причем выводить ее не на экран, а в отдельный файл. Нужна система именования файлов протокола (main000.p, main001.p, …) и различная степень детализации информации (lp – параметр детализации информации). Основная проблема – емкость файловой системы => информация должна быть компактной и интегральной.

8. Проблемы балансировки загрузки.

Причины, приводящие к дисбалансу:
1) неоднородность алгоритма;
2) неоднородность ВС на аппаратном уровне;
3) неустойчивость работы АПС (процессоры, каналы связи, ОС и т.д.).
Что делать? Ответ: 1) установить наличие и причину дисбаланса; 2) устранить или ослабить дисбаланс. Для реализации п. 1) необходимо иметь механизм замера времени выполнения одинаковых фрагментов кода. Для реализации п. 2) необходимо использовать один из способов балансировки загрузки.
Виды балансировки: статическая и динамическая (централизованный и децентрализованный алгоритм).
Пример. Обработка данных в продолжительном цикле. На каждом шаге цикла с помощью p вычислителей обрабатывается N данных. Обозначим $t_1, \ldots, t_p$ – времена расчета шага цикла на соответствующем вычислителе, $t_s = (t_1+\ldots+t_p)/p$ – среднее время на шаг цикла, $\sigma=max|(t_k-t_s)/t_s|*100\%$ – дисперсия (разброс) времен. Задача балансировки состоит в минимизация дисперсии.
Решение с применением статической балансировки загрузки.
Можно заранее узнать производительности вычислителей $q_1, \ldots, q_p$. Тогда на первом шаге цикла можно распределить N данных в соответствии с производительностью вычислителей:
$N_1/N_2 = q_1/q_2, \ldots, N_{p-1}/N_p = q_{p-1}/q_p, N_1+\ldots+N_p = N \Rightarrow N_1=N*q_1/(q_1+\ldots+q_p), \ldots, N_p=N*q_p/(q_1+\ldots+q_p)$.
Это распределение оставить неизменным на последующих шагах цикла.
Решение с применением динамической балансировки загрузки.
Если информация заранее неизвестна или ситуация меняется, то производительности вычислителей можно определять на каждом шаге цикла: $q_1=N_1/t_1, \ldots, q_p=N_p/t_p$. Затем определять новые значения $N_1, \ldots, N_p$ для следующего шага цикла. При этом используются два алгоритма:

а) точный алгоритм: $N_k^{'} = Nq_k / \left(q_1 + \ldots + q_p\right)$;

б) приближенный диффузный алгоритм: $N_k^{'} = N_k + \tau Nq_k / \left(q_1 + \ldots + q_p\right), \ \tau \sim 0.05$.