МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Распределённые системы"

ОТЧЕТ

о выполненном задании

студента 424 группы факультета ВМК МГУ

Яндиева Абдуллаха Ахметгиреевича

Москва, 2022 г.

Задание 1

Программа моделирует выполнение операции редукции `MPI_MAXLOC`.

Максимум вычисляется по следующей формуле:

$$max\left(\binom{value_1}{rank_1}, \binom{value_2}{rank_2}\right) = \binom{value}{rank},$$

где

$$value = \max\left(value_1, value_2\right)$$

$$rank = \begin{cases} rank_1, & if\ value_1 > value_2 \\ \min(rank_1, rank_2), & if\ value_1 = value_2 \\ rank_2, & if\ value_1 < value_2 \end{cases}$$

На входе мы имеем транспьютерную матрицу размером 4*4, в каждом узле которой находится один процесс со своим определенным значением: 0 процесс содержит значение матрицы в позиции (0, 0), 1 процесс - (0, 1), …, 16 процесс - (3, 3). Стоит отметить, что вместе со значением матрицы также хранится и номер процесса, то есть все значения хранятся и передаются парами: (Число, Номер процесса).

Описание алгоритма

Шаг 1

Сначала процессы с четными номерами передают свои значения процессам с нечетными: 0 процесс передает 1 процессу, 2 процесс - 3 процессу и т. д. Нечетные процессы в свою очередь вычисляют максимум (по формуле выше) своего и переданного от соседа значений. Полученное значение считается новым значением процесса.

Шаг 2

Затем аналогичная передача происходит среди нечетных процессов (которые уже обновили свои значения на 1 шаге): 1 процесс передает новое значение 3 процессу, 5 процесс - 7 процессу и т.д. Процессы, которые получили новые значения (а именно 3, 7, 11, 15 процессы) будем называть центральными. Центральные процессы в свою очередь вычисляют максимум своего и переданного от нечетного соседа значений. Полученное значение считается новым значением соответствующего процесса.

Шаг 3

Далее происходит обмен между 3–7 и 11–15 процессами. Соответственно, 3 и 7 процессы вычисляют максимумы 3 и 7 процессов, 11 и 15 - своих процессов. Полученные значения считаются новыми значениями центральных процессов.

Шаг 4

Затем уже происходит обмен между 3–11 и 7–15 процессами. Заметим, что в 3 и 7 процессах содержится одно и то же значение максимума левой половины процессов, а в 11 и 15 - максимума правой половины процессов. Таким образом, после данных обменов и вычисления максимумов все четыре центральных процесса будут содержать искомое значение максимума всей матрицы.

Шаг 5

Осталось передать полученные значения остальным процессам. На данном шаге каждый из центральных процессов передает свое значение соседу слева: 3 процесс - 2 процессу, 7 процесс - 6 процессу и т. д. Эти процессы переопределяют свое значение максимумом, полученным от соседа справа. Итого 8 процессов содержат искомый максимум: 2, 3, 6, 7, 10, 11, 14, 15 процессы.

Шаг 6

Наконец, каждый из восьми процессов (2, 3, 6, 7, 10, 11, 14, 15) передает свое значение процессу с номером на 2 меньше, чем свой: 2 процесс - 0 процессу, 3 процесс - 1 процессу, 6 процесс - 4 процессу и т. д. Соответственно, другие 8 процессов переопределяют свое значение полученным значением максимума.

Таким образом, все 16 процессов содержат максимальное значение матрицы, где максимум считается по формуле выше.

Временная оценка работы алгоритма

Для оценки времени работы алгоритма нужно посчитать количество пересылок данных. Если внимательно приглядеться, то можно заметить, что каждый шаг алгоритма содержит в себе ровно одну пересылку. На 1, 2, 5, 6 шагах это односторонняя пересылка, а на 3 и 4 шагах - двусторонняя пересылка, но при этом процессы работают параллельно: отправляют свои значения и ожидают получения нового числа. Хотя и используются блокирующие операции, гарантируется, что блокировок происходить не будет за счет того, что каждый процесс сначала отправляет свое число, а лишь потом встает в ожидание получения нового значения.

Таким образом, имеем следующую формулу для оценки:

$$T = 6 * (Ts + \texttt{sizeof(int)} * Tb)$$

Так как `sizeof(int)` = 4, Ts = 100, Tb = 1, имеем:

$$T = 6 * (100 + 4 * 1) = 624$$

Задание 2

Доработана MPI-программа для алгоритма Sor2D, реализованная в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Добавлены контрольные точки для продолжения работы программы в случае сбоя. Для реализации был выбран следующих сценарий при сбое: **вместо процессов, вышедших из строя, создаются новые MPI-процессы, которые используются для продолжения расчетов.**

Описание алгоритма

Так как описание MPI-программы для алгоритма Sor2D было дано в рамках отчета по курсу “Суперкомпьютеры и параллельная обработка данных”, в данном отчете представлено лишь описание дополнений к этой программе - обработки случаев со сбоем во время работы программы. Для примера реализации была взята программа по ссылке из одной из лекций.

Для начала мы определяем переменную `allowed_to_kill` и присваиваем ей значение `1`. Это вспомогательная переменная, которая показывает, можно ли убить процесс с номером `1`, для демонстрации работы алгоритма. В случае, если мы уже убили некоторый процесс, то при работе с новым коммуникатором мы зануляем переменную `allowed_to_kill`. И после непосредственно уничтожения процесса (`raise(SIGKILL)`) мы также зануляем переменную `allowed_to_kill`.

Далее мы определяем переменную `do_recover` и в начале присваиваем ей значение `0`. Эта переменная показывает, необходимо ли восстановить работу системы. Новое значение переменная получает в случае, если произошел прыжок на метку. Помимо переменной `do_recover`, индикатором необходимости восстановления является и тот факт, что коммуникатор `parent` не является нулевым (`MPI_COMM_NULL`). Для получения родителя используется MPI-процедура `MPI_Comm_get_parent`. После инициализации значения `parent` происходит проверка, является ли `parent` нулевым. Если является, значит, коммуникатор у нас искомый, и мы при помощи `MPI_Comm_dup` дублируем его. Иначе, нам требуется исправить работу системы. Для этого вызывается фунция `app_needs_repair` от нулевого коммуникатора.

Функция `app_needs_repair` возвращает `true`, если приложению необходимо переделать некоторые итерации алгоритма, и возвращает `false` в противном случае. В этой функции мы проверяем, является ли это первым случаем, когда мы увидели ошибку для данного коммуникатора. Если так, то мы меняем коммуникатор `world`, определенный в начале программы в виде макроса, иначе ничего не делаем и возвращаем `false`. Мы храним коммуникатор, чтобы обработчик ошибок оставался подключенным до тех пор, пока пользователь не завершит все ожидающие операции. Ожидается, что пользователь завершит все операции для данного коммуникатора, прежде чем выполнять новые операции в новом коммуникаторе. Если пользователь не выполнит все операции до смены коммуникатора, а обработчик вызовется в новом коммуникаторе, старый коммуникатор может быть освобожден, пока еще не завершены все операции, и может возникнуть фатальная ошибка, когда эти операции будут окончательно завершены. Освобождение осуществляется при помощи вызова `MPI_Comm_free`, а замена коммуникатора - при помощи `MPIX_Comm_replace` (описание см. ниже). Далее внутри этой функции вызывается `app_reload_ckpt`, которая перезапускает контрольную точку: возобновляет нужную итерацию, предотвращая дальнейшее появление ошибок.

MPIX_Comm_replace принимает на вход коммуникатор и возвращает (в качестве 2 параметра) указатель на новый коммуникатор. Сначала идет проверка на то, является ли текущий процесс новым. Если это так, то этот процесс ожидает присвоения нового значения ранга, которое будет отправлено из старого коммуникатора процессом с номером 0. Соответственно определяется родительский коммуникатор и при помощи MPI_Recv, процесс получает новое значение ранга. Если же это не новый процесс, то создается определенное количество процессов, которые нужно заменить. Сперва обновляется коммуникатор (создается новый), чтобы он не содержал в себе мертвые процессы. Для этого вызывается процедура MPIX_Comm_shrink. Определяется размер полученного коммуникатора, и из общего числа процессов вычитается это число, чтобы получить количество мертвых процессов. Если мертвых процессов не найдено, то освобождаем новый коммуникатор при помощи MPI_Comm_free, и дальше работаем со старым. Если же сбой произошел, то устанавливаем на новый коммуникатор обработчик ошибок по умолчанию (MPI_ERRORS_RETURN).

Далее при помощи функции MPI_Comm_spawn создаем нужное количество новых процессов (количество было посчитано ранее). В качество первого параметра (команды для выполнения) передаем нашу искомую программу argv[0]. Сравниваем результат работы данной функции с MPI_SUCCESS, тем самым проверяем успешность порождения нужного числа новых процессов. С полученным значением флага вызываем MPIX_Comm_agree, чтобы привести в согласование значение флага и группу неудачных процессов в коммуникаторе. Если мы не получили MPI_SUCCESS, то делаем MPIX_Comm_revoke, которая останавливает все взаимодействия между процессами, связанные с заданным коммуникатором, освобождаем новый коммуникатор и прыгаем на метку redo для повторного порождения недостающего числа процессов.

После того, как мы все-таки получили MPI_SUCCESS после MPI_Comm_spawn, мы вычисляем ранг процесса в старом и новом окружении, и процесс с рангом 0 при помощи операций работы с группами процессов MPI, высылает всем новым процессам их новые ранги. Для определения группы процессов по коммуникатору используется MPI_Comm_group. Для получения группы мертвых процессов используется MPI_Group_difference, которая вычисляет группу процессов, которые присутствуют в старом коммуникаторе (первом параметре), но отсутствуют в новом коммуникаторе (втором параметре). Для получения нужного номера ранга (ранга мертвого процесса для установления соответствия с новым процессом) используется MPI_Group_translate_ranks. В цикле эти номера отправляются всем новым процессам. Промежуточно использованные группы освобождаются при помощи MPI_Group_free.

Затем мы объединяем коммуникатор, полученный в результате вызова MPI_Comm_spawn для восстановления исходного коммуникатора. Для этого делаем вызов MPI_Intercomm_merge с одинаковым приоритетом. В случае, если нам вернулся плохой флаг, мы снова возвращаемся к метке redo, чтобы заново породить процессы (перед этим, конечно же, освободили новый коммуникатор).

После того, как все отработало успешно, остается лишь переупорядочить ранги в новом коммуникаторе в порядке нахождения процессов в исходном коммуникаторе. Для этого используется MPI_Comm_split, который удаляет лишние процессы и переупорядочивает ранги, чтобы все выжившие процессы оставались на своих прежних местах. Мы проверяем, что все отработало правильно, и если это не так, то мы снова прыгаем на redo. После того, как мы преодолели этот шаг, остается лишь взять обработчик ошибок из старого коммуникатора и положить в новый.

Реализация MPIX_Comm_replace завершена.

Затем мы создаем обработчик ошибок `errhandler_respawn` при помощи вызова `MPI_Comm_create_errhandler`. Сначала мы определяем класс ошибки при помощи функции `MPI_Error_class`. Если это какая-та сторонняя ошибка, не связанная с уничтожением процесса и вызовом операции `MPIX_Comm_revoke` (`MPIX_ERR_PROC_FAILED` и `MPIX_ERR_REVOKED` соответственно), то мы при помощи вызова `MPI_Abort` досрочно завершаем работу MPI-программы с кодом соответствующей ошибки. Иначе мы вызываем `MPIX_Comm_revoke`. А дальше вызываем уже известную функцию `app_needs_repair`, только уже не от `MPI_COMM_NULL`, а от коммуникатора `world`.

Дальше производятся вычисления (здесь никаких изменений по сравнению с прошлогодней программой нет). После того, как все процессы вычислили свои значения, мы смотрим на значение `allowed_to_kill` и на одной из итераций убиваем процесс с номером `1`. Встаем на барьерную синхронизацию и ждем пока все процессы завершат свою работу. После вычислений мы проверяем не достиг ли номер итераций контрольной точки (сверяем `iteration` с заранее установленным значением `CKPT_STEP`). Если время контрольной точки пришло, то удостоверяемся в целостности системы. Далее мы натыкаемся на метку `do_sor`. Здесь мы копируем либо исходные данные, если не пришлось восстанавливать систему, либо данные, полученные после обнаружения сбоя.

Таким образом и происходит обработка случая, когда некоторые процессы уничтожаются во время работы MPI-программы.