Использование многопоточных процессов в среде ParJava

М.С. Акопян <manuk@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В работе описывается подход, применяемый в ParJava по разработке многопроцессно-многопоточных (МПМП) программ. Разработан API и поддерживающая его библиотека, которая позволяет писать параллельные МПМП приложения на языке Java оставаясь в рамках стандарта MPI. Использование потоков в программе позволяет лучше утилизировать ресурсы многоядерного процессора. В рамках работы реализована МПМП программа быстрого преобразования Фурье на языке Java. Проведенные эксперименты показали, что МПМП программа работает быстрее, чем многопроцессная программа.

Ключевые слова: параллельные вычисления; многоядерностые процессоры; параллельные по данным программы MPI, многопоточные Java программы, многопроцессно-многопоточные программы.

DOI: 10.15514/ISPRAS-2015-27(2)-1

Для цитирования: Акопян М.С. Использование многопоточных процессов в среде ParJava. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 5-23. DOI: 10.15514/ISPRAS-2015-27(2)-1.

1. Введение

В настоящее время процессоры, предлагаемые на рынке, базируются в основном на многоядерной архитектуре. Это повышает продуктивность процессоров - уменьшая стоимость процессора (по сравнению с многопроцессорной системой с аналогичным количеством выполняемых модулей) и повышая его производительность. Так же это дает прикладным программистам возможность использовать преимущество общей памяти.

В многопроцессной программе каждый процесс имеет свою область памяти, и если одному процессу при вычислении необходимы данные соседнего процесса, то необходимо провести обмен данными между процессами. Однако если в рамках каждого процесса использовать несколько потоков, которые имеют доступ к общим данным процесса, то это позволит уменьшить накладные расходы по обращению к данным программы. Преимущество общей памяти сопровождаются ее недостатками - необходимы меры по

синхронизации и своевременному доступу к общим данным, во избежание состояний гонок.

При использовании многоядерных процессоров на узле кластера для разработки параллельных программ можно использовать следующие подходы:

1. Многопроцессная программа. nx1 – на узле запускается n процессов по одному потоку в каждом.
2. МПМП программа. 1xn – на узле запускается один процесс, в котором используются n потоков.

Когда в процессе используются много потоков, то внутри каждого процесса обычно производится приватизация с целью уменьшения критических секций. Однако в случае отсутствия критических секций приватизация не нужна (как в случае с МПМП версией БПФ).

В данной статье рассматривается подход, применяемый в ParJava [1] при разработке многопроцессно-многопоточных (МПМП) программ. Среда программирования ParJava позволяет разрабатывать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI. Был разработан API [2] и поддерживающая его библиотека, которая позволяет разрабатывать многопоточные MPI программы на основе стандартного библиотеки java.util.concurrent [3].

В статье приводится описание МПМП программы быстрого преобразования Фурье на языке Java. Проведены серии экспериментов на вычислительном кластере. Представлены графики сравнения многопроцессной и МПМП версий программы.

2. Модель выполнения параллельной многопоточной программы.

Среда программирования ParJava позволяет разрабатывать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI. Коммуникационная библиотека mpiJava.mpi [4] реализующая MPI основана на библиотеке mpiJava[5], который представляет собой привязку языка Java через интерфейс JNI к существующей реализации MPI (MPICH,LAM …). В библиотеке реализованы оберточные функции для стандарта MPI1.1 [6].

Как показывает опыт, использование потоков в рамках одного узла на современных кластерах с многоядерными процессорами может увеличить производительность параллельной программы за счет использования общей памяти и уменьшения накладных расходов.

Многопоточное программирование посредством низкоуровневого интерфейса java.lang.Thread [7] является устаревшим и не считается лучшим решением в достижении параллелизма из-за проблем с производительностью. Начиная с версии 1.5 в Java введена библиотека java.util.concurrent предоставляющая высокоуровневый интерфейс к потокам в Java. Разработанная библиотека

mpi.threads использует и расширяет эффективные инструменты предоставляемые библиотекой java.util.concurrent.

Однако библиотека java.util.concurrent была разработана не для высокопроизводительных вычислений, поэтому в реализации mpi.threads некоторые методы оригинальной библиотеки были закрыты, некоторые адаптированы для применения в рамках модели SPMD. Библиотека mpi.threads не является реализацией OpenMP [8], но многие подходы оттуда используются при работе с библиотекой: модель выполнения потоков, распараллеливаемые инструкции оформляются в виде пользовательских заданий, локальные в рамках задания переменные, разделяемые переменные, операция редукции при формировании распараллеливаемого фрагмента, критические секции, атомарные конструкции, барьеры и.т.д.

Модель выполнения потоков в библиотеке mpi.threads аналогична модели выполнения потоков в OpenMP. На рис. 1 приведена модель выполнения МПМП Java–программы, где N – количество процессов программы (на каждом узле запускается один процесс), n – количество ядер на каждом узле. Рассмотрим поведение параллельной многопоточной программы в рамках одного MPI процесса в среде ParJava.

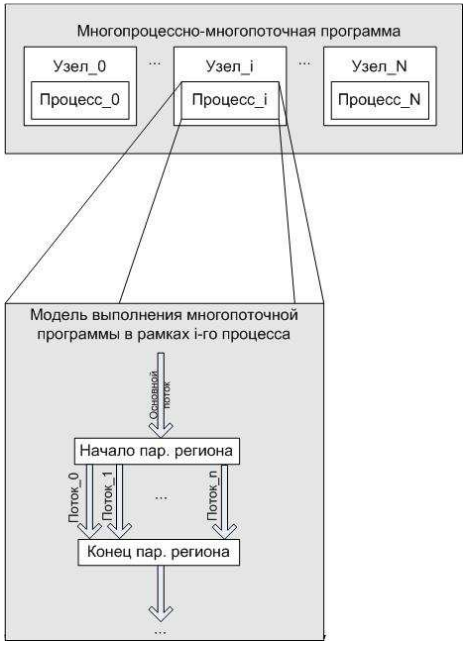

Рис. 1. Модель выполнения МПМП Java–программы.

В отличие от MPI и PGAS [9] моделей, где изначально работают N

В отличие от MPI и PGAS [9] моделей, где изначально работают N вычислительных модулей (процессы в случае MPI и потоки в случае PGAS), при применении библиотеки mpi.threads в начале программы выполняется один поток (основной поток), который последовательно выполняет инструкции программы. В определенных точках программы, где необходимо выполнить параллельно на нескольких потоках некий набор инструкций Ins (параллельный регион), пользователь формирует задания, содержащие инструкции Ins, и передает ссылки на задании библиотеке времени выполнения. Полученные пользовательские задания выполняются каждый в отдельном потоке. Для обеспечения функциональности барьера после формирования заданий пользователь должен явно вызвать соответствующую библиотечную функцию. В результате основной поток блокируется и ждет завершения выполнения сформированных ранее заданий выполняющихся в параллельных потоках. Синхронизация между потоками производится посредством критических секций, семафоров, атомарными операциями над переменными и.т.д.

В целях повышения производительности библиотека mpi.threads предоставляет возможность пользователю в рамках MPI процесса создавать пул потоков. При создании пула создаются и инициализируются фиксированное количество рабочих потоков. После чего рабочие потоки блокируются в ожидании пользовательских заданий. Также в пуле создается очередь заданий. Распараллеливаемый фрагмент последовательной программы должен быть преобразован (заменен) в группу заданий. Во время выполнения программы пользователь формирует новые задания и передает ссылки на них в очередь заданий, откуда задание передается первому свободному потоку, в котором и выполняется данное задание. Использование пула потоков позволяет избежать накладных расходов при создании и запуска новых потоков в ходе выполнения программы.

Пользовательское задание в ParJava представляет собой объект пользовательского класса наследника от системного класса mpiJava.threads.PJTask. В пользовательском классе должен быть реализован метод run(), содержащий исходный код задания. Обычно метод run() содержит цикл (гнездо циклов) исходной программы, который нужно распараллелить между потоками. Параметры задания устанавливаются с помощью метода setParams(…). При распараллеливании гнезда циклов пользователь должен распределить итерации цикла по заданиям с помощью параметров задания setParams(loop_start,loop_end,…) (ниже будет приведен пример распараллеливания цикла).

При работе с параллельными потоками используется родная модель памяти Java. Вся память, выделяемая в рамках задания, доступна потоку, который выполняет это задание. При формировании новых заданий память основного потока не наследуется автоматически. При необходимости использования переменных (или значений переменных) основного потока в задании,

пользователь при формировании задания должен передать в метод setParams(…) задания формальную переменную по ссылке (или по значению). При передаче переменных по ссылке, они становятся разделяемыми (shared) переменными и доступ к таким переменным в избегании состязаний (race condition) нужно осуществлять осторожно (необходимо пользоваться семафорами, критическими секциями, если хотя бы один из обращений к данным переменным является операцией записи).

Пример. Приведем пример (см. рис.2) использования потоков посредством

```
//инициализация массива A
…
//инициализация массива B
…
for(i=1; i <N; ++i)
{
 //тело цикла
 A[i] = B[i] + …;
 …
}
//печать массива A
…
```

Рис. 2.Последовательная версия исходного кода

библиотеки mpi.threads при распараллеливании гнезда циклов. Пусть имеется гнездо циклов с независимыми итерациями, которое нужно выполнить параллельно в нескольких потоках. Пусть количество потоков в пуле равно M. Итерации цикла распределяются равномерно по заданиям. В этом случае пространство итераций цикла [1,N] разбивается на M равных частей и формируется группа из M заданий (задание представляет собой диапазон итераций основного цикла, которые будут выполняться в одном потоке). На рис.3 приведен исходный код задания.

```
class LoopTask extends PJTask{
private int loop_start,loop_end;
private double[] refA;
private double[] refB;
```



```
public void setParams(int loopStart, int loopEnd,double[] A,double[] B)
{
   loop_start = loopStart;
   loop_end = loopEnd;
   refA = A;
   refB = B;
}
public void run()
{
 for(i= loop_start; i < loop_end; ++i)
 {
  //тело цикла
  A[i] = B[i] + …;
    …
 }
}
}
```

Рис.3 Исходный код задания

Задания распределяются между свободными потоками из пула, после чего основной поток блокируется и ожидает окончания выполнения M заданий. На рис.4 приведен фрагмент исходного кода основного потока многопоточной программы.

```
//создание пула потоков (tPool) с M потоками.
…
LoopTask[] loopTask;
//инициализация массива A
…
//инициализация массива B
…
int remaining = N%M;
int chunk = N/M;
int start = 1;
int end = start+(chunk-1) + (remaining >0?1:0);
for(int j = 0; j < M; ++j)
{
 loopTask[j].setParams(start,end,A,B);
 tPool.setTask(loopTask[j]);
 start = end + 1;
 end = start + (chunk-1) + (--remaining >0?1:0);
```

```
}
//Основной поток блокируется в ожидании выполнения заданий
tPool.waitForAll();
//печать массива A
…
```

Рис 4. Исходный код распараллеленной программы.

3. Результаты численных расчетов

Рассмотрим параллельную программу быстрого преобразования Фурье FT из набора NAS Parallel Benchmarks [10,11]. Первоначальная версия NPB на языке Java был разработан и реализован в университете Coruna [12].

В рамках работы проведенной в данной статье, был реализован многопоточный вариант программы FT: коммуникации между процессами программы обеспечиваются функциями MPI, а для взаимодействия между потоками в каждом процессе используются функции из библиотеки времени выполнения mpiJava.threads. Проведена серия экспериментов по сравнению параллельной программы основанной только на библиотеке MPI и параллельной программы на основе MPI и потоков Java.

Приведем описание параллельной программы FT. В данной статье рассматривается 3D версия программы FT (расчетная область представляет собой трех мерную матрицу комплексных чисел). Пусть имеется последовательность $u=\{u_0, u_1,\dots u_{N-1}\}$. Дискретное преобразование Фурье (задача 1D FFT) преобразует последовательность u в другую последовательность U, где

$$U_k = \sum_{n=0}^{N-1} u_n e^{\frac{-2\pi i k n}{N}}$$

В реализованном алгоритме FT дискретное преобразование Фурье применяется на 3D сетке размерностью LxMxN. В этом случае формула преобразование Фурье принимает следующий вид:

$$F_{q,r,s}(u) = \sum_{l=0}^{L-1}\sum_{k=0}^{M-1}\sum_{j=0}^{N-1} u_{j,k,l} e^{\frac{-2\pi i j q}{L}} e^{\frac{-2\pi i k r}{M}} e^{\frac{-2\pi i l s}{N}}$$

Данная задача вычисляется на высокопроизводительной вычислительной платформе с распределенной памятью. Параллельную программу FT можно запустить как с использованием 1D декомпозиции, так и 2D декомпозиции данных.

В задаче 3D FFT при применении 1D декомпозиции пространство данных разбивается на слои по направлению оси x– каждому процессу отдается один

слой (см. рис. 5). В рамках каждого процесса имеются все локальные данные для применения 1D FFT по направлениям y и z.

В каждом процессе параллельные вычисления производится в 4 этапа

1. Вдоль направления y применяется 1D FFT.
2. Вдоль направления z применяется 1D FFT.
3. Производится глобальное транспонирование по направлению x, что представляет собой коллективную коммуникацию AllToAll между всеми процессами. В результате у всех процессов появляются данные для счета вдоль оси x.
4. Вдоль направления x применяется 1D FFT.

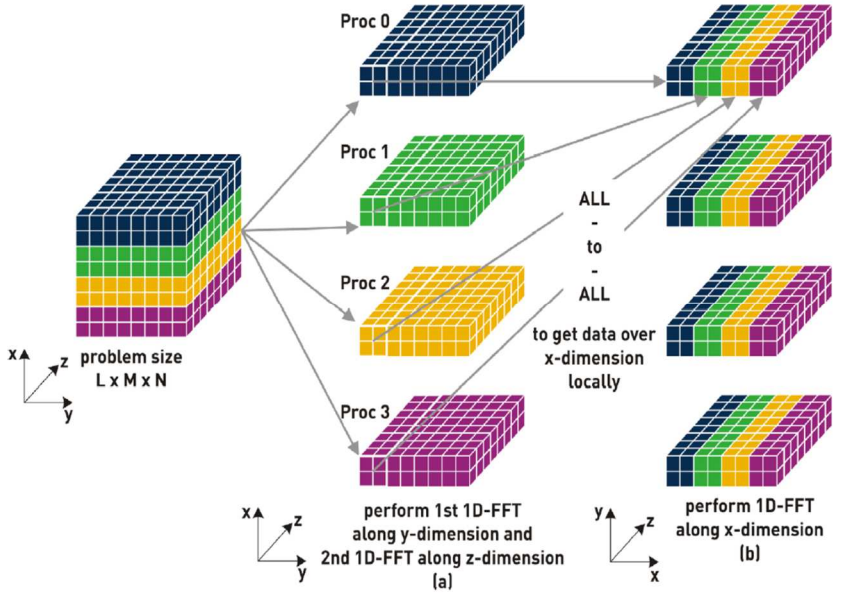

Рис. 5. 1D декомпозиция в решении задачи 3D FFT

При использовании 1D декомпозиции используется только одно глобальное транспонирование для получения в локальную память процесса данных необходимых для счета. Недостатком 1D декомпозиции является ограничение масштабируемости (максимального параллелизма) размером наибольшей длины 3D сетки данных. Однако степень параллелизма можно увеличить количеством ядер на каждом узле кластера при использовании МПМП версии алгоритма.

В задаче 3D FFT при применении 2D декомпозиции пространство данных разбивается как по оси x так и по оси z (см. рис. 6). В рамках каждого процесса имеются все локальные данные для применения 1D FFT только по

направлению оси y. Для применения 1D FFT по осям z и x необходимо провести транспонирование матрицы.

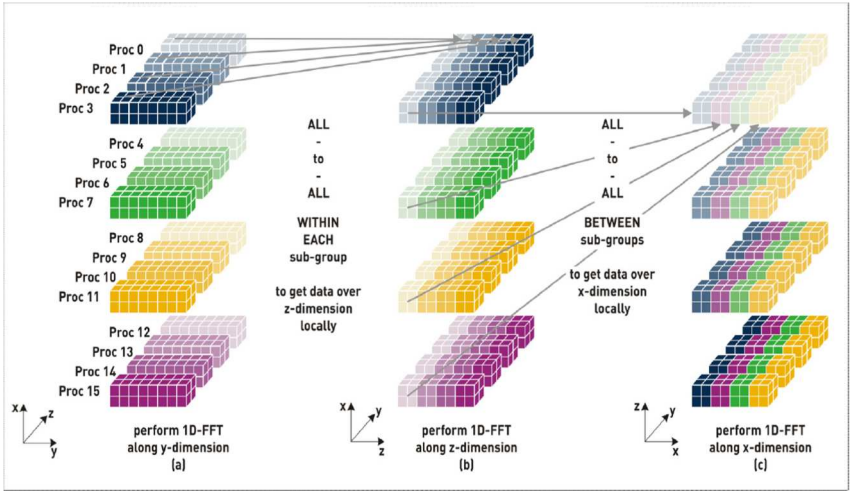

Рис. 6. 2D декомпозиция в решении задачи 3D FFT

В случае 2D декомпозиции вычисления производятся в 5 этапов

1. Вдоль направления y применяется 1D FFT
2. Глобальное транспонирование по оси z
3. Вдоль направления z применяется 1D FFT
4. Глобальное транспонирование по оси x
5. Вдоль направления x применяется 1D FFT

Пусть исходные данные представляют собой куб N^3. В этом случае при применении 1D декомпозиция параллельная программа масштабируется O(N), а при 2D декомпозиция $O(N^2)$.

Программа быстрого преобразования FT тестировалась на кластере МВС-100К (результаты см. рисунок XXX1), на 1410-ти 4-ядерных процессора Intel(R) Xeon(R) CPU X5365 с частотой 3.00GHz (два процессора на каждом узле), с интерфейсной платой HP Mezzanine Infiniband 4x DDR и 8Gb памяти на каждом узле.

FT_T представляет собой параллельную программу быстрого преобразования Фурье с использованием трех мерной расчетной матрицы, где в качестве коммуникаций используется интерфейс MPI, а также в рамках одного процесса используются несколько потоков Java. Рассмотрим некоторые особенности параллельного приложения FT_T. В программе FT_T применяется 1D декомпозиция. На каждом узле кластера запускается по одному процессу, а в рамках каждого процесса в определенных точках программы используются TSize потоков. На каждом узле кластера МВС-100К

имеется 8 ядер и количество используемых потоков должно быть меньше или равно 8-ми во избежание просадки производительности параллельной программы. При проведении исследований использовались два варианта максимального количества потоков TSize={4,8}.

Как уже было отмечено ранее, в целях повышения производительности для управления потоками используется пул потоков. В каждом процессе параллельной программы при инициализации данных создается пул потоков с максимальным количеством потоков TSize. После чего рабочие потоки блокируются в ожидании пользовательских заданий.

Профилирование программы показало, что большая часть времени работы программа проводит в функциях вычисления одномерного 1D FFT, а также в функциях по транспонированию трехмерной матрицы по разным осям. Было выделено шесть функций, которые необходимо распараллелить в рамках процесса. В результате были созданы шесть классов пользовательских заданий:

- два класса для 1D FFT по осям z и y (по оси x используется тот же класс, что и для оси z),
- три класса для локального транспонирования в рамках одного процесса (узла),
- один класс расчетного характера.

При достижении этих функций процесс формирует TSize заданий, инициализирует необходимые начальные параметры и помещает ссылки на сформированные задания в очередь заданий, откуда задания передаются первому свободному потоку, в котором и выполняется данное задание.

В результате распараллеливается расчетная часть – локальное транспонирование в рамках одного процесса (узла) по осям z и y. Транспонирование по оси x производится с применением коллективной коммуникационной функции AlltoAll, которое в данной реализации вызывается в каждом процессе (не в потоке). Поэтому в описываемый работе распараллеливание транспонирования по оси x не проводилось.

В оригинальной версии FT на языке Java использовались локальные массивы, которые аллоцировались каждый раз при вызове функций. Профилирование показало, что некоторые функции, в которых использовались такие локальные массивы, вызываются от десяти до сотен тысяч раз. В результате память набивалась устарелыми данными очень быстро, и проводились многократные сборки мусора, что увеличивало время работы программы. После проведения оптимизаций, большая часть таких массивов были заменены на глобальные. Это позволило снизить количество сборок мусора до одного, что привело к улучшению производительности параллельного приложения.

В приведенных ниже графиках исследуется масштабируемость по Амдалью, то есть фиксируется объем рассчитываемой матрицы и исследуется время выполнения параллельной программы на разных количествах используемых

ядер. Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче примерно 5Gb, количество внешних итераций 20. В тестах использовалась 64-разрядная версия Java.

В приведенных ниже рисунках FT представляет собой параллельное приложение БПФ на языке Java с применением интерфейса MPI. В данном случае рассматривается параллельная многопроцессная программа без применения многопоточности. FT запускается с применением 2D декомпозиции: для запуска с N(8,16,32,64) ядрами применяется декартовое разбиение пространства процессов [N1,N2], где N=N1*N2 и для запуска используется N1 узлов, на каждом узле N2 процессов. FT_T представляет собой МПМП программу и является собой модификацией FT. В данном случае применяется 1D декомпозиция: для запуска с N(8,16,32,64) ядрами используется линейное пространство процессов [N1], где N1 количество узлов и на каждом узле в определенных точках программы запускаются TSize=N2 потоков (N=N1*N2).

На рис. 7 представлены графики зависимости времени выполнения параллельной программы БПФ от числа используемых ядер.

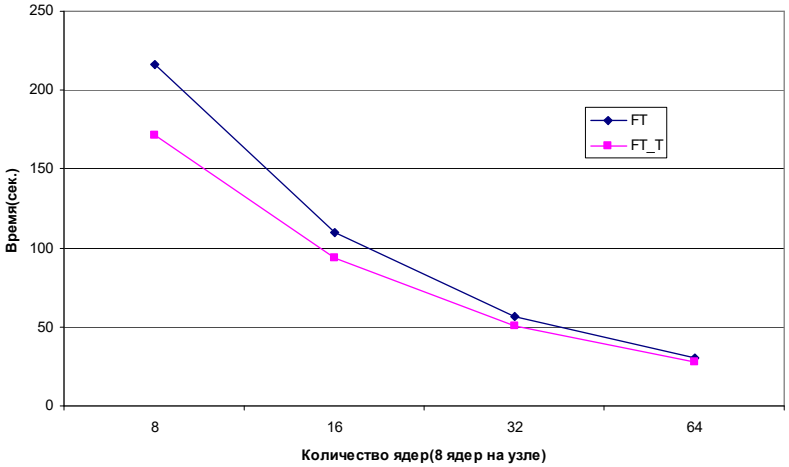

Рис. 7. Сравнение времени выполнения многопроцессной программы БПФ с МПМП версией.

На рис. 7 и рис. 8 на каждом узле используются по восемь ядер: для программы FT N2=8, для программы FT_T TSize=8. МПМП программа FT_T выполняется быстрее, чем многопроцессная программа FT на 9,5%-20%. В FT_T были распараллелены на потоки функции связанные с локальным транспонированием и расчетами.

Инициализация же данных в каждом процессе выполняется в последовательно (используется один поток). То есть степень параллелизма в данном случае меньше чем для программы FT. Если распараллелить на потоки и инициализационные функции, то время выполнения программы FT_T сократиться еще больше.

Нужно заметить, что с увеличением количества используемых ядер преимущество FT_T над FT уменьшается. Профилирование программы показало, что связано это с глобальным транспонированием по оси x, которое производится посредством коллективной коммуникационной функции AlltoAll между процессами программы. На рис. 8 приведены графики зависимости времени выполнения коммуникационных функций в программах FT и FT_T от количества используемых ядер.

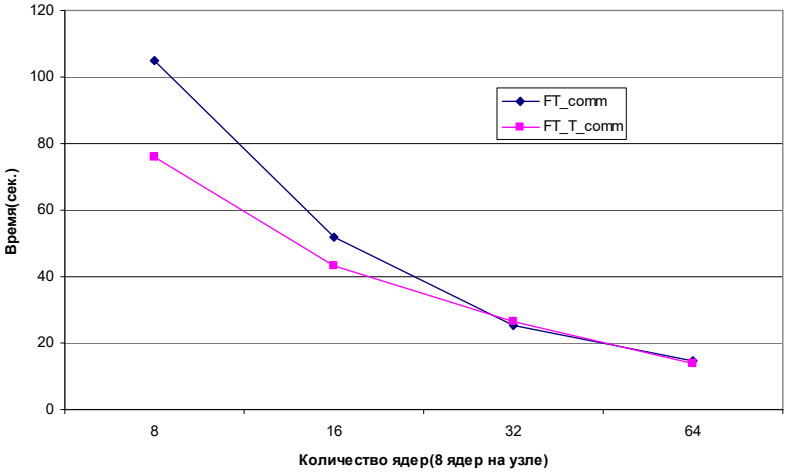

Рис. 8. Сравнение времени выполнения коммуникаций в многопроцессной программе БПФ с МПМП версией.

Как видно из графиков в начальных точках (8 ядер, 16 ядер) время выполнения коммуникаций в программе FT_T заметно меньше, чем для коммуникаций в FT, но с увеличением количества ядер разрыв уменьшается. Это связано с тем, что в FT_T количество обменивающихся процессов TSize раз меньше чем в FT, а размер пересылаемых сообщений гораздо больше. В случае с коллективной коммуникацией AlltoAll время выполнения функции меньше, когда задействованы больше процессов. В этом случае задействованы больше сетевых карт, больше каналов связи и соответственно сокращается простои по ожиданию тех или иных процессов.

На рис. 9 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT. На данном рисунке FT_1Loc и FT_1Fin относятся к подготовке транспонированию по оси x, FT_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера), FT_2Loc и FT_2Fin относятся к подготовке транспонированию по оси z, FT_2Glo относится к транспонированию по оси z (глобальная коммуникация между процессами внутри узлам кластера).

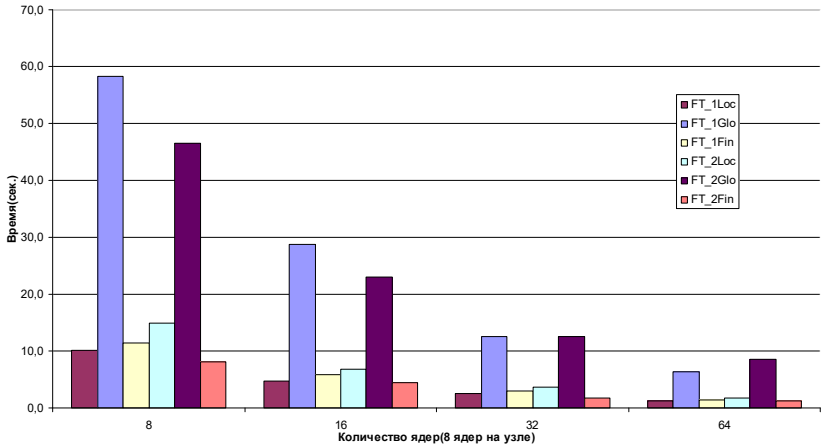

Рис.9. Временной профиль транспонирования в многопроцессной программе БПФ.

На рис. 10 приведены графики времени выполнения функции транспонирования и ее составных частей для программы FT_T. На данном рисунке FT_T_1Loc и FT_T_1Fin относятся к транспонированию по оси z, FT_T_1Glo относится к транспонированию по оси x (глобальная коммуникация между узлами кластера). В программе FT_T отсутствует необходимость коммуникаций при транспонировании по оси z поскольку все потоки в рамках одного процесса имеют доступ к необходимым данным благодаря общей памяти. В этом случае для повышения производительности (кэш попадания) производится лишь копирование в последовательный массив.

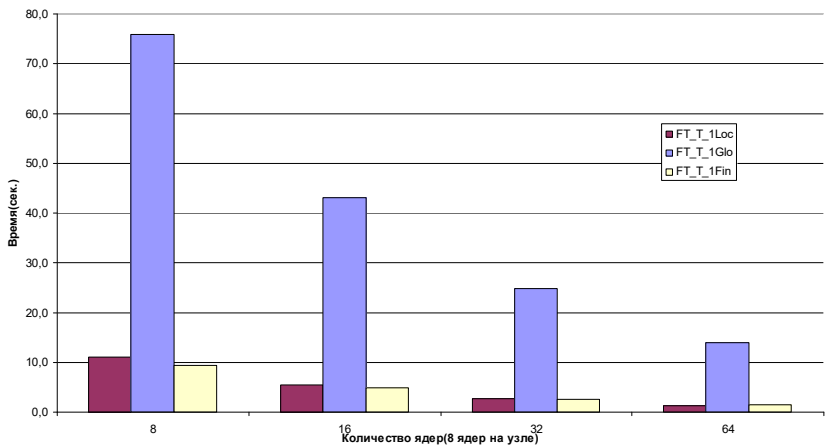

Рис. 10. Временной профиль транспонирования в МПМП программе БПФ.

Из рис. 9, 10 видно, что основная причина деградации производительности программы FT_T это эффект связанный с графиком FT_T_1Glo. В программе FT производятся две коммуникации FT_1Glo и FT_2Glo. При увеличении количества ядер в FT_2Glo не меняется количество обменивающихся процессов, но сокращается размер сообщений. А в FT_1Glo увеличивается количество обменивающихся процессов, одновременно уменьшается размер сообщений. Ситуация с FT_T_1Glo похожа на FT_1Glo, однако в данном случае размер сообщений больше и в функции FT_1Glo параллельно выполняются N2 AlltoAll коммуникаций.

Увеличение количества используемых ядер приводит к существенному сокращению времени выполнения параллельных участков программы, а последовательная часть программы остается почти неизменной, что приводит к увеличению ее доли. Поскольку же степень параллелизма в FT_T меньше чем в FT, FT_T и ускоряется меньше.

Таким образом, в текущей версии программ FT, FT_T при использовании от восьми до тридцати двух ядер МПМП программа FT_T работает в среднем на 10-20% быстрее, чем аналогичная многопроцессная программа FT. Но при дальнейшем увеличении количества ядер преимущество сокращается. В дальнейшем планируется замена схемы коммуникаций в МПМП версии с целью увеличения степени параллелизма, что приведет к ускорению программы.

4. Заключение.

Библиотека mpiJava, разработанная в среде ParJava, позволяет разрабатывать параллельные МПМП приложения на языке Java оставаясь в рамках стандарта MPI. В этом случае параллельная программа запускается в нескольких

процессах, где в начале программы выполняется один поток (основной поток), который последовательно выполняет инструкции программы. В определенных точках программы, где необходимо выполнить параллельно на нескольких потоках некий набор инструкций, пользователь формирует задания, содержащие инструкции, и передает ссылки на задании библиотеке времени выполнения. Полученные пользовательские задания выполняются каждый в отдельном потоке.

В рамках данной работы была реализована МПМП программа быстрого преобразования Фурье на языке Java. Проведенные эксперименты показали, что МПМП программа (FT_T) работает быстрее, чем многопроцессная программа (FT) в среднем на 9,5%-20%. Однако с приростом количества ядер преимущество FT_T над FT уменьшается. Связано это в основном с глобальным транспонированием по оси x, которое производится посредством коллективной коммуникационной функции AlltoAll между процессами программы. В каждом процессе уменьшается доля параллельных вычислений по сравнению с последовательными (инициализация, коммуникации). В дальнейшем планируется замена схемы коммуникаций в МПМП версии с целью увеличения степени параллелизма, что приведет к ускорению программы.

Список литературы

[1]. Иванников В. П., Аветисян А. И., Гайсарян С. С., Акопян М. С. Особенности реализации интерпретатора параллельных программ в среде ParJava. // «Программирование» 2009, №1, с. 10-25

[2]. М.С. Акопян. Расширение модели ParJava для случая кластеров с многоядерными узлами. Труды Института системного программирования РАН, том 23, 2012, с. 13-32

[3]. http://docs.oracle.com/javase/tutorial/essential/concurrency/

[4]. http://www.ispras.ru/ru/parjava/mpijava.php

[5]. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[6]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998

[7]. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[8]. Barbara Chapman, Gabriele Jost, Ruud van van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). ISBN-13: 978-0262533027, MIT, October 2007

[9]. Husbands P, Iancu C, Yelick KA (2003) A performance analysis of the Berkeley UPC compiler. In: International conference on supercomputing, pp 63–73

[10]. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

[11]. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

[12]. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.

Using Multithreaded Processes in ParJava Environment

M.S. Akopyan <manuk@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. Modern processors are based on multicore architectures. Such an approach improves the productivity of processors by decreasing the cost of each processor and increasing its performance. When using multicore processors in nodes of HPC cluster we could use following approaches utilizing node resources: (1) multiprocess program (nx1 - running n processes on a node using one thread in each process); or (2) multiprocess-multithreaded (MPMT) (1xn - running one process on a node and inside of a process n threads may work sharing program data of the process). When using multiple threads in a process inside each process privatization is usually performed to reduce critical sections. In this article we consider the second approach, which will bring better results for parallel application presented in this article because of lack of critical sections. The API and appropriate library has been developed and implemented for MPMT applications. The library allows developing parallel applications using MPI interface and inside of each process it is possible to run a few threads. The parallel MPMT application of FT (Fast Furier Transformation) on Java has been developed. The comparison of multiprocess version of FT to MPMT version of FT has been made. Tests on implemented application show 9,5-20% performance improvement. The profiling of developed application shows the bottleneck of MPMT FT is mostly in communication scheme between nodes. Improving the communication scheme will bring better results.

Keywords: parallel computing; parallel SPMD programs using MPI; multi-core; multithreaded Java programs; multiple process-multithreaded programs.

DOI: 10.15514/ISPRAS-2015-27(2)-1

For citation: Akopyan M.S. Using Multithreaded Processes in ParJava Environment. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 5-22 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-1

References

[1]. Ivannikov V.P., Avetisyan A.I., Gaissaryan S.S., Akopyan M.S. Implementation of Parallel Interpreter in the Development Environment ParJava. Programming and Computer Software. 2009. Volume 35, Issue 1. pp. 6-17. doi: 10.1134/S0361768809010034

[2]. Akopyan M.S. Rashirenije modeli ParJava dlja klasterov s mnogojadernymi uzlami [Extension of ParJava model for HPC clusters with multicore nodes]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 13-32 (in Russian).

[3]. http://docs.oracle.com/javase/tutorial/essential/concurrency/

[4]. http://www.ispras.ru/ru/parjava/mpijava.php

[5]. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Revised version, August 1999.

[6]. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The complete Reference, Volume 1, The MPI Core, Second edition. / The MIT Press. 1998

[7]. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

[8]. Barbara Chapman, Gabriele Jost, Ruud van van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). ISBN-13: 978-0262533027, MIT, October 2007

[9]. Husbands P, Iancu C, Yelick KA (2003) A performance analysis of the Berkeley UPC compiler. In: International conference on supercomputing, pp 63–73

[10]. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. THE NAS PARALLEL BENCHMARKS. THE NAS PARALLEL BENCHMARKS. RNR Technical Report RNR-94-007, March 1994

[11]. H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.

[12]. Dami´an A. Mall´on, Guillermo L. Taboada, Juan Touri˜no, and Ram´on Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. // Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09). Weimar, Germany, Feb 2009, pp. 181-190.