Расширение модели ParJava для случая кластеров с многоядерными узлами[1]

М.С. Акопян
manuk@ispras.ru

Аннотация. В работе описывается расширение модели параллельной SPMD программы возможностью использования потоков Java. Использование потоков в программе позволяет лучше утилизировать ресурсы многоядерного процессора. Разработанная модель позволяет оценивать время выполнения параллельной программы с явными обращениями к библиотеке MPI, где в каждом процессе можно использовать параллельные потоки Java. Однако следует учесть трудности, возникающие при использовании потоков в среде Java. В работе приводятся рекомендации призванные улучшить производительность многопроцессно-многопоточной программы связанные с управлением памяти JVM, настройкой сборщика мусора, управлением локальных буферов и.т.д.

Ключевые слова: параллельные вычисления; моделирование параллельной по данным программы; оценка времени выполнения; оценка масштабируемости; многоядерность; доводка производительности Java программ.

1. Введение

В начале 2000-ых годов каждый узел высокопроизводительной вычислительной платформы с распределенной памятью (кластер) содержал процессор с одним вычислительным устройством (ядро). Процесс параллельной программы использовал все доступные ресурсы узла. С развитием технологий стало возможно за минимальные накладные расходы на чипе процессора разместить более одного ядра, тем самым увеличив количество выполняемых модулей, что привело к повышению производительности процессора. Также увеличились размер кэша первого уровня, количество конвейерных устройств. Однако такие аппаратные ресурсы как кэш второго уровня, канал памяти представляют собой разделяемые ресурсы. На данный момент появились процессоры с 8-ю ядрами. Как показывает опыт, использование потоков в рамках одного узла на современных кластерах с многоядерными процессорами может увеличить

[1] Работа выполнена при финансовой поддержке Минобрнауки РФ, контракт 07.514.11.4001.

производительность параллельной программы за счет использования общей памяти и уменьшения накладных расходов.

Среда ParJava[1] предоставляет прикладному программисту набор инструментов, поддерживающих разработку прикладных параллельных программ для вычислительных систем с распределенной памятью (высокопроизводительных кластеров) на языке Java с явными обращениями к библиотеке MPI [2]. Многие инструменты среды ParJava могут выполняться на инструментальном компьютере, используя вместо разрабатываемой параллельной программы ее модель [3], которая может интерпретироваться как на целевой вычислительной системе, так и на инструментальном компьютере. Модель параллельной программы адекватно отражает ее поведение на кластере, позволяя исследовать динамические характеристики (в частности, время выполнения) разрабатываемой программы, или ее частей. Изменения, внесенные в [4] ускорили время интерпретации модели и позволили проводить интерпретацию реальных параллельных приложений с большим объемом данных на инструментальном компьютере.

В данной статье представлено расширение модели возможностью использования потоков Java - среда ParJava позволяет разрабатывать и моделировать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI, причем каждый процесс MPI может содержать определенное количество потоков Java, утилизируя тем самым ядра процессора и общие ресурсы.

В разделе 2 описывается использование потоков в параллельной MPI программе и приводится краткое описание библиотеки для работы с потоками Java. Раздел 3 посвящен описанию расширения модели ParJava возможностью интерпретации Java потоков. В разделе 4 приводится описание эффектов призванных повысить производительность параллельной Java программы. В разделе 5 описываются результаты численных экспериментов.

2. Использование Java потоков в параллельной MPI программе. Библиотека времени выполнения для работы с потоками

Согласно стандарту MPI в параллельной программе каждый MPI процесс представляет собой многопоточную программу. В каждом потоке могут вызываться функции MPI, однако сами потоки не адресуемы (параметр rank в коммуникационных функциях MPI относится к процессу). Сообщение, отправленное процессу с идентификатором rank, может быть получено любым потоком, выполняющимся в данном процессе. Функции для работы с потоками в стандарте не определены и оставлены на усмотрение пользователя.

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.

Среда ParJava позволяет разрабатывать и моделировать параллельные приложения на современном языке Java, оставаясь в рамках промышленного стандарта MPI, причем каждый процесс MPI может содержать определенное количество потоков Java. Коммуникации в параллельных SPMD программах в среде ParJava обеспечиваются двумя способами: а) обмен данных с помощью коммуникационных функций MPI (библиотека mpiJava.mpi) между процессами программы, б) функции взаимодействия (синхронизации) между потоками одного процесса в рамках одного узла вычислительной платформы (библиотека mpiJava.threads).

Коммуникационная библиотека mpiJava.mpi реализующая MPI основана на пакете mpiJava[2], который представляет собой привязку языка Java через интерфейс JNI к существующей реализации MPI (MPICH,LAM …). В библиотеке реализованы оберточные функции для стандарта MPI1.1[5].

Пакет mpiJava был расширен методами поддерживающими использование потоков согласно стандарту MPI:

- Реализованы функции из стандарта MPI MPI_INIT_THREAD, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN
- Разработан пакет mpi.threads реализующий функции взаимодействия между потоками основанный на пакете java.util.concurrent[6].

Описание пакета mpiJava.threads приведено [7], ниже дается краткое описание основных методов для работы с потоками. Методы из пакета mpiJava.threads можно разбить на следующие группы:

- Создание и управление потоками. Методы из данной группы позволяют создавать пул потоков, помещать новые задания в очередь заданий и.т.д.
- Методы атомарных операций. Методы из данной группы позволяют производить две операции над переменными программы атомарно (обращение к данным переменным другими потоками возможно только после завершения данного метода).
- Методы для работы с критическими секциями и семафорами.
- Методы, не вошедшие в другие группы.

Разработанная библиотека времени выполнения предоставляет возможность пользователю в рамках MPI процесса создавать пул потоков и выполнения программы добавлять в очередь заданий пула новые задания. Использование пула потоков позволяет избежать накладных расходов при создании и запуска новых потоков в ходе выполнения программы. Синхронизация между потоками производится посредством критических секций и семафоров.

В ParJava работа с потоками в рамках процесса MPI ведется следующим образом: в каждом процессе создается пул с фиксированным количеством

потоков. В ходе работы программы пользователь формирует пользовательские задания и передает пулу потоков. Менеджер пула передает задание первому свободному потоку, в котором и выполняется данное задание. В конце программы пул потоков закрывается.

Замечание. В среде ParJava введено ограничение, согласно которому в каждом процессе параллельной Java+MPI+threads программы используются одинаковое количество потоков.

Замечание. В среде ParJava введено ограничение, согласно которому в каждом процессе пул потоков создается в последовательной части основного потока (main) и закрывается в конце выполнения основного потока. В одном процессе нельзя создать больше одного пула потоков.

Пользовательское задание 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. Взаимодействие между потоками программы производится с помощью таких механизмов как критические секции, семафоры, атомарные операции над переменными и.т.д.

Таким образом, разработанная библиотека дает возможность пользователю разрабатывать параллельные приложения на языке Java с явными обращениями к MPI и использовать потоки Java в каждом процессе программы.

3. Моделирование Java потоков и функций взаимодействия между потоками в интерпретаторе ParJava

Пусть на входе имеется параллельная MPI программа на языке Java, где в каждом MPI процессе запускается одинаковое количество потоков программы. Необходимо расширить модель параллельной программы в среде ParJava возможностью моделирования потоков Java, а также разработать и реализовать версию интерпретатора ParJava поддерживающую обработку потоков и функции взаимодействия (синхронизации) между потоками из библиотеки mpiJava.threads.

Каждый MPI-процесс моделируемой программы представляется в ее модели с помощью логического процесса (LProc). Логический процесс определен как последовательность действий (примеры действий: выполнение базового блока, выполнение операции обмена и т.п.). Каждое действие имеет определенную продолжительность. В логическом процессе определено понятие модельных часов. Начальное показание модельных часов каждого логического процесса равно нулю. После интерпретации очередного действия к модельным часам соответствующего логического процесса добавляется

значение времени, затраченного на выполнение этого действия (продолжительности). Продолжительность каждого действия, а также значения исследуемых динамических параметров базовых блоков, измеряются заранее на целевой платформе.

Пользовательские потоки моделируются с помощью расширенного логического процесса (ELProc). Каждый объект класса ELProc обладает локальными модельными часами, время которых хранится в переменной time. При установлении нового задания (pr_SetTask) запускается пользовательский поток, и время его модельных часов устанавливается равным времени модельных часов родительского процесса. При интерпретации модели интерпретатор обновляет переменную **time** потока для каждого фрагмента (базовый блок, сбалансированное гнездо циклов и.т.д.) выполненного в рамках данного потока. После завершения работы поток посылает сообщение пулу об окончании и сообщая ему время своего выполнения.

Если в основном потоке после запуска группы новых заданий с помощью метода SetTask используется блокирующий метод WaitForAll, то выполнение основного потока блокируется. Как только все потоки закончили выполнение, максимальное время выполнения запущенных потоков прибавляется к модельным часам основного потока.

Для моделирования функции взаимодействия между потоками в интерпретаторе среды ParJava используются следующие базовые примитивы:

pr_CreateThreadPool – создание пула потоков. Пользовательские задания могут выполняться только потоками из созданного пула потоков.

pr_SetTask – установка нового задания. Выбирается свободный поток из пула и ему дается новое задание на выполнение.

pr_CreateAtomic – создание атомарной переменной.

pr_CompareAndSet – атомарное сравнение и присвоение переменной.

pr_CreateSemaphore – создание семафора.

pr_Lock – блокирующий вызов для входа в регион ограниченного доступа.

pr_TryLock – неблокирующий вызов для входа в регион ограниченного доступа.

pr_Unlock – выход из региона ограниченного доступа.

pr_Copy – копирование сообщения.

Функции из библиотеки времени выполнения для работы с потоками полностью моделируются в интерпретаторе с помощью вышеописанных примитивов.

3.1. Оценка времени выполнения базовых примитивов моделирования потоков Java.

В данном разделе описывается метод оценки времени выполнения функций взаимодействия между потоками. Время выполнения базовых примитивов моделирующих операции над потоками определяется следующим образом:

pr_CreateThreadPool(n) – время создания одного потока обозначим как $\mathbf{time}_{\mathbf{tCreate}}$. В таком случае время выполнения данного примитива определяется как

$$\text{Time(pr\_CreateThreadPool(n))} = n\ \mathbf{time}_{\mathbf{tCreate}} + \mathbf{time}_{\mathbf{tnArray}}$$

где $\mathbf{time}_{\mathbf{tnArray}}$ время необходимое для создания массива для хранения n потоков.

pr_SetTask(task, isBlocking) – пусть $\mathbf{time}_{\mathbf{tStart}}$ время запуска потока, а $\mathbf{time}_{\mathbf{tRun}}$ время выполнения потока с заданием task. В этом случае

$$\text{Time(pr\_SetTask(task, isBlocking))} = \mathbf{time}_{\mathbf{tStart}} + (\text{isBlocking} ?\ \mathbf{time}_{\mathbf{tRun}} : 0)$$

Переменная $\mathbf{time}_{\mathbf{tRun}}$ определяется во время интерпретации и равно разности показаний модельных часов пользовательского потока и родительского потока.

Время работы примитивов **pr_CreateAtomic(atomic,initValue)**, **pr_CreateSemaphore(initN, mode), pr_Unlock(sem,n)** константно и определяется посредством тестов – пусть $\mathbf{time}_{\mathbf{aCreate}}$ время создания атомарной переменной, $\mathbf{time}_{\mathbf{sCreate}}$ время создания семафора, $\mathbf{time}_{\mathbf{release}}$ время разблокировки семафора.

$$\text{Time(pr\_CreateSemaphore)} = \mathbf{time}_{\mathbf{sCreate}}$$

$$\text{Time(pr\_Unlock)} = \mathbf{time}_{\mathbf{release}}$$

$$\text{Time(pr\_CreateAtomic)} = \mathbf{time}_{\mathbf{aCreate}}$$

pr_TryLock(sem,n) - пусть $\mathbf{sem}_{\mathbf{counter}}$ значение счетчика в семафоре. Переменная $\mathbf{sem}_{\mathbf{ts}}$ представляет собой временную метку семафора. Когда поток освобождает регион с ограниченным доступом (примитив pr_Unlock) переменной семафора $\mathbf{sem}_{\mathbf{ts}}$ присваивается показание модельных часов потока.

$$\text{Time(pr\_TryLock(sem,n))} = \mathbf{time}_{\mathbf{cmp}} + (((\mathbf{sem}_{\mathbf{counter}} - n) \geq 0) ?$$

$$\text{pr\_CompareAndSet(atomic, } \mathbf{sem}_{\mathbf{counter}}, (\mathbf{sem}_{\mathbf{counter}} - n) : 0)$$

где $\mathbf{time}_{\mathbf{cmp}}$ время сравнения $(\mathbf{sem}_{\mathbf{counter}} - n) \geq 0$.

pr_Lock(sem,n) – пусть $\mathbf{time}_{\mathbf{lock}}$ время блокировки потока и определяется следующим образом $\mathbf{time}_{\mathbf{lock}} = (\mathbf{time} < \mathbf{sem}_{\mathbf{ts}})$? $(\mathbf{sem}_{\mathbf{ts}} - \mathbf{time})$: 0. Тогда

$$\text{Time(pr\_Lock(sem,n))} = \text{Time(pr\_TryLock(sem,n))} + (\text{pr\_TryLock(sem,n)} ?$$

$$\mathbf{time}_{\mathbf{lock}} : 0)$$

pr_CompareAndSet(atomic,expect,update) - $\mathbf{time_{aCAS}}$ время сравнения и присвоения переменной. Эта операция производится следующим образом – вначале открывается критическая секция, после чего производиться сравнение и присвоение переменной, и закрытие критической секции. Тогда время выполнения примитива определяется как

$$\text{Time(pr\_CompareAndSet)} = \mathbf{time_{aCAS}} + \text{Time(pr\_Lock(sem,1))} + \text{Time(pr\_Unlock(sem,1))}$$

pr_Copy(n) – пусть $\mathbf{time_{copy}}$ время копирования одного байта и определяется с помощью тестов. В этом случае

$$\text{Time(pr\_Copy())} = n\ \mathbf{time_{copy}}$$

Параметры $\mathbf{time_{tCreate}}$, $\mathbf{time_{tnArray}}$, $\mathbf{time_{tStart}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$, $\mathbf{time_{cmp}}$, $\mathbf{time_{copy}}$ определяются с помощью тестов на узле вычислительной платформы.

Замечание. Значение параметров $\mathbf{time_{tStart}}$, $\mathbf{time_{sCreate}}$, $\mathbf{time_{release}}$, $\mathbf{time_{aCreate}}$, $\mathbf{time_{aCAS}}$, $\mathbf{time_{cmp}}$ сильно меньше чем значение остальных параметров и их вклад в определении времени выполнения коммуникаций незначителен, поэтому в дальнейшем будем считать их равными нулю.

Время выполнения функций из групп создания и управления потоками, методов атомарных операций, методов для работы с критическими секциями элементарно вычисляются, используя оценки времени выполнения базовых примитивов приведенных выше.

Таким образом, имея оценки времени выполнения базовых примитивов определяется время выполнения основных функций взаимодействия между потоками в пользовательской программе.

4. Доводка производительности параллельной программы в среде Java

В данном разделе приводится описание эффектов призванных повысить производительность параллельной программы. Вначале сравнивается время выполнения обмена данными между процессами программы и время выполнения чтения данных между потоками программы. После чего приводится описание нескольких эффектов связанных с реализацией JVM, влияющих на производительность параллельной программы. Утилизация специфических факторов описанных в данном разделе прикладным программистом может привести к потенциальному росту производительности параллельной программы.

4.1. Операция «обмена данными» между процессами/потоками Java

Рассмотрим время выполнения **операции «чтение данных соседнего процесса/потока»** для многопроцессной программы, в котором используются неблокирующие, блокирующие коммуникаций MPI, и для многопоточной программы.

Пусть имеется многопроцессная программа P1 и многопоточная программа P2. Для программы P1 (P2) в каждом процессе (потоке) производятся вычисления и в определенный момент процесс (поток) 0 становится потребителем (p_0) и должен получить доступ к данным процесса (потока) 1 который в этом взаимодействии является производителем (p_1).

В многопроцессной программе P1 при использовании **неблокирующих** коммуникаций время выполнения операции «чтение данных соседнего процесса» определяется следующей формулой:

$$\text{time}_{\text{wait}} = \begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ (t_0 - t_{ready}) > time_{comm}\ ?\ 0 : time_{comm} - (t_0 - t_{ready}), t_{irecv} < t_{ready} < t_0 \\ (t_0 - t_{irecv}) > time_{comm}\ ?\ 0 : time_{comm} - (t_0 - t_{recv}), t_{ready} < t_{irecv} \end{cases}$$

где t_{irecv} - момент времени вызова iRecv в процессе p_0, t_0 - момент времени вызова wait в процессе p_0, t_{ready} - момент времени когда в процессе p_1 готовы данные, time_{comm} - время необходимое для пересылки данных из процесса-производителя к процессу-потребителю и копирования в буфер памяти в процессе-потребителе, time_{wait} – время выполнения функции wait.

При использовании блокирующих коммуникаций:

$$\text{time}_{\text{wait}} = \text{time}_{\text{recv}} = \begin{cases} (t_{ready} - t_0) + time_{comm}, t_{ready} > t_0 \\ time_{comm}, t_{ready} < t_0 \end{cases}$$

где time_{recv} – время выполнения функции Recv, t_0 - момент времени вызова Recv в процессе p_0, t_{ready} - момент времени, когда в процессе p_1 готовы данные, time_{comm} - время необходимое для пересылки данных из процесса p_1 к процессу p_0 и копирования в буфер памяти в процессе p_0.

В случае с **многопоточной** программой P2 поток производитель p_1 вызывает функцию блокирования lock и начинает вычислять данные. Как только данные вычислены он разблокирует критическую секцию. Поток-потребитель p_0, не может считать данные пока p_1 не разблокирует критическую секцию (p_0 блокируется в lock-е). Время выполнения операции «чтение данных соседнего потока» определяется следующей формулой:

$$\text{time}_{\text{lock}} = \begin{cases} 0; t_{ready} \le t_0 \\ t_{ready} - t_0, t_{ready} > t_0 \end{cases}$$

где t_0 - момент времени вызова lock в потоке-потребителе p_0, t_{ready} - момент времени, когда в потоке-производителе p_1 готовы данные, $time_{lock}$ – время выполнения функции lock.

Отсюда видно, что в любом случае (при использовании блокирующих или неблокирующих коммуникаций и с учетом реализации обмена данными в MPI на основе общей памяти) в многопроцессной программе P1 на операцию «чтение данных соседнего процесса» тратится больше времени, чем в многопоточной программе P2.

Когда процесс вызывает функцию wait (явно при неблокирующей посылке или посредством recv), то при определенных условиях (не готовы данные или производится пересылка) вызов блокируется. Блокирование происходит внутри реализации MPI и в зависимости как реализована операция блокирования в MPI это может возыметь определенные последствия:

1. если lock реализован как lock-sleep, то произойдет переключение контекста и когда данные прибудут, процесс пробудится и опять будет произведен переключение контекста
2. если lock реализован посредством spin-lock, то процесс останется активным и переключений контекстов не произойдет.

В случае с **многопоточной** программой операцию lock определяет пользователь. Если он явно организует spin-lock, то очевидно контекст переключаться не будет. Когда пользователь вызывает функцию lock (например semaphore.acquire()), то какой именно подход блокирования будет выбран зависит от реализации JVM.

Переключение контекста не является бесплатной операцией – управление CPU передается JVM и операционной системе, которые сохраняют данные потока и отправляют в спячку (режим ожидания). Когда поток должен пробудиться, то система (JVM) загружает его фрейм и данные в память. И скорее всего это переключение повлечет за собой определенное количество кэш промахов, что в свою очередь также отразиться на производительности.

4.2. Синхронизация потоков при обращении к данным в общей памяти Java

При использовании общей памяти потоками параллельной программы значительный вес в доводке производительности программы получают эффекты, связанные с синхронизацией доступа потоков к общим данным. В Java синхронизация данных ведет к появлению дополнительных накладных расходов: синхронизация обеспечивается специальными инструкциями барьеров памяти (memory barriers), которые могут сбрасывать кэш, аппаратные буферы, а также повлиять на другие оптимизации компилятора (предотвратить переупорядочивание инструкций). Синхронизация бывает оспариваемая (**contended)** и неоспариваемая (**uncontended)**. «неоспариваемая» синхронизация имеет низкую стоимость (20-250 тактов)[8]. «Неоспариваемая»

синхронизация может обрабатываться внутри JVM, в то время как «оспариваемая» требует вмешательства операционной системы, что увеличивает ее стоимость. Динамический компилятор Java может сократить стоимость синхронизации, заменив «оспариваемую» синхронизацию «неоспариваемой», когда докажет, что за данный барьер никогда не производится спор между потоками (contention). Существует три способа сократить «спор»:

- Сократить продолжительность блокирования
- Сократить частоту запросов блокирования
- При возможности заменить исключающую блокировку механизмами позволяющими улучшить параллелизм

4.3. Эффекты, связанные со сборщиком мусора в Java

В **многопроцессной** программе каждый процесс запускается на отдельном JVM. Необходимая память для работы процесса выделяется системой управления памяти **своего JVM**. В этом случае, для каждого процесса работает **свой сборщик мусора(gc)**. В случае же с **многопоточной** программой память необходимая всем потокам программы выделяется на одном JVM, и это надо учесть при запуске программы в опциях интерпретатора(xmx,…). Помимо этого здесь работает только **один gc**, который обслуживает все потоки программы. Следовательно, давление на gc возрастает пропорционально количеству потоков программы.

В JVM освобождением не нужной памяти занимается сборщик мусора, который время от времени запускается в отдельном потоке и может стать узким местом в параллельной программе. С целю улучшения производительности необходимо уменьшить время работы gc, а в идеальном случае и вовсе свести на ноль. Сборщик мусора перебирает кучу (heap) и выделяет (маркирует) те области, на которые есть живые ссылки. После этапа маркировки производится чистка всех немаркированных областей. Обычно большинство объектов выделенных в куче используются не долго (умирают рано), они хранятся в молодом поколении. Другие объекты могут жить до конца работы программы, и они хранятся в долгоживущем поколении. Процесс сборки мусора можно оптимизировать, используя эффект ранней смерти. С целю уменьшения времени работы gc общая память в куче разбивается на несколько областей:

1. Молодое поколение(young generation) – хранятся молодые объекты.
2. Долгоживущее поколение(tenured) – хранятся долгоживущие объекты.
3. Перманентная область (permanent) – здесь хранятся служебные данные JVM.

Как только в поколении не остается места производиться сборка мусора для данного поколения: минорная сборка(minor collection) для молодого

поколения и мажорная сборка (major collection) для долгоживущего поколения. Большинство выделенных объектов умирают молодыми, следовательно, минорная сборка занимает меньше времени, потому что время сборки прямо пропорционально количеству живых объектов. При создании объекта в программе, производится попытка выделить память в молодом поколении, если этого не удается (молодое поколение переполнено), то производится минорная сборка (minor collection), чистящая молодое поколение. Во время минорных сборок, объекты, которые прожили достаточно долго, переносятся в долгоживущее поколение. Как только долгоживущее поколение переполняется, производится мажорная сборка, которая работает намного медленнее из за большого количества живых объектов в нем.

Для уменьшения влияния сборщика мусора на производительность программы рекомендуется:

1. Большие матрицы и общие данные выделить в самом начале программы.
2. Вычислить объем данных выделенных в первом шаге, и соответствии с этим задать минимальный и максимальный размер кучи (Xmx,Xms).
3. Задать размеры поколений (NewRatio, NewSize, MaxNewSize).

Чем больше размер молодого поколения, тем меньше будет минорных сборок. С другой стороны, чем больше размер молодого поколения, тем меньше размер долгоживущего поколения, что в свою очередь приведет к частым мажорным сборкам. Поэтому размеры поколения нужно выбрать так, что большие матрицы, выделенные в начале программы, попадали сразу в долгоживущее поколение, а так же оценить размер памяти, который будет выделяться по ходу выполнения программы, и задать соответствующий размер молодого поколения так, чтобы не было (по возможности) минорных сборок.

4.4. Эффекты, связанные с буфером локальной памяти потоков в Java

В многопоточной программе выделение памяти потоками производится параллельно. В общем случае все потоки выделяют память в общей куче. Каждый раз при выделении памяти в общей куче(heap) система управления памяти ставит lock, выделяет память и отпускает lock, что может повлиять на производительность общей программы. Поэтому рекомендуется большую матрицу и общие данные выделить в самом начале программы, а локальные данные для каждого потока выделять в локальной области, что позволит избежать lock-ов на общей куче.

Для улучшения производительности при запуске каждого потока JVM создает для него локальный буфер для выделения памяти(TLAB). Во время работы потока при выделении памяти, вначале производится попытка выделить память из области TLAB-а и если эта попытка оказывается неудачной, то

память выделяется в общей куче со всеми вытекающими последствиями. При запуске программы необходимо знать, сколько памяти будет выделено во время работы потока и задать соответствующие корректировки относительно работы с TLAB-ами с помощью опций интерпретатора Java (ResizeTLAB, TLABSize,…).

5. Результаты численных расчетов

Модель, приведенная в данной статье, был реализован в среде ParJava и апробирован на параллельной программе FT из набора NAS Parallel Benchmarks [9,10]. NPB на языке Java был разработан и реализован в университете Coruna [11].

В рамках работы проведенной в данной статье, оригинальная реализация FT была оптимизирована с учетом эффектов описанных в разделе 4 и в результате время выполнения программы уменьшилась в более чем два раза. Помимо этого был реализован многопоточный вариант программы: коммуникации между процессами программы обеспечиваются функциями MPI, а для взаимодействия между потоками в каждом процессе используются функции из библиотекой времени выполнения mpiJava.threads. Проведена серия экспериментов по сравнению параллельной программы основанной только на библиотеке MPI и параллельной программы на основе MPI+threads. Была проведена оценка времени выполнения многопроцессно-многопоточной программы и результаты приведены в данном разделе.

Приведем описание параллельной программы FT. 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 задаче 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. В связи с этим реализованы два вида декомпозиции данных 1D декомпозиция и 2D декомпозиция.

В задаче 3D FFT при применении 1D декомпозиции пространство данных разбивается на слои – каждому процессу отдается один слой (см. рис. 1).

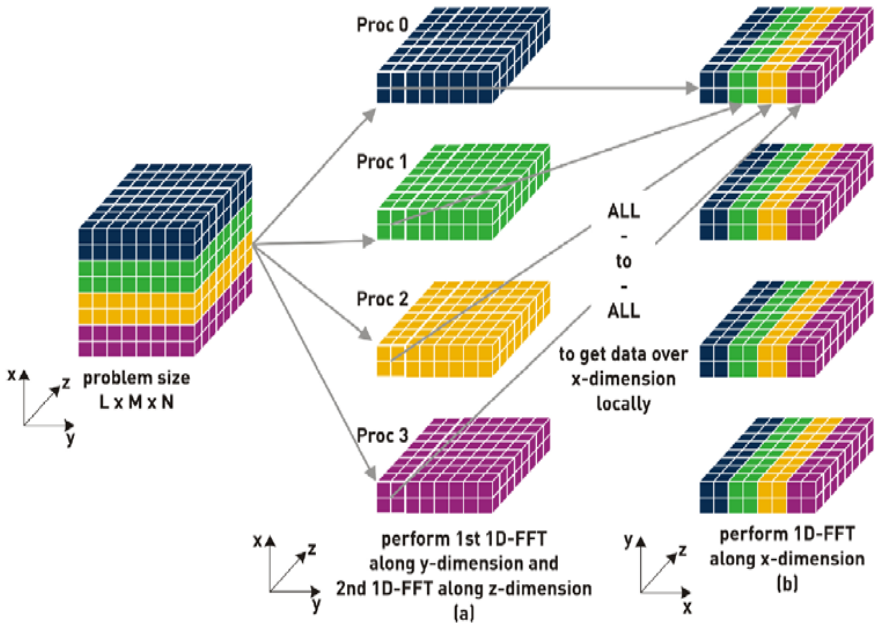

Рис. 1. 1D декомпозиция в решении задачи 3D FFT.

Параллельные вычисления в этом случае производится в 3 этапа

1. 2D FFT(либо 2 раза по 1D FFT) вдоль двух локальных направлениях в рамках процесса (оси z и y).
2. Глобальное транспонирование, что представляет собой коммуникацию AllToAll между всеми процессами. В результате у всех процессов появляются данные для счета вдоль оси x.
3. 1D FFT вдоль 3-го направления.

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 комбинированного алгоритм 1D декомпозиции и распараллеливания по потокам в рамках одного узла.

В случае 2D декомпозиции (см рис. 2) вычисления производятся в 5 этапов

1. 1D FFT в одном направлении (ось y).
2. глобальное транспонирование.

3. 1D FFT по второму направлению (ось z).
4. глобальное транспонирование.
5. 1D FFT по третьему направлению (ось x).

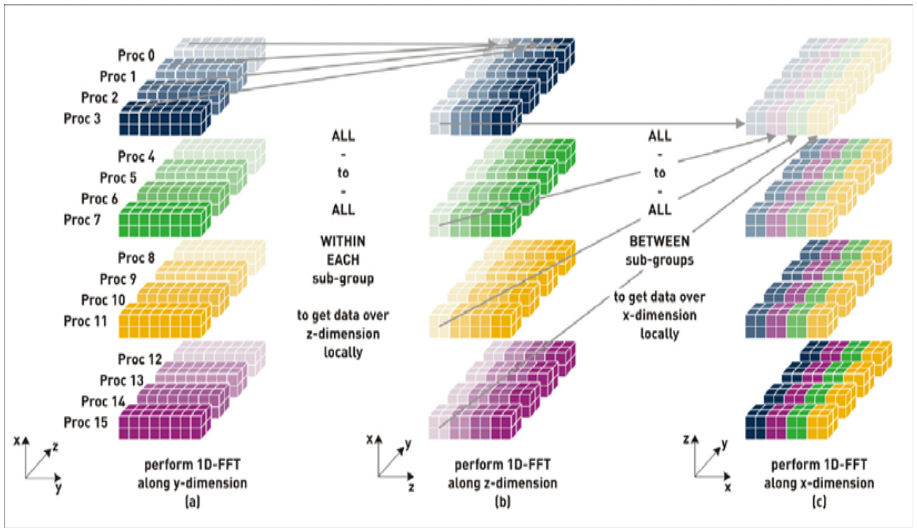

Рис. 2. 2D декомпозиция в решении задачи 3D FFT.

Пусть исходные данные представляют собой куб N^3. В этом случае при 1Dдекомпозиция 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.

На рис. 3 представлены графики зависимости времени выполнения от числа используемых ядер вычислительной платформы для программы быстрого преобразования Фурье (БПФ). Размер рассчитываемой матрицы 512x512x256, объем памяти требуемой в задаче примерно 5Gb, количество внешних итераций 20. Использовалаясь 64-разрядная версия среды Java.

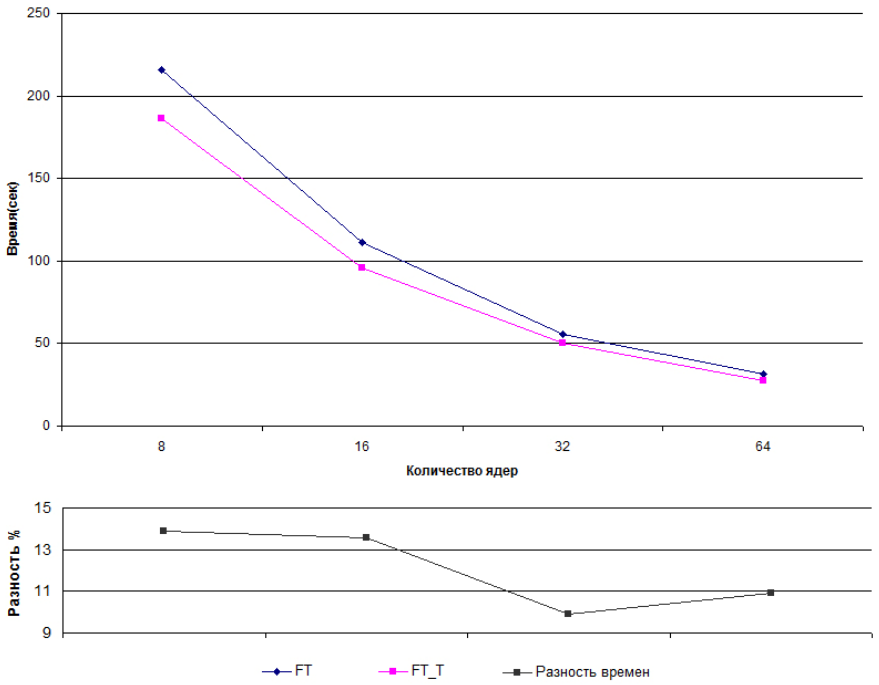

Рис. 3. Зависимость Времени выполнения программы быстрого преобразования Фурье от количества используемых ядер.

На приведенном графике FT представляет собой параллельную программу быстрого преобразования Фурье на языке Java с явными обращениями к MPI. В этом случае применялся 2D декомпозиция, на каждом узле кластера запускается по восемь процессов. FT_T представляет собой параллельную программу быстрого преобразования Фурье, с использованием и интерфейса MPI, и потоков Java. В случае с FT_T применяется 1D декомпозиция – на каждом узле запускается один процесс, внутри которого при обработке гнезд циклов программы порождаются по восемь потоков. «Разность» представляет собой пройентное соотношение разности времен программы FT и FT_T. Алгоритм, используемый в NPB, предполагает, что количество используемых процессов является степенью двойки.

Время выполнения программы FT_T(MPI и потоки Java) 9-14% меньше, чем аналогичной программы FT, где используются только процессы MPI.

Число процессов для FT принимало значения 8 (вычисления проводились на 1 узле), 16 (на 2 узлах), 32 (на 4 узлах) и 64 (на 8 узлах), т.е. на каждом ядре каждого узла был запущен *MPI*-процесс. Для программы FT_T в каждой точке используется по одному процессу на узле, но в каждом процессе используется

по 8 потоков. Таким образов в количество используемых потоков будет 8(на 1 узле), 16 (на 2 узлах), 32(на 4 узлах) и 64(на 8 узлах).

На рис. 4 FT_T представляет собой ускорение программы, измеренное при выполнении программы на вычислительной платформе, а FT_T_Interp - ускорение, предсказанное интерпретатором ParJava на инструментальной машине. При сравнении со временем реального запуска на вычислительной платформе получились существенные погрешности.

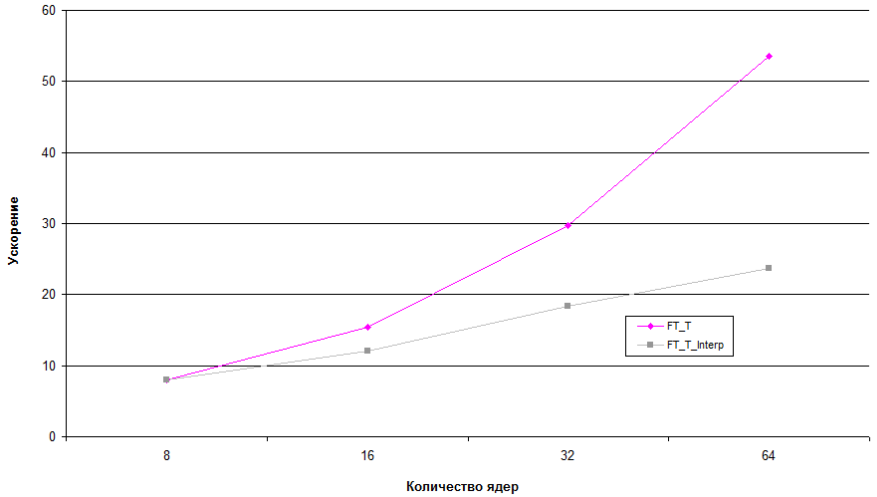

Рис.4. Сравнение предсказанного и фактически полученного ускорения программы FT_T.

Исследования показали, что причина в модели коммуникации метода AlltoAll в интерпретаторе ParJava. Для моделирования коммуникационной функции AlltoAll был реализован алгоритм попарного обмена сообщений с большой длинной. В результате время коммуникаций сократилось больше чем в два раза. На рис. 5 приведено сравнение ускорения по амдалю для программы FT_T и ускорения предсказанного интерпретатором модели после реализации новой версии функции AlltoAll.

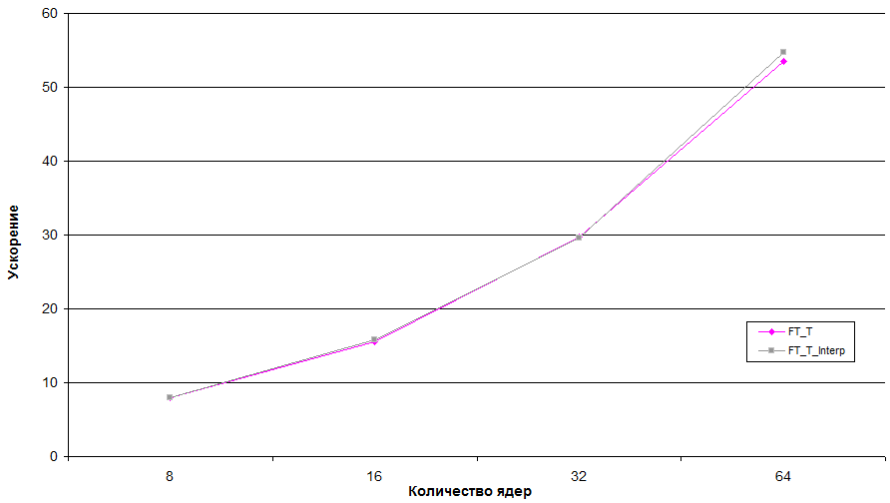

Рис. 5. Ускорение программы БПФ (новая модель коммуникации AlltoAll в интерпретаторе).

Из рис. 5 видно, что интерпретатор модели достаточно точно(3-7%) предсказал ускорение программы при увеличении количества процессов/потоков.

6. Заключение

Расширение модели возможностью работы с потоками было разработано и реализовано в среде ParJava. Была разработана библиотека времени выполнения для работы с потоками и соответствующие модельные функции в интерпретаторе модели в среде ParJava. Разработан метод, оценивающий динамические характеристики (время выполнения) функций взаимодействия между потоками.

Была реализована параллельная программа быстрого преобразования Фурье FT_T на основе параллельной программы FT из тестовых бенчмарков NPB[9,10]. Программа FT_T представляет собой параллельную по данным MPI программу, где в каждом процессе для вычисления основных циклов программы порождаются параллельные потоки (по количеству ядер процессора). Как показали эксперименты программа FT_T за счет использования общих ресурсов и меньших коммуникаций работает быстрее FT на 9-14%.

Однако следует учесть трудности с которыми сталкивается прикладной программист, разрабатывающий параллельное приложение с применением потоков Java (управление памятью, управления локальными буферами

потоков, уменьшение «оспаривания», и.т.д.). В работе приведены основные рекомендации призванные улучшить производительность параллельного приложения с учетом нюансов связанных с JVM при работе с многопоточной программой.

Реализованная модель многопроцессно-многопоточной программы была применена для предсказания параллельного приложения FT_T. Погрешность предсказаний составляет 3-7%.

Список литературы

[1] В.П. Иванников, А.И. Аветисян, С.С. Гайсарян, В.А. Падарян. Оценка динамических характеристик параллельной программы на модели. // «Программирование» 2006, №4, с. 21–37
[2] 12. 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.
[3] 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
[4] А.И. Аветисян, М.С. Акопян, С.С. Гайсарян. Методы точного измерения времени выполнения гнезд циклов при анализе JavaMPI-программ в среде ParJava. Труды Института системного программирования РАН, том 21, 2011, с. 83-102
[5] 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
[6] http://docs.oracle.com/javase/tutorial/essential/concurrency/
[7] http://www.ispras.ru/ru/parjava/mpijava.php
[8] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Java Concurrency In Practice. Addison Wesley Professional, May 19, 2006, p. 384
[9] 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
[10] H. Jagode, “Fourier Transforms for the BlueGene/L Communications Network”, Master’s thesis, University of Edinburgh, 2006.
[11] 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.