Исследование и разработка шаблонов неэффективного поведения в параллельных MPI, UPC приложениях

М.С. Акопян, Н.Е. Андреев [1]
manuk@ispras.ru, andreev.nikita@gmail.com

Аннотация**.** В данной статье рассматриваются шаблоны в параллельных программах, приводящих к потере производительности. Рассматриваются шаблоны как в параллельных MPI приложениях для вычислительных систем с распределенной памятью, так и в параллельных UPC программах для систем с разделенным глобальным адресным пространством (PGAS). В работе предложен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI приложениях и UPC программах. Это позволяет сократить время доводки параллельных приложений.

Ключевые слова: параллельное программирование, семантические ошибки, шаблоны неэффективности, MPI, UPC

1. Введение

Большинство разработанных на сегодняшний день инструментов анализа производительности для различных библиотек и языков параллельного программирования используют низкоуровневые подходы к анализу производительности параллельных программ. Чаще всего это пролифилировочные утилиты, либо визуализаторы трасс. На выходе, в результате анализа, программист получает таблицы и графики со статистикой о выполнении программы. Подобная информация не дает четкого представления о возможных проблемах и узких местах работы приложения. Разработчик вручную просматривает графики в поисках причин замедления

1 Работа проводится при финансовой поддержке Министерства образования и науки Российской Федерации, ГК от «07» марта 2013 г. № 14.514.11.4061

программы и потенциальных возможностей оптимизации. В условиях быстрого роста количества ядер в современных высокопроизводительных вычислительных системах, количество информации, которую необходимо обрабатывать программисту, становится неприемлемо большим, а ручные методы анализа - неприменимыми. Поэтому для оптимизации параллельных приложений в современных условиях, необходимы новые методы анализа производительности, выполняющие полную или частичную автоматизацию обработки получаемой информации.

В работе предложен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI [1] приложениях и UPC [2] программах. Метод базируется на анализе данных полученных во время исполнения параллельной программы в режиме сбора информации (post-mortem анализе). В работе приводится описание шаблонов для программ с использованием MPI и UPC.

В разделе 2 приводится описание аналогичных работ. В разделе 3 рассматривается метод автоматизированного обнаружения шаблонов в MPI, UPC программах. Раздел 4 содержит описание шаблонов неэффективного поведения в MPI программах, использующих коммуникации точка-точка. В разделе 5 приводится описание шаблонов в UPC программах приводящих к потере производительности.

2. Обзор работ

Одна из наиболее известных систем для улучшения производительности параллельных приложений является TAU [3]. TAU – это набор инструментов для анализа производительности параллельных программ, является результатом усилий исследователей Университета Орегона и Исследовательского центра Juelich из Лос-Аламоса. TAU предоставляет набор статических и динамических инструментов, которые с помощью взаимодействия с пользователем производят комплексный анализ параллельных приложений на языках Fortran, C++, C, Java, и Python. Также в TAU разработан статический инструмент автоматического инструментирования. В системе TAU инструмент Hercule [4] представляет собой прототип модуля, использующего базу знаний для выявления и определения причины «узких мест» производительности в соответствии с парадигмой программирования (таких как master-worker, pipeline и.т.д.) вместо модели программирования (MPI, OpenMP). Инструмент Hercule позволяет анализировать приложения, написанные в любой модели программирования. Однако данный инструмент не может обрабатывать приложения, разработанные с применением комбинации разных парадигм.

Система PPW [5] разработана в лаборатории HCS (High-performance Computing and Simulation) в университете Флориды. Система была создана для анализа производительности параллельных PGAS программ (в частности UPC и SHMEM программ). Вначале программа инструментируется и запускается.

В результате работы инструментированной программы собирается профиль программы (статистические данные времени выполнения) и трасса программы (трасса создается в собственном формате). Собранная трасса может быть использована для анализа и выявления узких мест параллельной программы. Также существуют конверторы трассы параллельной программы в известные форматы, что позволяет использовать известные инструменты визуализации (Vampir, JumpShot, и.т.д.) для ручной оптимизации пользователем. PPW является активно разрабатываемым пакетом, имеет графический интерфейс и богатый функционал. Однако методы, лежащие в основе пакета, являются низкоуровневыми и не используют автоматизированных подходов к анализу.

Система Scalasca [6] представляет собой набор инструментов предназначенных для анализа производительности и был спроектирован специально для использования на больших системах с десятками тысяч ядер, но она также хорошо зарекомендовала себя для маленьких и средних HPC платформ. Scalasca поддерживает измерение и анализ конструкций MPI, OpenMP и гибридные программные конструкции, широко используемые в HPC приложения написанных на языках C, C++, Fortran. Система бала разработана в суперкомпьютерном центре Julich и в немецкой исследовательской школе наук моделирования (Jülich Supercomputing Centre and the German Research School for Simulation Sciences). Scalasca поддерживает инкрементальный анализ производительности, который совмещает информацию времени выполнения с глубоким анализом характера параллелизма с помощью трассы событий. Вначале параллельное приложение инструментируется. При запуске каждый процесс создает файл трассы, содержащий записи для локальных событий данного процесса. После того как выполнение параллельной программы завершилось, Scalasca позволяет проводить post-mortem анализ трассы событий. Вначале локальные трассы разных процессов сливаются в единую трассу. При этом для синхронизации часов различных процессов используется метод, описанный в [7]. После слияния локальных трасс в глобальную, можно использовать инструмент EXPERT [8, 9] для выявления шаблонов неэффективности. Инструмент EXPERT последовательно сканирует события в глобальной трассе и пытается отыскать предопределенные шаблоны, входящие в дистрибутив системы. В трассе могут встречаться только терминальные события (SEND, RECV и.е.д.). Каждое событие помимо других свойств содержит также временную метку. Шаблоном называется комбинация терминальных событий удовлетворяющих определенным предикатам (обычно предикат включает в себя условия относительно временных меток событий с разных процессов). В системе определены порядка тридцати шаблонов для MPI, OpenMP и SHMEM программ.

3. Описание метода автоматизированного обнаружения шаблонов

Метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI-программах и UPC программах базируется на анализе данных полученных во время исполнения параллельной программы в режиме сбора информации (post-mortem анализе). Для автоматизированного обнаружения шаблонов необходимо вначале получить информацию времени выполнения о критических конструкциях-функциях, которые потенциально могут привести к шаблонам определенных типов. После чего проводится анализ собранной информации на предмет наличия того или иного шаблона. Разрабатываемый подход базируется на использовании свободно распространяемых библиотек из системы Scalasca [6].

Таким образом, алгоритм автоматизированного обнаружения семантических ошибок в параллельных MPI-программах состоит из следующих этапов:

Этап 1. Сбор данных времени выполнения параллельной программы.

Этап 2. Анализ данных полученных на Этапе 1 и выявление шаблонов в параллельной программе.

Этап 3. Создание отчета о выявленных ошибках с привязкой к исходному коду параллельной программы.

Построение трассы параллельного приложения состоит из этапа инструментирования и выполнения инструментированной программы на целевой платформе. Инструментирование программы представляет собой добавление в определенных позициях оригинальной программы вызовы к инструментальной библиотеке. Во время выполнения программы эти вызовы регистрируют наступление определенного события и производят запись в трассу. После этого инструментированная программа переносится на целевую платформу и производится запуск параллельной программы. В результате для каждого процесса программы создается его трасса.

На втором этапе, после получения трассы событий применяется post-mortem анализ - трасса параллельной программы анализируется с целю выявления предопределенных семантических ошибок. Каждому шаблону соответствует определенный критерий (предикат от временной метки события, временной метки соответствующего парного события и.т.д.). Для выявления шаблонов перебираются события из трассы и, при выполнении определенного предиката, регистрируется соответствующий шаблон.

На третьем этапе собранные данные передаются генератору отчетов, который создает итоговый отчет в удобном формате. Итоговый отчет о выявленных ошибках содержит список описателей ошибок. Каждый элемент в списке представляет собой кортеж {type, node, process/thread, file, line, comment,…}, где type – тип ошибки, comment – диагностическое текстовое сообщение об ошибке (при возможности рекомендации). node, process/thread, file, line

определяют адрес узла, идентификатор процесса/потока, имя файла, номер строки в файле, где проявляется данная ошибка.

4. Выявляемые шаблоны в MPI приложениях

Рассмотрим шаблоны с применением коммуникаций точка-точка в параллельных MPI-программах. Пусть F представляет собой коммуникационную функцию MPI. Обозначим через Time_start(F) начальный момент времени непосредственно перед началом функции F. Time_end(F) представляет собой временную метку события после функции F. Обозначим через I_T(pid, p_i, c_j) время простоя процесса с идентификатором pid в результате коммуникации c_j={sendId,recvId} вызванное обнаруженной семантической ошибкой p_i. sendId и recvId представляют собой внутренние идентификаторы посылки и приема соответственно. Пусть ε пороговое значение (на данный момент получено экспериментальным путем).

Замечание. Если I_T (pid, p_i, c_j) < ε, будем считать, что при коммуникации c_j шаблон p_i не проявился.

В текущих реализациях библиотеки MPI (например, MPICH2 [10], MVAPICH2 [11], OpenMPI [12]…) при использовании блокирующих функций пересылок точка-точка используются следующие внутренние протоколы:

а) При размере пересылаемого сообщения меньше чем предопределенная константа используется протокол опережающей посылки. В этом случае пересылка не блокируются. Верхняя граница размера для небольших сообщений отличается в разных реализациях MPI, также как и при разных настройках библиотеки MPI. В реализации MVAPICH2-1.7 при текущих настройках верхняя граница составляет 64KB.
б) При размере пересылаемого сообщения больше границы описанной в пункте а) используется протокол рандеву.

4.1 Шаблоны, связанные с простоем при использовании блокирующих точка-точка коммуникаций

При пересылке сообщений от одного процесса к другому возникают ситуации простоя (idle) либо на одной, либо на другой стороне. Данный эффект на корректность вычислений не влияет, однако он отрицательно отражается на скорости выполнения программы. Устранение этих простоев (при возможности) приведет к увеличению производительности параллельной программы.

Ранняя стандартная посылка. Рассмотрим случай, когда посылка начинается раньше, чем прием (рис. 1). И в этом случае процесс-отправитель теряет время. Критерий шаблона:

Time_start(MPI_Send) < Time_start(MPI_Recv)

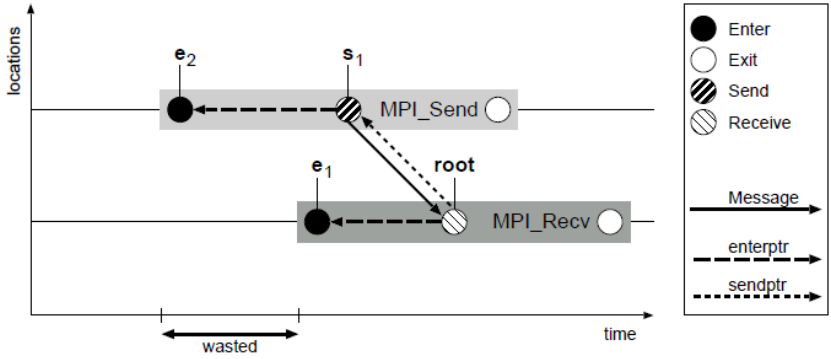

Рис. 1. Поздний прием с передачей MPI_Send.

Замечание. Здесь следует учесть особенности связанные с реализацией MPI (также об этом упоминается в стандарте MPI). Если реализация библиотеки MPI использует протокол опережающей посылки, то функция MPI_Send будет локальной (не блокирующей). Следовательно, простоя в этом случае наблюдаться не будет и необходимо исключить этот случай во избежание ложных срабатываний.

Поздняя стандартная посылка. Рассмотрим случай, когда получение сообщения начинается раньше, чем посылка (рис. 2). И В этом случае процесс-получатель теряет время. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Send)

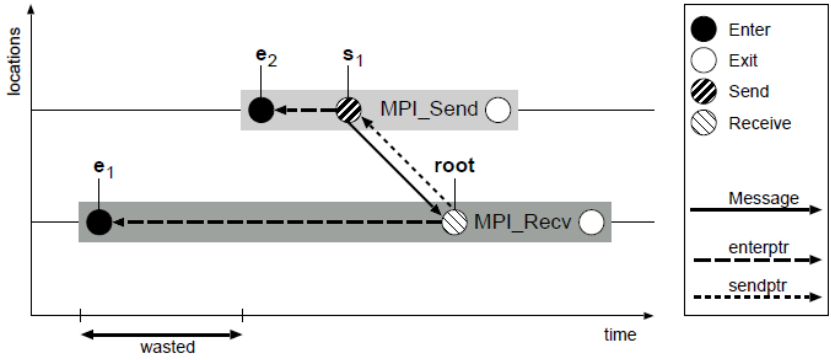

Рис. 2. Ранний прием с передачей MPI_Send.

Поздняя буферизированная посылка. Функция MPI_Bsend является локальной функцией (отправитель копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера). Пусть имеется ситуация как на рис. 2, но вместо

функции MPI_Send используется MPI_Bsend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Bsend. В этом случае принимающая сторона простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Bsend)

$$I\_T(pid, pi, cj) = \begin{cases} 0, & \text{если (Time\_start(MPI\_Bsend)-Time\_start(MPI\_Recv))} < \varepsilon \\ \text{Time\_start(MPI\_Bsend)-Time\_start(MPI\_Recv)}, & \text{иначе} \end{cases}$$

Ранняя буферизированная посылка. Посылка сообщения начинается раньше, чем соответствующий прием. Однако в данном случае процесс-отправитель не простаивает, потому что функция MPI_Bsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

Поздняя синхронная посылка. Пусть имеется ситуация как на рис. 1, но вместо функции MPI_Send используется MPI_Ssend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Ssend. В этом случае принимающая сторона (процесс 1) простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Ssend)

$$I\_T(pid, pi, cj) = \begin{cases} 0, & \text{если (Time\_start(MPI\_Ssend)-Time\_start(MPI\_Recv))} < \varepsilon \\ \text{Time\_start(MPI\_Ssend)-Time\_start(MPI\_Recv)}, & \text{иначе} \end{cases}$$

Ранняя синхронная посылка. Пусть имеется ситуация как на рис. 2, но вместо функции MPI_Send используется MPI_Bsend. Посылка сообщения начинается раньше, чем соответствующий прием. Синхронная посылка использует протокол рандеву – отправляющий ждет, пока сообщение будет принята на стороне принимающего и после чего возвращает управление. То есть из-за того что MPI_Recv начинается позже, отправитель простаивает. Критерий шаблона:

Time_start(MPI_Ssend) < Time_start(MPI_Recv)

$$I\_T(pid, pi, cj) = \begin{cases} 0, & \text{если (Time\_start(MPI\_Recv) - Time\_start(MPI\_Ssend))} < \varepsilon \\ \text{Time\_start(MPI\_Recv) - Time\_start(MPI\_Ssend)}, & \text{иначе} \end{cases}$$

Поздняя посылка по готовности. Пусть имеется ситуация как на рис. 2, но вместо функции MPI_Send используется MPI_Rsend. Прием сообщения начинается раньше, чем соответствующая посылка MPI_Rsend. В этом случае принимающая сторона (процесс 1) простаивает в ожидании. Критерий шаблона:

Time_start(MPI_Recv) < Time_start(MPI_Rsend)

$$I\_T(pid, pi, cj) = \begin{cases} 0, & \text{если (Time\_start(MPI\_Rsend)-Time\_start(MPI\_Recv))} < \varepsilon \\ \text{Time\_start(MPI\_Rsend)-Time\_start(MPI\_Recv)}, & \text{иначе} \end{cases}$$

Ранняя посылка по готовности. Пусть имеется ситуация как на рис. 1, но вместо функции MPI_Send используется MPI_Rsend. Посылка сообщения начинается раньше, чем соответствующий прием. Согласно стандарту MPI [1] данная ситуация является ошибочной. Однако в текущей реализации MPICH2 [10], MVAPICH2 [11] MPI_Rsend отображается на MPI_Send. В результате ошибки не выдается, а сообщение передается получателю. Таким образом, шаблоны ранней посылки по готовности можно разбить на два случая:

а) Размер сообщения меньше определенной константы. Тогда применяется опережающая посылка, и операция MPI_Rsend является локальной. Следовательно, в этом случае простоя не будет и такая ошибка не приводит к снижению эффективности программы.
б) Размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_Recv) приведет к простою процесса отправителя.

На рис. 1 приводится графическое представление выше описанного шаблона. Критерий шаблона:

$$\text{Time\_start(MPI\_Rsend)} < \text{Time\_start(MPI\_Recv)}$$

$$I\_T(pid, pi, cj) = \begin{cases} 0, & \text{если (Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend))} < \varepsilon \\ \text{Time\_start(MPI\_Recv) - Time\_start(MPI\_Rsend)}, & \text{иначе} \end{cases}$$

4.2 Шаблоны, связанные с «неправильным порядком сообщений»

Эффект с «неправильным порядком сообщений» может возникнуть когда в процессе-получателе сообщения ожидаются в одном порядке, а процесс-отправитель посылает сообщения в обратном порядке (рис. 3*Рис.*). Переупорядочив сообщения можно не только добиться ускорения программы, но также потребуется меньший размер буфера для хранения необработанных сообщений.

Если посылка-прием используют протокол рандеву, то очевидно программа попадает в дедлок. Но если посылка локальная (буферизированная посылка, либо обычный send, но размер сообщения маленький и сообщение в реализации MPI отправляется через внутренний MPI буфер), то блокировки не будет, а будет неэффективная организация коммуникаций.

Неправильный порядок с применением MPI_Send. При применении пары функций {MPI_Send, MPI_Recv} учитывая особенности реализации MPI_Send возможны два варианта:

а) При маленьких сообщениях вызовы MPI_Send не блокируются и можно получить семантическую ошибку «неправильный порядок».
б) При больших сообщениях для завершения функции MPI_Send необходимо, чтобы соответствующая функция MPI_Recv была начата. Но поскольку сообщения принимаются в обратном порядке, то данная программа останется в дедлоке.

Рис. 3*Рис.* представляет графическое представление шаблона неэффективного поведения при пересылке сообщений в неправильном порядке с применением функции MPI_Send.

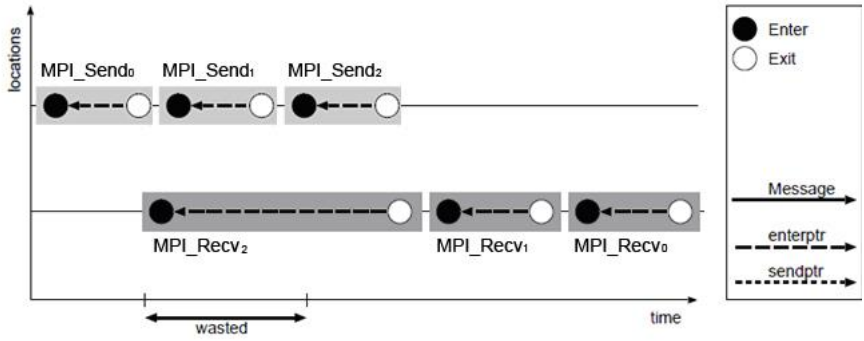

Рис. 3- Шаблон «неправильный порядок» при применении пары функций {MPI_Send, MPI_Recv}.

Ниже представлен критерий шаблона:

$Time\_end(MPI\_Send_0) < Time\_end(MPI\_Send_1) < Time\_end(MPI\_Send_2)$ и $Time\_end(MPI\_Recv_2) < Time\_end(MPI\_Recv_1) < Time\_end(MPI\_Recv_0)$ и $Time\_start(MPI\_Recv_2) < Time\_start(MPI\_Send_2)$

Неправильный порядок с применением MPI_Ssend. Шаблон неправильный порядок сообщений не возможен при применении пары функций {MPI_Ssend, MPI_Recv}, поскольку в этом случае последовательные вызовы MPI_Ssend не встретив соответствующих MPI_Recv-ов заблокируются.

Неправильный порядок с применением MPI_Bsend. Пусть имеется ситуация как на рис. 3, но вместо функции MPI_Send используется MPI_Bsend. Шаблон «неправильный порядок» может возникнуть при использовании пары функций {MPI_Bsend, MPI_Recv }. Ниже представлен критерий шаблона:

Time_end(MPI_Bsend$_0$) < Time_end(MPI_Bsend$_1$) < Time_end(MPI_Bsend$_2$) и Time_end(MPI_Recv$_2$) < Time_end(MPI_Recv$_1$) < Time_end(MPI_Recv$_0$) и Time_start(MPI_Recv$_2$) < Time_start(MPI_Bsend$_2$)

Неправильный порядок с применением MPI_Rsend. Пусть имеется ситуация как на рис. 3*Рис.* , но вместо функции MPI_Send используется MPI_Rsend. Как показали дальнейшие исследования в реализации MVAPICH-а при работе с {MPI_Rsend,MPI_Recv}-ом важно не абсолютное запаздывание соответствующего MPI_Recv-а, а относительный порядок относительно остальных MPI_Recv-ов в очереди сообщений на стороне процесса-получателя. На принимающей стороне (p_1) существует очередь сообщений, где хранятся все сообщения, которые прибыли (либо прибыло служебное сообщение-запрос на начало пересылки), но еще не были обработаны (не было вызова соответствующего MPI_Recv-а). После того как в p_1 был вызван MPI_Recv$_2$ система времени выполнения берет первый элемент в очереди и поскольку MPI_Rsend$_0$ не соответствует MPI_Recv$_2$, то система регистрирует эту ошибку, но выдача ошибки откладывается. Далее подбирается следующий элемент из очереди (MPI_Rsend$_1$) и поскольку соответствие найдено производится прием сообщения в буфер MPI_Rsend$_1$. Пользовательская программа продолжает выполнять инструкции. Если в программе не будет вызова MPI_Recv$_0$, то программа завершится, и ошибки не будет. Если же где то дальше находится MPI_Recv$_0$, то программа аварийно завершается с выдачей соответствующего сообщения. Следовательно, шаблон «неправильный порядок» не применим к случаю, когда используется последовательность передачи сообщений посредством MPI_Rsend-ов и обратная последовательность приема сообщений MPI_Recv-ами.

4.3 Шаблоны, связанные с не-блокирующими точка-точка коммуникациями

Пусть имеется пара вызовов {MPI_Isend,MPI_Wait} в процессе p_0 и {MPI_Irecv, MPI_Wait} в процессе p_1. Рассмотрим семантические ошибки, которые возникают в этом случае процессах pid$_0$ и pid$_1$.

Ожидание на стороне отправителя при использовании {MPI_Isend, MPI_Irecv}. Рассмотрим процесс pid$_0$. В этом случае после вызова функции MPI_Isend управление возвращается в процесс pid$_0$ и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. В трассе событий содержится также временные метки для каждого события, следовательно, разница временных меток событий после и перед вызовом MPI_Wait позволит вычислить время простоя процесса. Критерий шаблона:

$$I\_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$$I\_T(pid_0,p_i,c_j) > 0$$

Помимо определения прогноза можно выдать диагностическое сообщение с оценкой оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait.

$$O\_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если Time\_start(MPI\_Isend) > Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irecv)-Time\_start(MPI\_Isend)}) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Isend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}. При приеме сообщения в процессе pid_1 возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс pid_1 и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. В трассе событий содержится также временные метки для каждого события, следовательно, разница временных меток событий после и перед вызовом MPI_Wait позволит вычислить время простоя процесса. Критерий шаблона:

$$I\_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$$I\_T(pid1,pi,cj) > 0$$

Оценка оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time\_start(MPI\_Isend) < Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Isend)-Time\_start(MPI\_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Ibsend, MPI_Irecv}. В этом случае ошибки не будет, потому что не-блокирующая функция MPI_Ibsend является локальной – функция копирует сообщение в буфер и возвращает управление, а система времени выполнения MPI занимается отправкой сообщения из буфера.

Ожидание на стороне получателя при использовании {MPI_Ibsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Ibsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае шаблон может проявиться только на стороне получателя. По аналогии с

шаблоном при использовании пары **{MPI_Isend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I\_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$I\_T(pid_1,pi,cj) > 0$

Оценка оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time\_start(MPI\_Ibsend)} < \text{Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Ibsend)-Time\_start(MPI\_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Issend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Issend,MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае после вызова не-блокирующей синхронной функции MPI_Issend управление возвращается в процесс pid_0. Если вызов соответствующей функции MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. По аналогии с шаблоном при использовании пары **{MPI_Issend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I\_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$I\_T(pid_0,p_i,c_j) > 0$

Оценка оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O\_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если Time\_start(MPI\_Issend)} > \text{Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irecv)-Time\_start(MPI\_Issend)}) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Issend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Issend, MPI_Irecv}. При приеме сообщения в процессе pid_1 возникает аналогичная ситуация. После вызова функции MPI_Irecv управление возвращается в процесс pid_1 и выполняются вычислительные инструкции, после чего вызывается функция MPI_Wait. Если вызов MPI_Wait был произведен слишком рано, то процесс блокируется и простаивает. По аналогии с шаблоном при использовании пары **{MPI_Issend, MPI_Irecv}** критерий шаблона вычисляется как:

$$I\_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$I\_T(pid1,pi,cj) > 0$

Оценка оптимальной дистанции($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time\_start(MPI\_Issend) < Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Issend)-Time\_start(MPI\_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне отправителя при использовании {MPI_Irsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Irsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. Данный случай является пересечением шаблонов с использованием блокирующих функция {MPI_Rsend,MPI_Recv} и шаблонов ожидание на стороне получателя при использовании {MPI_Isend, MPI_Irecv}. Следовательно, шаблон может проявиться только тогда, когда размер пересылаемого сообщения больше определенной константы. В этом случае будет применяться протокол рандеву, что при поздней инициализации операции приема (MPI_IRecv) приведет к простою процесса отправителя. Таким образом, критерий шаблона представляется формулами:

$$I\_T(pid_0, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$I\_T(pid_0,p_i,c_j) > 0$

Оптимальная дистанция ($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait оценивается формулой:

$$O\_D(pid_0, pi, cj) = \begin{cases} \Delta Ti + T_{send}, & \text{если Time\_start(MPI\_Irsend) > Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irecv)-Time\_start(MPI\_Irsend)}) + \Delta Ti + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irsend, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

Ожидание на стороне получателя при использовании {MPI_Irsend, MPI_Irecv}. На стороне отправителя используется пара функций {MPI_Irsend, MPI_Wait}, на стороне получателя {MPI_Irecv, MPI_Wait}. В этом случае шаблон может возникнуть только на стороне получателя. По аналогии с шаблоном при использовании пары **{MPI_Isend, MPI_Irecv}** критерий шаблона представляется формулами:

$$I\_T(pid_1, pi, cj) = \begin{cases} 0, & \text{если (Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait))} < \varepsilon \\ \text{Time\_end(MPI\_Wait) - Time\_start(MPI\_Wait)}, & \text{иначе} \end{cases}$$

$I\_T(pid_1,pi,cj) > 0$

Оптимальная дистанция ($O\_D(pid,p_i,c_j)$) для вызова MPI_Wait оценивается формулой:

$$O\_D(pid_1, p_i, c_j) = \begin{cases} \Delta T_i + T_{send}, & \text{если Time\_start(MPI\_Irsend) < Time\_start(MPI\_Irecv)} \\ (\text{Time\_start(MPI\_Irsend)-Time\_start(MPI\_Irecv)}) + \Delta T_i + T_{send}, & \text{иначе} \end{cases}$$

где ΔT_i - время выполнения функции MPI_Irecv, T_{send} оценка времени реальной пересылки через коммуникационную сеть.

4.4 Близкая пересылка, передача сообщений

Рассмотрим программу, где в процессе pid_i применяется посылка и прием сообщения с процессом pid_j и операция посылки и приема находятся близко (рис. 4).

```
        if( rank == pid_i )
        {
        int *send_buf = (int *)malloc(sizeof(int) * 12001);
        int *recv_buf = (int *)malloc(sizeof(int) * 12001);

        MPI_Send(send_buf,12001,MPI_INT, pid_j,0,MPI_COMM_WORLD);
…
        MPI_Recv(recv_buf,12001,MPI_INT, pid_j,0,MPI_COMM_WORLD);
        }
        else if( rank == pid_j )
        {
        int * send_buf = (int *)malloc(sizeof(int) * 12001);
        int * recv_buf = (int *)malloc(sizeof(int) * 12001);
              MPI_Status stat;
              MPI_Status stat2;

        MPI_Recv(recv_buf,12001,MPI_INT, pid_i,0,MPI_COMM_WORLD);
…
        MPI_Send(send_buf,12001,MPI_INT, pid_i,0,MPI_COMM_WORLD);
        }
```

Рис. 4- Пример использования тесной посылки и приема сообщений процессами.

Если был найден такой шаблон и логика программы позволяет, то можно объединить функции MPI_Send, MPI_Recv в функцию MPI_Sendrecv, что позволит получить достаточно серьезный выигрыш по времени выполнения. Происходит это из-за того, что в реализациях MPI достаточно хорошо реализована функция MPI_Sendrecv. А также на нижнем уровне современные высокопроизводительные коммуникационные сети (Infiniband [13]) поддерживают дуплексную связь, что позволяет одновременно посылать и получать сообщение одному HCA. И в этом случае вместо того, чтобы pid_i ждал окончания операции MPI_Send и только после этого начал ждать

окончания MPI_Recv, посылка и получение производятся одновременно. Пусть Δ_{SR} некоторая предопределенная константа. Критерий шаблона:

$$(\text{Time\_start(MPI\_Recv)} - \text{Time\_end(MPI\_Send)}) < \Delta_{SR}$$

При нахождении этого шаблона выдается диагностическое сообщение. И пользователь, убедившись, что не существует зависимости по данным для буферов send_buf и recv_buf, может заменить пару вызовов функций {MPI_Send, MPI_Recv} на вызов функции MPI_Sendrecv.

5. Выявляемые шаблоны в UPC приложениях

Набор из двадцати шаблонов неэффективного поведения был разбит на четыре группы. В первую группу входят шаблоны для обнаружения задержек в коллективных операциях передачи данных, так же называемых операциями релокализации в языке UPC [2]. В первую группу входят восемь шаблонов. Во вторую группу вошли семь шаблонов, связанных с операциями явной и неявной синхронизации, которые присутствуют практически во всех языках параллельного программирования. Третья группа состоит из трех шаблонов, выполняющих анализ односторонних и коллективных операций передачи данных.

Последняя четвертая группа, состоящая из двух шаблонов, связана с моделью параллельного программирования «Главный-Подчиненные», в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков и работа распределяется между ними.

5.1 Блокировки в коллективных операциях передачи данных

В отличие от односторонних коммуникационных операций *upc_put()/upc_get()*, в коллективных операциях присутствуют сложные зависимости по данным, требующие синхронизации. Если программа написана неудачно и четкая координация между потоками нарушена, то неизбежно возникнут дополнительные задержки. Чтобы свести эту проблему до минимума, в UPC для всех операций релокализации были выделены три типа синхронизации, которые указываются последним аргументом в вызове функций отдельно для синхронизации на входе в операцию и на выходе из нее. С учетом этих особенностей, были сформулированы восемь шаблонов для анализа следующих операций релокализации данных: *upc_all_broadcast(), upc_all_scatter(), upc_all_gather(), upc_all_gather_all(), upc_all_exchange(), upc_all_permute(), upc_all_reduce() и upc_all_prefix_reduce()*.

Шаблон «Синхронизация на входе в коллективную операцию». Данный шаблон справедлив для всех операций релокализации, в которых используется тип синхронизации UPC_IN_ALLSYNC. Предположим для данного примера, что это операция широковещательной рассылки upc_all_broadcast(). В данном

случае каждый поток обязан дождаться на входе всех остальных. Когда потоки входят в операцию релокализации в разные моменты времени - это вносит нежелательные накладные расходы на синхронизацию. Данный шаблон не встречается в программной модели передачи сообщений и характерен только для языка UPC. Это объясняется тем, что стандарт MPI не накладывает строгих ограничений на порядок входа потоков в коллективную операцию. Будут ли потоки ожидать друг друга, как при использовании барьерной синхронизации, или же будет использован более оптимальный алгоритм, определяется реализацией, и этого нельзя отследить. В случае с UPC, синхронизация указывается программистом явно, что позволяет сформулировать новые более точные шаблоны.

Шаблон «Синхронизация на выходе из коллективной операции». Данный шаблон справедлив для всех операций релокализации, в которых используется тип синхронизации UPC_OUT_ALLSYNC. Шаблон также характерен только для языка UPC. Как и в шаблоне с синхронизацией на входе, все потоки обязаны дождаться друг друга, только теперь на выходе из операции.

Шаблон «Поздняя рассылка». Данный шаблон возникает при использовании синхронизации UPC_IN_MYSYNC на входе в операции один ко многим. К ним относятся такие операции языка UPC, как: upc_all_broadcast() и upc_all_scatter(). Если поток, рассылающий данные, входит в операцию позднее потоков, принимающих данные, то последние обязаны приостановить свое выполнение. Шаблон отражает время, потерянное в результате возникновения такой ситуации.

Шаблон «Ранняя сборка». Данный шаблон присущ операциям, выполняющим сборку данных, таким как upc_all_gather() и upc_all_reduce(), если для синхронизации на входе используется UPC_IN_MYSYNC. Данный шаблон аналогичен шаблону «Поздняя рассылка» за тем исключением, что причиной его возникновения является поток получатель данных, в том случае, если он входит в операцию позднее других.

Шаблон «Ранняя префиксная редукция». Данный шаблон уникален для операции префиксной редукции upc_all_prefix_reduce(). Возникает в случае использования синхронизации UPC_IN_MYSYNC. В данной операции результат редукции в потоке n зависит от редукции, выполненной в потоке n-1. Если хотя бы один из потоков 0..n-1 не вошел в операцию, поток n обязан ждать.

Шаблон «Синхронизация на входе в коллективную операцию многие ко многим». Данный шаблон в отличие от шаблона «Синхронизация на входе в коллективную операцию» возникает при использовании синхронизации UPC_IN_MYSYNC и только в операциях, которые отправляют данные от многих потоков ко многим, к которым относятся upc_all_gather_all() и upc_all_exchange(). Если в первом случае все потоки ждут друг друга, то в данном шаблоне они имеют право работать с данными потоков, которые уже

вошли в коллективную операцию. Тем не менее, если некоторые потоки еще не вошли в операцию, а с остальными обмен уже произошел, то потоку придется остановиться. Шаблон описывает потерянное в такой ситуации время. Искомые данные и потерянное время находятся аналогично первому случаю. Важно лишь отметить, что данный шаблон не достаточно точен, так как часть времени, попадающего в категорию потерянного, является полезным. Это время, потраченное на обмен с потоками, уже вошедшими в операцию. На практике невозможно определить, какую часть времени поток обменивался данными, а какую часть находился в ожидании. Поэтому данный шаблон можно рассматривать лишь как подозрение на неэффективное поведение. Если программа затратила значительную часть времени в такой ситуации, то нужно более подробно взглянуть на проблему.

Шаблон «Синхронизация на выходе из коллективной операции многие ко многим». Данный шаблон аналогичен предыдущему шаблону, с тем отличием, что синхронизация и соответственно потеря времени происходит на выходе из операции.

Шаблон «Ожидание внутри коллективной операции». Данный шаблон возникает в операциях upc_all_broadcast(), upc_all_scatter(), upc_all_gather() и upc_all_reduce() при использовании типа синхронизации UPC_IN_MYSYNC. Этот шаблон является дополнением к шаблонам «Поздняя рассылка» и «Ранняя сборка». При выполнении коллективной операции в языке UPC может возникнуть ситуация, когда поток находится в коллективной операции один. Это происходит, например, если поток входит в операцию первым, либо выходит последним или если часть потоков уже выполнили вычисления и вышли, а некоторые потоки еще не успели дойти до коллективной операции. Для нахождения потерянного времени в общем случае, достаточно из времени выполнения операции корневым потоком, вычесть время выполнения операции другими потоками, пересекающееся с корневой.

5.2 Операции синхронизации

В языке UPC явная синхронизация представлена операциями *upc_barrier(), upc_notify()/upc_wait() и upc_fence()*. В первом случае программа блокируется до тех пор пока все потоки не достигнут операции *upc_barrier()*, во втором случае, который называется барьером с расщепленной фазой, синхронизация разбивается на два этапа и накладывает менее строгие требования к синхронизации. Операция *upc_fence()* заставляет поток дождаться завершения всех неявных односторонних операций обращения к памяти других потоков. Кроме того, неявная синхронизация потоков происходит при выполнении атомарных операций работы с памятью *bupc_atomic*()*, а так же в операции динамического выделения памяти *upc_all_alloc()*.

Шаблон «Конкуренция за блокировку». Данный шаблон возникает, если один из потоков пытается захватить блокировку, которой в данный момент владеет другой поток. Шаблон является типичным для модели общей памяти.

Однако он хорошо ложится на программную модель PGAS, так как в ней используется модель распределенной общей памяти. Если в момент вызова функции upc_lock() блокировка уже захвачена, то поток останавливает свое выполнение до освобождения блокировки. Время, потраченной в данной ситуации, соответствует времени, проведенном в функции upc_lock().

Шаблон «Ожидание в барьере». Данный шаблон возникает в ситуациях, когда для синхронизации потоков в программе используется барьерная синхронизация. Если в приложении встретился барьер, то все потоки обязаны остановиться и дождаться друг друга. Шаблон фиксирует время, затраченное на выполнение барьерной синхронизации.

Шаблон «Завершение барьера». Это достаточно специфический шаблон, в том смысле, что в нормальной ситуации все потоки должны выходить из барьера в один и тот же момент времени. Любое, даже незначительное время, проведенное в данном шаблоне, может означать неэффективность реализации PGAS языка, либо помехи со стороны других процессов, работающих на том же расчетном узле.

Шаблон «Ожидание в операции динамического выделения памяти». Данный шаблон возникает в операции upc_all_alloc(). В языке UPC присутствуют ряд операций для динамического выделения общей памяти. Если память выделяется коллективно при помощи функции upc_all_alloc(), то возникают требования к синхронизации. В действительности выделение памяти происходит в потоке 0, после чего результат операции рассылается по всем потокам, аналогично операции upc_all_broadcast(). Если поток 0 входит в операцию позже других, то остальные потоки обязаны ждать рассылки результата, поэтому описание данного шаблона аналогично шаблону «Поздняя рассылка».

Шаблон «Ожидание в решетке». Если в момент выполнения операции upc_fence() существуют незавершенные односторонние операции обращения данного потока к памяти других потоков, то его выполнение блокируется до их завершения. Операция upc_fence(), вызванная в одном потоке, не влияет на другие. Так же, на блокировку выполнения потока операцией upc_fence() не влияют обращения других потоков в его память. Потерянное время расчитывается, как время выполнения операции upc_fence().

Шаблон «Ожидание в расщепленном барьере». Расщепленный барьер отличается от обычного барьера upc_barrier() тем, что он выполняется в два этапа. Когда поток завершает выполнение операций, требующих синхронизации, он вызывает upc_notify(). После чего поток может продолжать выполнять часть локального для потока программного кода, не требующего синхронизации. Когда его выполнение так же завершено, поток выполняет операцию upc_wait(), которая блокирует выполнение до тех пор, пока все остальные не выполнят операцию upc_notify(). Расщепленный барьер накладывает менее строгие требования на синхронизацию, но так же вызывает

блокировку потоков. Потерянное в шаблоне время рассчитывается, как время выполнения операции upc_notify().

Шаблон «Ожидание в атомарной операции». Атомарность выполнения даже таких операций, как увеличение значения переменной на величину другой переменной, часто не гарантируется в параллельном программировании. Такая операция как x += y трансформируется в несколько операций: READ(x) => t1, READ(y) => t2, t1 + t2 => t3, t3 => WRITE(x). Ни что не мешает втором потоку изменить значение y, после того, как операция уже начала выполнятся и значение x было уже считано в первом потоке. Спецификация языка UPC версии 1.3 вводит новый набор функций, которые позволяют атомарно выполнять ряд составных операций над переменными, например таких как «сравнение и замена». Для обеспечения атомарности этих операций, используется синхронизация. Данный шаблон описывает время, потерянное на синхронизацию в атомарной функции.

5.3 Операции передачи данных

Шаблоны из данной группы позволяют обнаружить «горячие» точки и узкие места программы, с точки зрения объема передаваемых между потоками данных.

Шаблон «Частая коммуникация». Данный шаблон позволяет идентифицировать блоки кода программы генерирующие наибольшее количество операций коммуникации. Так как коммуникации являются наиболее медленными операциями параллельного приложения, обнаружение «горячих» точек программы позволяет выявить места исходного кода, наиболее выгодные с точки зрения оптимизации.

Шаблон «Большие сообщения». В шаблоне анализируется размер отправленных и полученных сообщений, в результате выполнения коллективных и односторонних коммуникационных операций. Использование более крупных посылок данных в одном из блоков программы, по сравнению с другими, может создать «узкое» место в работе приложения. Шаблон позволяет обнаружить подобные операции передачи данных, которые в дальнейшем могут быть оптимизированы, например, путем использования неблокирующих коммуникационных операций.

Шаблон «Несбалансированные коммуникации». В шаблоне сравнивается время, проведенное разными потоками в одной и той же коммуникационной операции. В лучшем случае это время должно совпадать во всех потоках. Если потоки затрачивают существенно разное время в одной и той же операции, то это означает, что присутствует разбалансировка нагрузки, которая требует дальнейшей оптимизации.

5.4 Модель «Главный-Подчиненные»

Задача шаблонов из данной группы выявление возможной разбалансировки нагрузки между потоками и связанную с этим потерю процессорного времени.

Шаблон «Медленные подчиненные». Часто в параллельных программах используется подход главный-подчиненные, когда один из потоков выполняет генерацию данных, их отправку подчиненным потокам и последующую сборку результатов. Шаблон описывает ситуацию, когда главный поток блокирует свое выполнение в ожидании результатов от подчиненных потоков, вместо выполнения полезной работы.

Шаблон «Перегруженный главный». Данный шаблон описывает противоположную ситуацию, когда подчиненные ожидают данные от главного потока, вместо выполнения вычислений. Главный поток может затрачивать излишне большое время на сбор результатов, либо не успевать генерировать новые данные для расчетов. В обоих случаях имеет место разбалансировка нагрузки между потоками, являющаяся потенциальным местом для оптимизации.

6. Заключение

Инструменты анализа производительности оказывают помощь разработчикам параллельных приложений в оптимизации параллельного кода и ускоряют процесс доводки параллельных приложений. Большинство существующих инструментов основаны на методах ручного анализа. В статье приведен метод автоматизированного обнаружения шаблонов неэффективного поведения в параллельных MPI приложениях и UPC программах. Это становится особенно актуальным в условиях постоянного роста количество ядер в современных кластерах и увеличения сложности параллельных комплексов. В работе приводится описание шаблонов неэффективного поведения для MPI-программ и шаблонов разработаных для языка Unified Parallel C, который является одним из представителей программной модели Partitioned Global Address Space (PGAS).

Список литературы

[1]. 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.

[2]. W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. //14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[3]. Sameer S. Shende Allen D. Malony. “The Tau Parallel Performance System”, International Journal of High Performance Computing Applications, Volume 20 , Issue 2, Pages: 287 – 311, May 2006.

[4]. L. Li and A.D. Malony, “Model-Based Performance Diagnosis of Master-Worker Parallel Computations,” Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.
[5]. H. Su, M. Billingsley III, and A. George. "Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications," International Journal of High-Performance Computing Applications, Vol. 24, No. 4, Nov. 2010, pp. 485-510.
[6]. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Bernd Mohr. The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.
[7]. Felix Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003, ISBN 3-00-010003-2.
[8]. Wolf, F., Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11) (2003) 421–439.
[9]. Wolf, F., Mohr, B., Dongarra, J., Moore, S. Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004).
[10]. MPICH. http://www.mpich.org
[11]. MVAPICH. http://mvapich.cse.ohio-state.edu.
[12]. OpenMPI. http://www.open-mpi.org.
[13]. Infiniband. http://www.infinibandta.org.

Research and development of inefficiency patterns in MPI, UPC applications

M.S. Akopyan, N.E. Andreev
ISP RAS, Moscow, Russia, Thomas Duryea Consulting, Melbourne, Australia
manuk@ispras.ru, andreev.nikita@gmail.com

Abstract. Most of developed tools for analysis for various libraries (MPI, OpenMP) and languages for parallel programming use low level approaches to analyze the performance of parallel applications. There are a lot of profiling tools and trace visualizers which produce tables, graphs with various statistics of executed program. In most cases developer has to manually look for bottlenecks and opportunities for performance improvement in the produced statistics and graphs. The amount of information developer has to handle manually, increase dramatically with number of cores, number of processes and size of problem in application. Therefore new methods of performance analysis fully or partially handling output information will be more beneficial. To apply the same analysis tool to various parallel paradigm (MPI applications, UPC programs) paradigm-specific inefficiency patterns has been developed. In this paper code patterns resulting in performance penalties are discussed. Patterns of parallel MPI applications for parallel computing systems with distributed memory as well as for parallel UPC programs for systems with partial global address space (PGAS) are considered. A method for automatic detection of inefficiency patterns in parallel MPI applications and UPC programs is proposed. It allows to reduce the tuning time of parallel application.

Keywords: parallel programming, semantic errors, inefficiency patterns, MPI, UPC

References

[1]. 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.

[2]. W. Chen, C. Iancu, K. Yelick. Communication Optimizations for Fine-grained UPC Applications. 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005.

[3]. Sameer S. Shende Allen D. Malony. "The Tau Parallel Performance System", International Journal of High Performance Computing Applications, Volume 20 , Issue 2, Pages: 287 – 311, May 2006.

[4]. L. Li and A.D. Malony, "Model-Based Performance Diagnosis of Master-Worker Parallel Computations," Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.

[5]. H. Su, M. Billingsley III, and A. George. "Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications," International Journal of High-Performance Computing Applications, Vol. 24, No. 4, Nov. 2010, pp. 485-510.

[6]. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Bernd Mohr. The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.

[7]. Felix Wolf. Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003, ISBN 3-00-010003-2.
[8]. Wolf, F., Mohr, B. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11) (2003) 421–439.
[9]. Wolf, F., Mohr, B., Dongarra, J., Moore, S. Efficient Pattern Search in Large Traces through Successive Refinement. In: Proc. European Conf. on Parallel Computing (Euro-Par, Pisa, Italy), Springer (2004).
[10]. MPICH. http://www.mpich.org
[11]. MVAPICH. http://mvapich.cse.ohio-state.edu.
[12]. OpenMPI. http://www.open-mpi.org.
[13]. Infiniband. http://www.infinibandta.org