ИССЛЕДОВАНИЕ МАСШТАБИРУЕМОСТИ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ ДАННЫХ ТРАССИРОВЩИКА И АНАЛИЗАТОРА INTEL TRACE ANALYZER AND COLLECTOR НА ПРИМЕРЕ МОДЕЛИ АТМОСФЕРЫ NH3D

А.С. Антонов, А.М. Теплов

1. Введение

Параллельное программирование всегда связано с рядом сложностей, с которыми не сталкиваются программисты, создающие последовательные программы. Но только параллельные вычисления позволяют добиваться максимальной производительности, которая так необходима во многих современных научных расчётах. Современный численный эксперимент, проводимый на огромном суперкомпьютере, генерирует огромные объемы данных и может проходить до нескольких суток.

Вопрос рационального использования вычислительных систем, способных справиться с такими вычислениями, стоит очень остро. Он напрямую связан с масштабируемостью – свойством параллельных программ, характеризующим зависимость показателей эффективности работы параллельной программы от числа использованных процессоров. В данной статье описана методика исследования масштабируемости параллельной программы с использованием инструментария для анализа эффективности приложения Intel Trace Analyzer and Collector на примере атмосферной модели NH3D.

2. Масштабируемость параллельных программ

Для оценки качества программ введён ряд показателей эффективности параллельной программы[1], например:

Ускорение (*speedup*) $S = T_1/T_p$, где T_p — время исполнения распараллеленной программы на p процессорах, T_1 — время исполнения исходной последовательной программы.

Вычислительная сложность задачи W — количество основных вычислительных шагов лучшего последовательного алгоритма, необходимых для решения задачи на одном процессоре. W является некоторой функцией от размера входных данных программы. Если для простоты предположить, что каждый основной вычислительный шаг выполняется за единицу времени, то получим $W = T_1$.

Эффективность $E = S/p$ определяет среднюю долю времени выполнения параллельного алгоритма, в течение которого процессоры реально используются для решения задачи.

Стоимость (cost) вычислений $C = pT_p$.

$T_0 = pT_p - T_1$ — суммарные *накладные расходы (total overhead)*. При увеличении p значение, как правило, возрастает.

В идеальном случае приложение ускоряется линейно, а в некоторых случаях, при использовании аппаратных особенностей системы, можно получить и суперлинейное ускорение.

Если T_1 фиксировано, то при увеличении числа процессоров p эффективность E, как правило, уменьшается за счёт роста накладных расходов T_0. Получение большого ускорения за счёт большого числа процессоров обычно приводит к снижению эффективности. А если фиксировано число процессоров p, то эффективность E зачастую можно повысить, увеличивая вычислительную сложность решаемой задачи W (а значит, и время T_1).

Отношение ускорения или эффективности к объёму используемых ресурсов или размеру задачи характеризует то, насколько рационально программа способна использовать параллельную вычислительную систему. Для этого вводится понятие *масштабируемости* параллельной программы. Масштабируемость – свойство программы, определяющее зависимость её ускорения или эффективности, получаемых на вычислительной системе, от объёма использованных для этого ресурсов и размера задачи. Масштабируемость является одной из основных характеристик параллельной программы. Она позволяет получить представление о работе параллельной программы на данной вычислительной системе.

Исследовать масштабируемость параллельных программ исключительно важно, потому что использование мощных суперкомпьютеров оправдано только в случае, если программа эффективно распределяет вычисления. Зная масштабируемость, можно сделать выводы о работе параллельной программы и определить наиболее узкие места связки параллельная программа — вычислительная система, снижающие производительность вычислительного процесса. Необходимо учитывать многие факторы, такие как архитектура вычислительной системы, алгоритм, реализация в коде и использование данных. На основании данных о масштабируемости можно сделать выводы о необходимости оптимизации программы и причинах возникающих проблем.

Масштабируемость связки вычислительная система – параллельная программа состоит из масштабируемости каждого уровня этой связки: аппаратных ресурсов системы, распределения данных, алгоритмов, программного обеспечения (библиотеки, компиляторы), программы на данной системе.

Масштабируемость параллельных программ можно подразделить на несколько основных типов:

- *Сильная масштабируемость* (*strong scaling*) — зависимость производительности от количества процессоров p при фиксированной вычислительной сложности задачи (W = const).
- *Масштабируемость вширь (wide scaling)* – зависимость производительности от вычислительной сложности задачи W при фиксированном числе процессоров (p=const)
- *Слабая масштабируемость* (*weak scaling*) — зависимость производительности от количества процессоров p при фиксированной вычислительной сложности задачи в пересчёте на один узел (W/p = const).

3. Подходы к изучению масштабируемости

Существует несколько подходов к изучению масштабируемости программ. Каждый подход специфичен для своей области исследований. При выборе метода изучения масштабируемости необходимо учитывать цели исследования и интересующие свойства.

Методы исследования масштабируемости по способу получения данных можно разделить на две группы: *аналитические[2][4]* и *эмпирические[3]*.

Эмпирический подход – это анализ изменения метрик эффективности реальной программы, основывающийся на данных, полученных при многократном выполнении программы на конкретной вычислительной системе. Этот подход используется наиболее часто.

Аналитический подход – это получение необходимых характеристик программы с помощью моделирования хода её выполнения по заданному принципу на менее мощной вычислительной системе и построения прогноза масштабируемости данной программы на целевой вычислительной системе.

Аналитический подход подразумевает один из видов моделирования:

- алгебраический анализ;
- синтаксический разбор исходного текста программы;
- абстрактная интерпретация;
- симулирование выполнения.

При комбинировании методов эмпирического и аналитического исследования получают так называемые *смешанные* методы исследования масштабируемости.

Из смешанных методов выделяют *квазианалитический* метод**,** основанный на моделировании объёма вычислений на каждый узел и запуске на целевой системе. Точность его результатов сильно зависит от оценки объёма вычислений каждого процессора. Этот метод довольно прост в реализации и не требует вычислений в таком объёме, как эмпирический. Такой подход требует большего внимания к моделированию коллективных операций. Это связано с тем, что операции типа точка-точка на целевой конфигурации компьютера будут выполняться столько же времени, а время выполнения коллективных обменов будет зависеть от числа реально участвующих процессов.

Разные методы представляют результаты, которые говорят о масштабируемости в несколько различном контексте. Поэтому при выборе метода исследования необходимо определить, в каком контексте масштабируемость наиболее интересна.

Эмпирические методы требуют значительных затрат ресурсов и времени для сбора данных, но позволяют быстро получить «предсказание» масштабируемости. Аналитические методы для получения данных требуют больше времени на построение модели, а также иногда существенного времени работы модели на инструментальном компьютере. Очень сложную по структуре программу, использующую большое количество модулей и библиотек, проще исследовать эмпирически, чем анализировать полный ход её выполнения, но эмпирические методы применимы только при доступности целевой вычислительной системы.

Построение модели программы возможно только при условии полного и детального знания алгоритма работы и реализации. Для построения качественной модели требуется использование специального инструментария и аккуратный перенос на модель всех деталей работы программы. Для этого может потребоваться существенный объём оперативной памяти. К существенным преимуществам такого подхода нужно отнести возможность получения данных о масштабируемости приложения без привязки к конкретной вычислительной системе и/или схеме распределения данных. Это особенно важно, если масштабируемость исследуют на стадии разработки параллельной программы. В данном случае моделирование может позволить добиться наибольшей эффективности и масштабируемости.

4. Категории инструментов для исследования эффективности и масштабируемости параллельных приложений

Существующие инструменты для исследования масштабируемости программ можно разделить на несколько групп:

- инструменты для анализа показателей эффективности работающей программы (профилировщики, трассировщики, средства пакетного запуска, средства для инструментирования кода);
- инструменты для моделирования работы реального приложения (утилиты, моделирующее время выполнения на основе реальных трасс, исходного кода, ручного сбора данных, других моделей);
- инструменты для моделирования потенциальной масштабируемости разрабатываемого приложения (моделируют масштабируемость разрабатываемого приложения на предполагаемой архитектуре).

Наибольший практический интерес для нас представляют инструменты первой группы, потому что позволяют определить проблемы с эффективностью и причины их возникновения для уже существующих приложений.

Профилирование – процесс получения данных, которые обобщают поведение программы: сколько раз была вызвана некоторая функция; время выполнения функции; дерево или граф вызовов в программе; значения аппаратных счетчиков и т.п. Таким образом, профилирование помогает определить «узкие места» и «горячие точки» в программе. Осуществив профилирование, можно выявить основные части программы, работающие значительную часть общего времени выполнения. Оперируя этими данными, можно не тратить время на оптимизацию того кода, который незначительно влияет на производительность всего приложения. Профилирование требует низких накладных расходов по сравнению со сбором трасс и призвано дать пользователю ответ на вопрос, как работает его приложение в текущем виде. Оно может реализовываться через:

- семплирование – периодические прерывания ОС или прерывания по значению аппаратных счетчиков;
- инструментирование – вставка в программу дополнительного кода, измеряющего производительность.

Сбор трасс подразумевает запись потока событий: вход/выход в какой-то участок кода (функцию, цикл и т.д.); взаимодействие процессов (нитей) — приём/передача сообщений и т.д., запись того, в каком процессе/нити и когда случилось каждое событие. Событие обычно записывается в виде метки, номера процесса (нити), типа события и данных о событии. Трасса позволяет анализировать производительность и корректность программы с точки зрения того, какие процессы происходят в каждый конкретный момент её выполнения. Профиль программы может быть восстановлен по её трассе, но основной акцент все же делается именно на типе происходящих событий и их связях. Таким образом, на трассе фиксируются обмены данными между процессами и коллективные операции. Обычно трассировка требует инструментирования кода для получения более подробных данных об интересных нам событиях и об интересующих участках кода программы.

Intel Trace Analyzer and Collector – инструмент, позволяющий проводить сбор трасс с последующим анализом визуализированной трассы. Он состоит из двух основных составляющих: Trace Collector, осуществляющий сбор трасс, и Trace Analyzer, осуществляющий визуализацию и анализ. Поддерживается сбор трасс параллельных приложений на языках C/C++, Java, Fortran, использующих MPI, явные и неявные нити (OpenMP) или гибридную модель (MPI+OpenMP). Существуют версии для Linux и Microsoft Windows для архитектур IA32, Intel64. Данный инструмент проводит анализ трассы приложения, в котором позволяет определять дисбаланс вычислительной нагрузки в программе. С помощью него можно симулировать поведение программы при различных условиях, например, при очень медленных коммуникациях. Он также позволяет определить тупиковые ситуации в программе и ошибки в использовании типов данных, буферов, коммуникаторов, операций точка-точка и коллективных операций.

5. Описание методики исследования масштабируемости

Исследование масштабируемости модели проходило в несколько этапов.

На первом этапе производился сбор данных о работе приложения и о том, какие данные могут быть использованы для оценки скорости работы. Для этого по выбранным критериям собирались данные о работе приложения в начальном виде. Необходимо провести сбор данных таким образом, чтобы свести к минимуму накладные расходы и минимизировать влияние факта сбора данных на полученный результат. Производился запуск задачи фиксированного размера на разном числе используемых процессоров. Это необходимо для определения начальной картины масштабируемости приложения. Полученные данные говорят нам о наличии проблем масштабируемости и необходимости более глубокого анализа.

Измерялось время выполнения различных частей программы. Изначально в силу итерационной структуры работы алгоритма обработки данных был предложен метод измерения числа выполненных итераций за фиксированное время, но от него пришлось отказаться из-за того, что на этапе инициализации модели время выполнения различных процедур отличалось довольно сильно. Это влияло на число выполненных итераций, и потому более точным оказалось измерение среднего времени итерации отдельно от времени выполнения инициализации.

Исследуемая программа запускалась на разных конфигурациях вычислительной системы. Начиная с минимальной конфигурации число использованных процессоров пошагово увеличивалось.

После каждого запуска анализировались собранная статистика работы приложения. Данные фиксировались в логе работы. Из полученных данных о времени работы вычисляли ускорение работы и эффективность. На следующем этапе приложение инструментировалось для профилирования и определения ускорения различных частей кода. Проводилась еще одна серия запусков, в которой собирались данные о работе отдельных частей программы. Те участки, время которых росло с ростом числа процессоров, рассматривались более детально на предмет причин проблем масштабируемости и путей повышения эффективности.

При использовании Intel Trace Analyzer and Collector для углубленного анализа, данные собирались схожим образом. После каждого запуска, собранная трасса конвертировалась и архивировалась для дальнейшего анализа.

По времени выполнения отдельных частей программы вычислялись их характеристики, такие как ускорение и эффективность работы.

6. Описание модели NH3D

Исследование масштабируемости параллельного приложения проводилось на примере трёхмерной гидротермодинамической негидростатической модели атмосферы[5], разрабатываемой в НИВЦ МГУ имени М.В.Ломоносова. В модели используется параметризация основных физических процессов (радиация, турбулентность, тепловлагоперенос в деятельном слое суши и водных объектов).

Программа написана на языке Фортран с использованием технологии параллельного программирования MPI. Общий объём кода составляет примерно 75 тысяч строк. В основе параллельной реализации модели для многопроцессорных систем с распределенной памятью лежит двумерное разбиение трёхмерной расчётной области. Локальные массивы на отдельных MPI-процессах содержат поля переменных модели только в соответствующей подобласти. Для решения уравнений гидротермодинамики в модели реализована явная схема по времени («чехарда»), что позволяет при расчёте всех полей по эволюционным уравнениям применять пересылки элементов массивов только на границах подобластей.

Модель построена на пошаговом преобразовании данных и выводе получаемых результатов в файл. Почти все время выполнения программного кода приходится на цикл интегрирования уравнений модели по времени main_loop в головной программной единице nh3d_main. Подпрограммы, вызываемые в этом цикле и реализующие компоненты численного алгоритма, не имеют логических ветвлений и выполняются детерминировано.

Такая структура программы позволила существенно упростить анализ производительности и профилирование, потому что на каждом шаге модели выполняются идентичные операции. Для уменьшения объёмов собираемой информации и упрощения анализа в целом число шагов модели было сведено к минимуму, который давал достаточную статистику работы.

Цель исследования заключалась в оптимизации ключевых процедур исходного кода приложения, позволяющей добиться повышения масштабируемости этих процедур на больших конфигурациях вычислительной системы. Работа состояла из нескольких этапов классической схемы оптимизации приложения:

1) Профилирование приложения с выделением узких мест (процедур) при использовании большого количества вычислительных ядер. Профилирование проводилась на двух уровнях. На первом уровне диагностики отдельно оценивалось время выполнения приложения в целом и процедур библиотеки MPI. Минимальной конфигурацией была выбрана конфигурация с 4 процессорами. На меньших конфигурациях работали другие логические ветки кода. Для этого использовался пакет ITAC (Intel Trace Analyzer and Collector). На втором уровне производилась оценка времени выполнения отдельных процедур приложения. Для этих целей проводилось несколько типов профилирования. Изначально для сбора общей статистической информации проводилось профилирование с помощью вывода среднего времени работы каждой процедуры в файл на каждом шаге по времени модели. По результатам профилирования выбирались процедуры, время выполнения которых становилось критичным для времени выполнения одного шага модели в целом. При выборе процедур принимались во внимание особенности структуры кода, численные алгоритмы и организация MPI-обменов в данном приложении. На основе собранных данных были выявлены наиболее узкие места работы приложения. На основе собранных данных было проведено более детальное инстурментирование исходного кода для профилирования трассы приложения для анализа с помощью Trace Analyzer.

2) Оптимизация выбранных процедур проводилась, исходя из полученных данных на всех этапах анализа. Так, на первом этапе анализа была выявлена проблема с выводом в файл. Модель регулярно записывала результат своей работы в файл. При этом использовался только один активный процесс, и все остальные процессы ожидали окончания записи. Проблема заключалась в том, что процесс выводил существенный объём данных, время вывода было существенным даже на небольших конфигурациях вычислительной системы и абсолютно не уменьшалось с увеличением конфигурации. Поэтому на больших конфигурациях системы время вывода в файл составляло очень существенную долю времени работы модели в

целом. Регулярность вывода не была критичной для работы алгоритма и потому вывод в файл решили оставить только в конце всей работы. В перспективе планируется переход на параллельный вывод данных.

Второй оптимизацией, основанной на результатах профилирования, стало использование модернизированного варианта двух процедур, используемых на каждом шаге модели. Профилирование показало, что на больших конфигурациях вычислительной системы основу времени каждого шага составляло время выполнение двух процедур: MOMENT_MPI и THERMO_MPI. Процедура MOMENT_MPI решает три уравнения движения, а процедура THERMO_MPI — уравнение притока тепла и уравнения переноса излучения в атмосфере. Ускорение обеих процедур переставало расти при увеличении числа процессоров больше 25. На конфигурациях около 100 процессоров время увеличилось в 10 раз для процедуры MOMENT_MPI и в 40 раз для процедуры THERMO_MPI по сравнению со временем их выполнения на 25 процессорах. Более детальное профилирование показало, что процедуры используют адаптацию одного и того же алгоритма усреднения данных на границе области SPONGE, что и приводило к таким результатам. Отключение его использования в настройках работы модели позволило существенно уменьшить время одного шага модели и масштабируемость всего приложения в целом. После этого ускорение шага модели продолжало расти до конфигураций вычислительной системы, содержащих в полтора раза больше процессоров.

7. Исследование модели NH3D с помощью ITAC

При исследовании модели NH3D использовался пакет ITAC. На первом этапе исследования анализ трассы позволил исследовать общий характер поведения параллельной программы на вычислительной системе. В собранной трассе без пользовательского инструментирования исходного кода программы отображались два класса функций: время, затраченное на вызовы функций MPI, и время выполнения кода приложения. Даже без дополнительного инструментирования исходного кода приложения можно определить ряд узких мест работы программы. Визуально легко определить наличие последовательных участков программы. Такие участки соответствуют выводу программы в файл, а также процессу инициализации данных приложения. Они ухудшают масштабируемость всего приложения в целом, потому что не получают никакого ускорения на любом масштабе вычислительной системы. На собранной трассе можно установить, какие функции MPI увеличивают разбалансировку работы приложения. Это также приводит к ухудшению масштабируемости приложения, потому что при синхронизации часть процессов будет вынуждена ждать завершения выполнения других процессов.

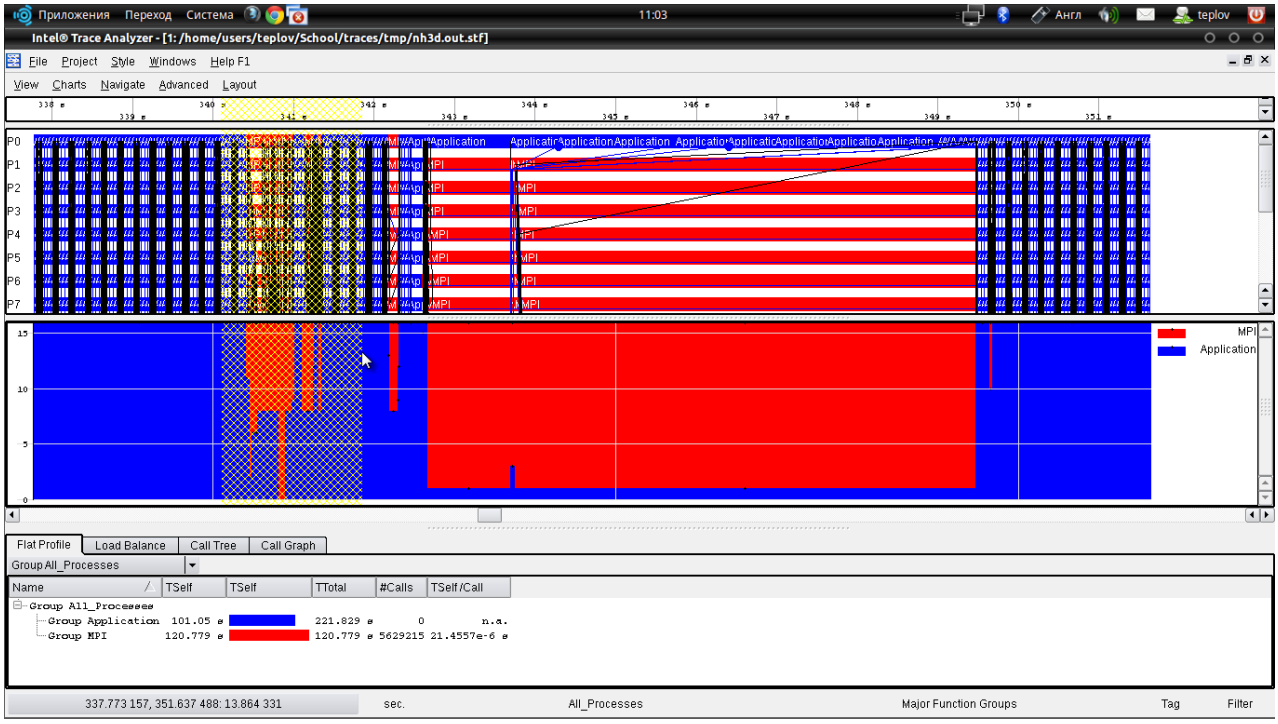

Рис 1. Пример участка трассы работы приложения, на котором работу выполняет только один процесс

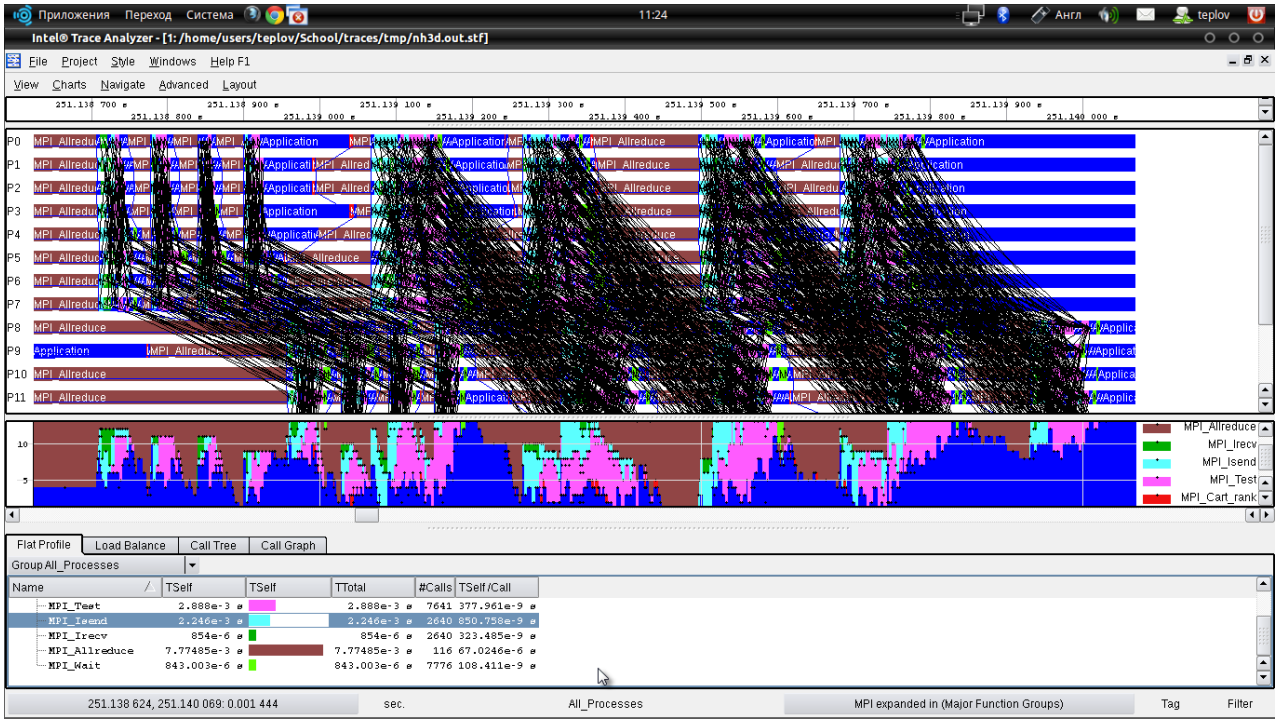

Рис 2. Пример участка трассы приложения, на котором работа процессов разбалансирована

Для выявления потенциальных узких мест удобно пользоваться количественной диаграммой (на рисунках находится под диаграммой событий). На ней отображается число процессов, участвующих в выполнении кода приложения или MPI-взаимодействий. Идеальная картина – когда все процессы заняты той или иной полезной деятельностью. Если наблюдается «лестница», то этот участок работает не оптимально. Таких мест в модели NH3D было найдено достаточно много.

На втором этапе анализа с помощью профилирования были выявлены проблемные процедуры, которые сильно замедлялись с увеличением числа процессоров. Было определено месторасположение узких мест, препятствующих улучшению производительности. Для выяснения причин отсутствия ускорения нам необходимо было провести более детальный анализ работы этих процедур. Для этого проводилось инструментирование исходного кода программы. С помощью API, предоставляемого ITAC, исходный код программы был модифицирован таким образом, чтобы на трассе приложения были выделены проблемные процедуры.

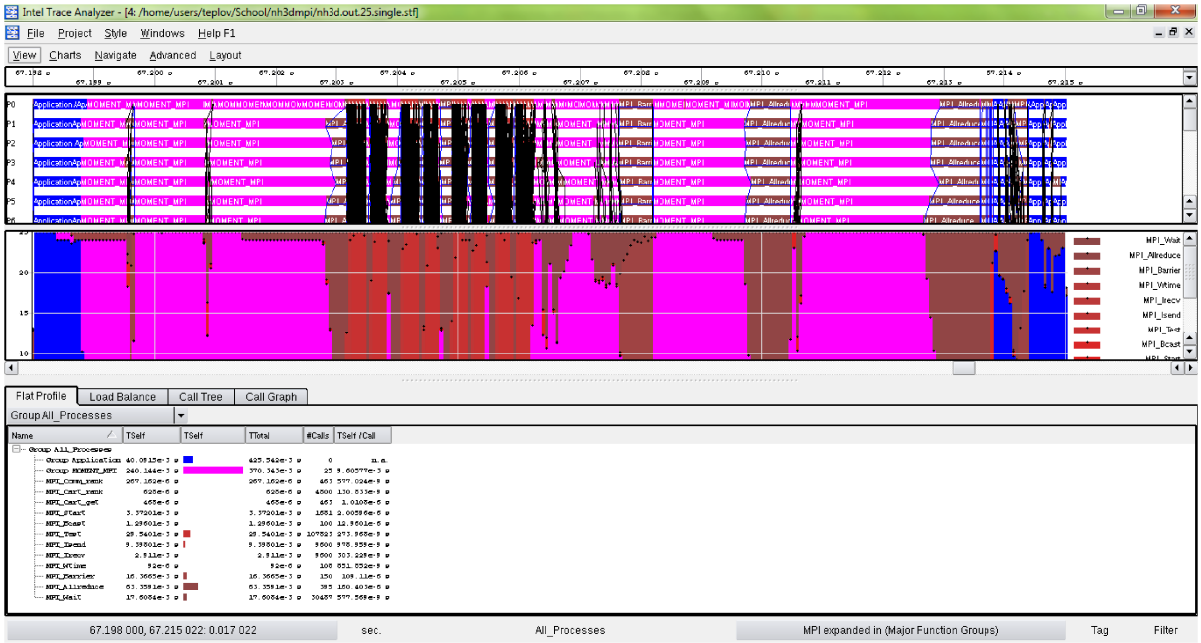

Рис 3. Участок трассы программы, на котором процедура MOMENT_MPI использует алгоритм SPONGE

В коде были выделены два дополнительных класса функций: функции инициализации и функции одного шага по времени. Это сделано с целью отображения в разных группах функций, вызываемых и при инициализации и при шаге модели по времени. Для такого разделения необходимо добавить в вызов каждой функции параметр, в котором указывался идентификатор вызывающего класса функций.

Сбор трасс проводился с использованием алгоритма усреднения данных на границе области SPONGE и без него. На рисунке 3 показана трасса работы процедуры MOMENT_MPI на 25 процессорах с использованием алгоритма SPONGE, а на рисунке 4 - без его использования.

При сравнении двух трасс легко заметить отличия в характере профиля работы процедуры и влияние процедуры SPONGE_MPI. Влияние данной процедуры усиливается с ростом числа процессоров. Таким образом, можно заключить, что SPONGE_MPI является узким местом, сильно замедляющим работу приложения.

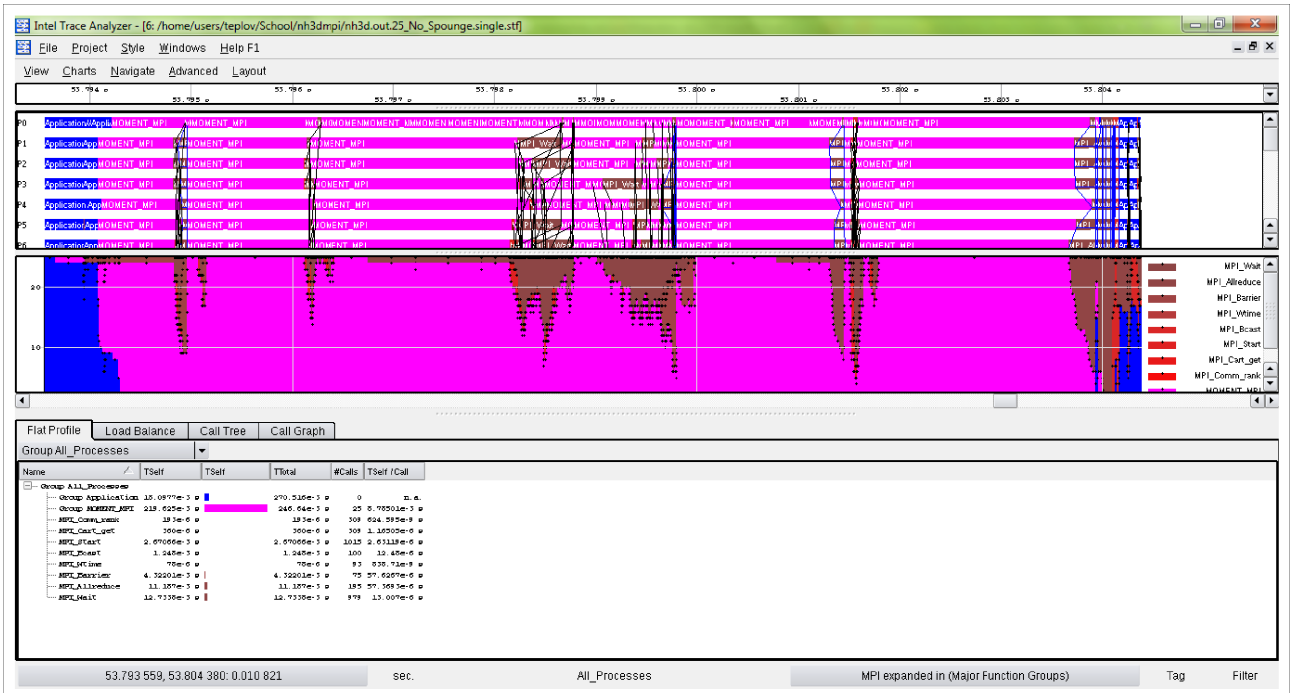

Рис 4. Участок трассы программы, на котором процедура MOMENT_MPI не использует алгоритм SPONGE

На следующем этапе работы проводилось более глубокое инструментирование шагов модели по времени. На трассу выводились данные о всех процедурах, которые используют вызовы процедур MPI. Основную сложность при инструментировании вызвали вложенные процедуры, которые используются в нескольких процедурах. Необходимо корректно учитывать отображение на трассе времени выполнения кода вложенных процедур (run time) и кода самой процедуры (self time) всех процедур. Если основу времени выполнения процедуры занимает выполнение вложенной процедуры, то больше внимания при оптимизации необходимо уделить вложенной процедуре.

В ходе такого типа анализа была выявлена процедура UPUVT_MPI, которая вызывается из многих процедур, и потому занимает существенную часть времени шага модели. Работа процедуры была проанализирована на основе собранных трасс и оптимизирована, исходя из результатов анализа.

8. Данные о масштабируемости приложения, полученные с помощью ITAC

Целью работы являлся анализ масштабируемости приложения, и потому данные, предоставляемые инструментами анализа эффективности, должны быть рассмотрены в совокупности всех проведенных экспериментов. Важной информацией для анализа масштабируемости приложения являются статистические данные о работе процедур на различных конфигурациях вычислительной системы. Представление о масштабируемости приложения возможно получить только после профилирования всех процедур и получения сводной статистики серии экспериментов.

Анализ статистических данных о метриках эффективности параллельного приложения, вне зависимости от выбранной метрики и способа сбора данных, позволяет сделать выводы о наличии проблем масштабируемости приложения. Для данной задачи использование ITAC позволило быстро собрать статистические данные экспериментов. Удобной оказалась способность Trace Analyzer оперировать с трассами большого размера и с фильтрацией ненужных данных.

После определения наличия проблемы для выявления её места в коде использовалась количественная диаграмма. По ней определялись участки кода, в которых происходит разбалансировка работы. С увеличением конфигурации вычислительной системы она сильно влияет на масштабируемость.

Более детальный поиск проблем с помощью инструментирования исходного кода позволил оценить вклад каждой процедуры в масштабируемость всего приложения и выявить наиболее критичные из них. После инструментирования кода статистика о работе выделенного участка также собиралась в сводную картину масштабируемости как всего приложения, так и его частей.

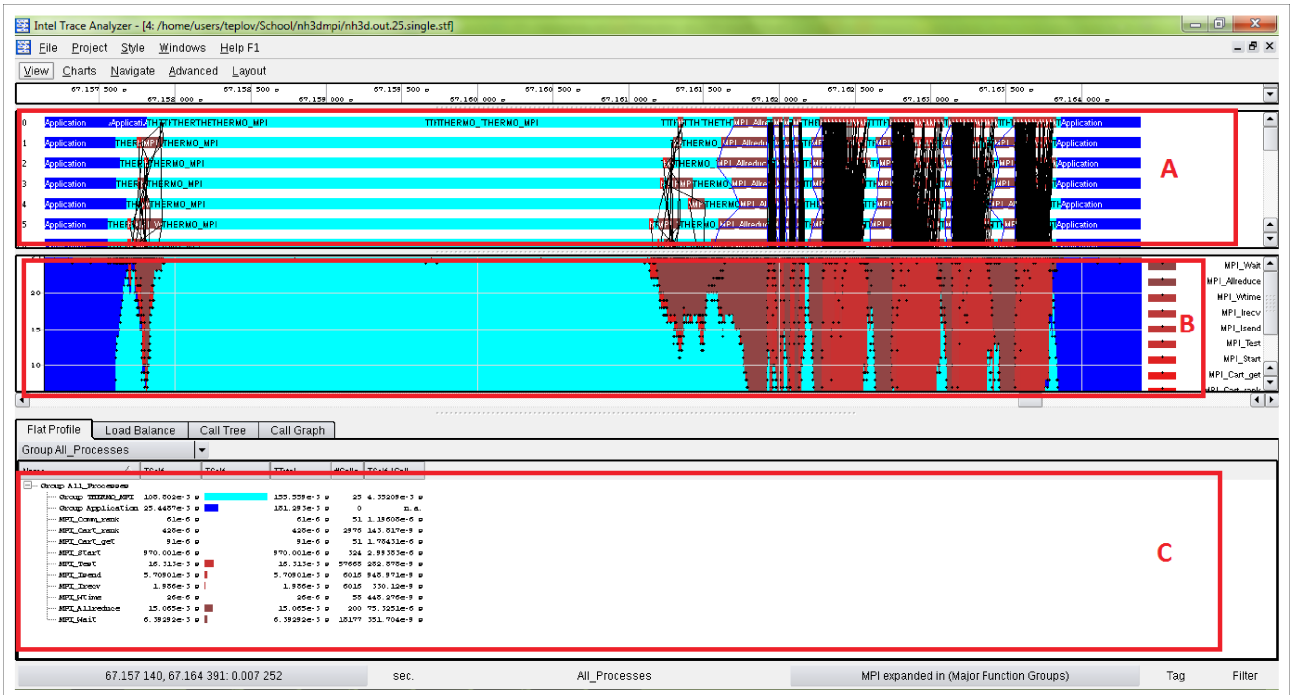

Рис 5. A – диаграмма событий. B – качественная диаграмма. C – профиль отображенного сегмента трассы

После локализации узких мест необходимо сделать выводы о причинах плохой масштабируемости в найденных участках. Узкие места более детально исследовались на диаграммах событий. Изучая в совокупности диаграммы событий (рис 5. A) и качественные диаграммы (рис 5. B), делались выводы о том, какие вызовы MPI приводят к разбалансировкам, а также каков их вклад. Для анализа данных серии экспериментов возможностей анализатора по сравнению различных трасс было недостаточно. Сравнение двух трасс не давало полноценной картины масштабируемости. Однако, сравнивая минимальные и максимальные конфигурации по числу использованных процессоров, можно получить представление о том, как сильно меняется профиль работы.

Другой проблемой стала сложность поиска узких мест при больших масштабах вычислительной системы. Общее число отображаемых событий можно сократить, применяя фильтрацию отображаемых событий. При таких масштабах визуализацию диаграммы событий невозможно уместить на одном экране, поэтому анализ затруднён. К сожалению, возможности автоматического определения проблем эффективности в данном инструменте нет.

Инструмент не позволил выявить проблемы распределения данных. Анализатор разбалансировки приложения предоставляет также данные об объёмах и количестве коммуникационных сообщений.

Делая выводы об использовании ITAC в целях исследования масштабируемости приложения, нужно отметить, что данный инструмент предоставляет достаточно данных для того, чтобы сделать однозначные выводы о существовании проблем масштабируемости. Возможностей инструмента достаточно для локализации проблем, а также получения представления о масштабируемости отдельных частей приложения. При детальном анализе узких мест с помощью ITAC можно получить представление о характере проблем исходного кода. Инструмент может анализировать трассы работы на больших конфигурациях вычислительной системы. И потому, несмотря на сложность анализа трасс с большим количеством процессоров, ITAC предоставляет достаточно информации для того, чтобы определить наличие проблем масштабируемости, локализовать узкие места программы и получить представление о характере проблемы.

Работа выполняется при поддержке гранта РФФИ 10-07-00586-а.

ЛИТЕРАТУРА:

1. Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. Introduction to Parallel Computing. (2nd Edition), Pearson, 2003.

2. В.П.Иванников., С.С. Гайсарян, В.А. Падарян. Оценка динамических характеристик параллельной программы на модели // Программирование, №4, 2006. (http://www.ispras.ru/ru/ctt/publications.php)
3. M.Alabdulkareem, S.Lakshmivarahan, S.K.Dhall. Scalability analysis of large codes using factorial designs // Journal Parallel Computing. Volume 27. Issue 9, 8/1/2001. Pages 1145 – 1171. (http://portal.acm.org/citation.cfm?id=511426)
4. Dieter Muller- Wichards, Wolfgang Ronsch. Scalability of algorithms: An analytic approach // Journal Parallel Computing. Volume 21. Issue 6, June 1995. Pages 937 – 952. (http://portal.acm.org/citation.cfm?id=202638)
5. В.М.Степаненко, Д.Н.Микушин. Численное моделирование мезомасштабной динамики атмосферы и переноса примеси над гидрологически неоднородной территорией // Вычислительные технологии, 2008. Т.13, специальный выпуск 3. С. 104-110.