УДК 004.942

Симуляционный подход для нахождения производительности параллельных MPI-приложений на вычислительном кластере.

Григорий Речистов*† Павел Шишпор*†
Александр Иванов*† Владимир Пентковский*†‡

14 ноября 2011 г.

Содержание

1 Введение 3

2 Обзор методов оценки производительности 3

3 Метрики производительности 4

4 Модель работы приложения 5

5 Инструменты 7
- 5.1 Симуляционная часть 7
  - 5.1.1 Вычисление вероятностей событий MPI 9
  - 5.1.2 Определение длительностей вызовов MPI 9
- 5.2 Экспериментальная часть 11

6 Эксперимент 12

*Московский физико-технический институт.
†ЗАО «Интел» А/О.
‡Работа выполнена в рамках гранта, выделенного в соответствии с постановлением Правительства России №220 от 09.04.2010 г.

7 Заключение **13**
7.1 Результаты 13
7.2 Уточнение модели 14
7.2.1 Система MPI вызовов 14
7.2.2 Подсистема кэшей 14
7.2.3 Перегруженность процессорных ядер 15
7.2.4 Несколько ядер в одном приложении 15

Список иллюстраций

1 Диаграмма переходов клиента в сети, представляющей один MPI-процесс. 6
2 `MPI_Send()` – `MPI_Recv()` при $L_{network} \geqslant \delta_{skew}$. 10
3 `MPI_Barrier()`. 11
4 Экран программы Intel VTune для запуска Linpack. 12
5 Сравнение результатов производительности приложения, сообщаемых VTune и Linpack. 13
6 Диаграмма переходов клиентов в сети, представляющей перегруженную систему. 15

Список таблиц

1 Конфигурация Gen2. 8
2 Конфигурация Gen1. 8

1 Введение

Задачей данной работы являлась выработка методики оценки производительности параллельных приложений, использующих стандарт MPI [1], для ещё не построенных вычислительных систем, для которых доступны точные спецификации об их конфигурации.

Успешное решение поставленной задачи позволяет заранее определить эффективность существующего кода в будущем, а также находить найти потенциальные узкие места как в проекте вычислительной системы, так и в алгоритмах приложений.

Особенностью работы является ориентация на системы с большим числом вычислительных ядер (больше ста) и соответственно большим числом исполняемых параллельно процессов MPI.

В секции 2 приведён краткий обзор подходов к решению интересующей нас задачи. Секция 3 определены измеряемые метрики. В секции 4 описывается общая модель и основные формулы для нахождения метрик. Секция 5 повествует об используемых программных инструментах. В секции 6 приводятся полученные на данный момент результаты. Секция 7 подводит итог данной работы и определяет пути дальнейшего развития исследования.

2 Обзор методов оценки производительности

Задача нахождения производительности вычислительных систем принадлежит более широкому классу задач системного анализа. Известны следующие способы исследований ЭВМ.

- Натурный эксперимент с реальной вычислительной системой [2, 3]. Современные ЭВМ имеют множество встроенных в различные подсистемы датчиков и счётчиков событий, коорые можно использовать для монитоинга их производительности по различным параметрам. Очевидный недостаток — необходимость иметь доступ к аппаратуре, при этом часто эксклюзивный (чтобы результаты экспериментов не искажались влиянием параллельно исполняемых задач других пользователей). Кроме того, такой эксперимент невозможно поставить на ещё несуществующих конфигурациях.

- Детальное компьютерное моделирование (симуляция) [4]. При этом подходе создаётся программная модель ЭВМ, учитывающая временны́е задержки во всех важных для исследования узлах, при этом степень детализации их подсистем очень высокая — может доходить до моделирования событий изменения логического сигнала. Такие модели позволяют добиться надёжных и точных предсказаний как функциональных, так и временных характеристик систем. Ценой этому является сложность и длительность разработки такой модели, необходимость изучения детальной документации по всем узлам, а

также чрезвычайно низкая скорость её работы, часто перевешивающая все преимущества.

- Функциональная компьютерная симуляция [5, 6, 7]. При подобном моделировании акцент делается на достаточно точном повторении работы узлов системы за искючением длительностей операций. При этом пропадает необходимость реализации всех алгоритмов нижележащих систем. Такие симуляторы работают достаточно точно и при этом быстро, но измеренные по ним характеристики производительности, как правило, слабо соотносятся с реально измеренными показателями.

- Представление системы в виде сети с очередями обслуживания с последующим аналитическим описанием [8]. Методика, при которой абстрагируются от выполняемых системой функций и используют данные о длительности и взаимосвязи отдельных операций, концентрируясь только на метриках производительности. Несмотря на свою простоту, данный подход при правильном использовании может правильно и точно отражать такие явления, как заторы при ограниченной ёмкости/доступности ресурсов, наличие нескольких классов задач с разными приоритетами обслуживания, параллельная обработка задач и т.п. При этом создаётся диаграмма переходов «клиентов» между центрами обслуживания. Для применимости методики необходимо обладать некоторым набором численных характеристик изучаемой системы; они могут быть получены каким-либо другим способом (экспериментом, симуляцией и т.п.).

Ни один из описанных выше подходов полностью не покрывает цели данной работы, поэтому было решено использовать комбинирование методов, позволяющее получить разумный компромисс между сложностью, точностью и гибкостью.

3 Метрики производительности

Наиболее универсально описывающая работу любого приложения величина — это время выполнения T_{run} приложения над определённым набором входных данных от момента старта до момента вывода результатов. Именно оно определяет, как долго пользователь будет ждать готовность результатов вычислений. Естественный подход для получения T_{run} — натурный эксперимент.

К сожалению, для остальных методик, описанных в секции 2, нахождение непосредственно T_{run} связано со значительными усложнениями моделей и включением в них бóльшего количества параметров, сводящих компактность описаний на нет. Проводя аналогию с механическими системами, можно сказать, что нахождение полного времени работы приложения требует учёта «краевых эффектов», явлений установления колебаний в системе, что заставляет решать систему дифференциальных уравнений, описывающих эволюцию системы, вместо гораздо более простых уравнений для установившегося режима.

По этой причине нас будут интересовать другой тип характеристик производительности систем, связанный с установившимися режимами их работы. При этом мы налагаем на приложение следующее условие: в процессе своей работы оно проводит большую часть времени, циклически выполняя сравнительно небольшой блок кода.

Такую секцию машинного кода мы будем называть *ядром приложения*. Другое именование, встречающееся в литературе — *горячий* код (*англ.* hotspot).

Введём следующие характеристики ядра приложения[1].

- Число завершившихся машинных инструкций за один такт процессора — IPC (*англ.* instructions per second). Обратная ей величина — CPI (cycles per instruction) тоже используется.
- Число совершённых операций над числами с плавающей точкой за одну секунду, FLOPS (*англ.* floating point operations per second). Как правило, подразумевается работа с числами двойной точности (длиной 64 бита) [9].
- Скорость перекачки данных в памяти (англ. memory bandwidth) [10], измеряемая в (мега-,гига-)байт/с.

Описанные выше величины объединяет общая суть — они показывают, сколько элементарных операций, рассматриваемых для конкретной программы как «полезные», осуществляется за один такт работы системы. Правильный выбор критерия «полезности» зачастую определяет адекватность изучаемых метрик задачам оптимизации или сравнительного анализа приложенний.

В данной работе, однако, мы часто будем использовать обратную величину — количество циклов процессора, приходящихся на одну полезную операцию; это будет сделано для упрощения вида некоторых формул.

4 Модель работы приложения и формулы для производительности

Наша модель является сетью центров обслуживания, времена нахождения транзакций в которых определяются измеренными на симуляционных моделях или натурных экспериментах задержками с учётом степени загруженности этих центров. Полученная схема не является простейшей т.н. разделяемой (*англ.* separable) сетью очередей обслуживания, поэтому к ней применимы не все результаты теории; однако она всё же достаточно проста для последующего анализа. Дополнительные замечания:

- Мы моделируем каждый процесс в составе одной MPI программы независимо.

[1] Далее всюду по тексту, если специально не оговорено другое, под характеристиками приложения будут пониматься средние величины, измеренные за оговоренный период наблюдения.

- В одном процессе находится один поток исполнения (thread).

- MPI процессы могут находиться как на одном компьютере, так и на различных, соединённых сетью, узлах. Разница во временах коммуникаций в этих случаях будет учтена.

- Для каждого узла сети выполняется условие, что число процессов исследуемого приложения не превышает доступное на узле количество ядер. Другими словами, вычислительные узлы не *перегружены* (*англ.* oversubscribed).

Диаграмма переходов состояния одного MPI процесса приведена на рис. 4.

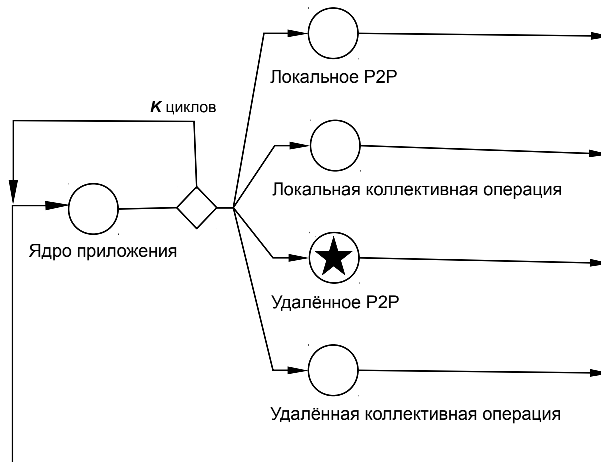

Рис. 1: Диаграмма переходов клиента в сети, представляющей один MPI-процесс. Клиент сети обозначен звездой (в данной схеме он один). Окружностями обозначены центры задержки, стрелки — пути перехода клиентов между центрами. Обратная дуга «K циклов» соответствует среднему количеству итераций ядра приложения, происходящих между двумя актами коммуникации.

В данной закрытой системе находится только один «клиент» — MPI процесс. Изначально он циклически находится в ядре приложения без коммуникаций с другими потоками; происходит K циклов каждый длительностью τ[1]; за каждый цикл происходит N_{event} интересующих нас событий (например, завершённых инструкций, операций с плавающей точкой, переданных байт). Затем поток переключается на выполнение одного из видов коммуникаций. Здесь мы выделяем две признака, образующих четыре класса:

1. При локальных операциях данные не покидают вычислительный узел и передаются, например, через общую память двух процессов. Конкретная реализация зависит от используемой библиотеки MPI.

[1]Все временные интервалы, если это не оговорено специально, измеряются в единицах тактов процессора.

2. При удалённых операциях данные проходят по сети через одно или несколько маршрутизующих устройств до процесса-получателя.

3. При одноадресной коммуникации (P2P от *англ.* peer to peer) получатель сообщения один.

4. В коллективных операциях участвуют несколько процессов.

Подобное разделение на классы обусловлено тем, что на величины задержек выделенных классов влияние оказывают как различные характерные времена исполнения, так и детали реализации библиотеки MPI. Например, коллективные операции могут потреблять время от $O(1)$ до $O(N^2)$ в зависимости от того, локальны они или удалённы, и того, насколько эффективный алгоритм для них использован [11].

С каждым классом событий сопоставлена его вероятность $P\{MPI_{i,n}\}$ и среднее время выполнения $Latency_{i,n}$.

Полное среднее время цикла обслуживания процесса:

$$T_{full} = K\tau + \sum_{i,n} P\{MPI_{i,n}\} \cdot Latency_{i,n} \quad (1)$$

Время, затрачиваемое на одно событие (CPE, cycles per event), равно

$$CPE = \frac{T_{full}}{K \cdot N_{event}} = CPE_{ideal} + CPE_{synch} \quad (2)$$

Здесь $CPE_{ideal} = \frac{\tau}{N_{event}}$ — скорость выполнения программы в ядре. Это «идеальная» скорость, которая могла бы быть достигнута, если бы коммуникации MPI происходили бы мгновенно; CPE_{synch} — приведённое к одному событию время, затраченное на коммуникации.

5 Инструменты

Для нахождения численных значений величин, входящих в формулу (1), нами используются результаты как натурных экспериментов, так и симуляции.

5.1 Симуляционная часть

Для моделирования работы всех узлов многопроцессорной системы в нашем исследовании используется программа Simics [12], позволяющая построить гибкую функциональную распределённую модель кластера. Параметры изучаемой системы (далее называемой Gen2) приведены в табл. 1; конфигурация физической системы ЭВМ, на которой модель запускалась (далее обозначаемой как Gen1), приведена в табл. 2.

Отношение числа ядер Gen2 к числу ядер Gen1 равно 9,33, однако возможностей Simics достаточно для того, чтобы загрузить Linux и запустить изучаемое приложение.

Параметр	Значение
Число выч. узлов	112
Число процессоров в одном выч. узле	2
Число ядер в одном процессоре	16
Тип процессора	Intel Xeon (Sandy Bridge)[a]
Частота процессоров	2,8 ГГц
ОЗУ одного узла	48 Гбайт
Сеть	Infiniband QDR 10 Гбит/с
Полное число ядер в системе	3584

[a]Процессор ещё не выпущен, модель создана по его публично доступной документации.

Таблица 1: Конфигурация Gen2.

Параметр	Значение
Число выч. узлов	16
Число процессоров в одном выч. узле	2
Число ядер в одном процессоре	12[a]
Тип процессора	Intel Xeon X5680 (Westmere)
Частота процессоров	3,33 ГГц
ОЗУ одного узла	24 Гбайт
Сеть	Infiniband QDR 10 Гбит/с
Полное число ядер в системе	384

[a]Опция Intel HT включена, поэтому обозначены все логические ядра.

Таблица 2: Конфигурация Gen1.

Simics используется для изучения поведения сетевой подсистемы при исполнении MPI-приложения. При этом записывается трасса вызовов функций MPI для последующего анализа и получения значений величин K, $P\{MPI_{i,n}\}$, $Latency_{i,n}$. Сохранение трассы позволяет при последующих её анализах задавать различные параметры сети (задержки, пропускную способность, топологию) и различные алгоритмы коллективных операций MPI. При этом отпадает необходимость запускать симуляцию в Simics, таким образом, экономится время. Однако при подозрениях на то, что собранная трасса не отражает адекватно поведение некоторой конфигурации сети, всегда остаётся возможность прогнать симуляцию для новых значений параметров.

5.1.1 Вычисление вероятностей событий MPI

Вероятность каждого из четырёх классов событий вычисляется по простой формуле из классической теории вероятности:

$$P\{MPI_{i,n}\} = \frac{N_{i,n}}{\sum_{j,k} N_{j,k}}, i \in (p2p, collective); n \in (local, remote) \quad (3)$$

Здесь $N_{i,n}$ — количество событий класса (i, n) в собранной трассе. Для обнаружения событий (т.е. вызовов функций библиотеки MPI) используется модификация библиотеки MPI: в каждый вызов добавляется одна т.н. «волшебная» машинная инструкция, не изменяющая ход работы приложения (Для архитектуры Intel IA-32 используется `CPUID` [13]), но детектируемая Simics; при обнаружении такой инструкции происходит запуск скрипта, анализирующего, какой вызов MPI был произведён, и записывающего в трассу данные о нём.

Таким образом достигается разумный баланс между:

- количеством работы по модификации изучаемого приложения — модифицируется только код библиотеки, которая может быть затем переиспользована для сборки различных приложений;
- искажениями, вносимыми в работу приложения — одна дополнительная инструкция на сотни, составляющие один вызов MPI;
- количеством информации, доступной для анализа — при обработке события исполнения «волшебной» инструкции нам доступно для изучения полное состояние не только процесса, в котором она исполнена, но и всех остальных вычислительных подсистем.

5.1.2 Определение длительностей вызовов MPI

Значения задержек отдельных MPI-функций при коммуникациях удалённых процессов зависит от параметров и топологии сети. Для локальных взаимодействий она зависит от реализации механизмов общей памяти. В обоих случаях также важен алгоритм реализации каждого вызова.

Для локальных взаимодействий значения задержек проще всего получить с помощью VTune. При этом предполагается, что полученные числа будут нечувствительны к изменениям структуры вычислительного кластера.

Длительности передач данных по сети удобнее получать из результатов моделирования с помощью Simics. При этом задержка, вызванная собственно передачей данных, зависит от топологии сети и наличия в ней возможности образования заторов (*англ.* contention). Ниже приведены варианты формул для нескольких сетей.

Сеть типа «mesh»

$$L_{network}(packet) = l_{media} + Size(packet)/Bandwidth, \quad (4)$$

где l_{media} — латентность соединений в секундах, $Size(packet)$ — размер пакета в байтах, $Bandwidth$ — ширина канала в байт/с.

Сеть с заданной топологией без заторов

$$L_{network}(packet) = \\ l_{media} + N_{hops} \cdot Size(packet)/Bandwidth + (N_{hops} - 1) \cdot l_{switch}, \quad (5)$$

где N_{hops} — число сегментов сети, через которые прошёл пакет, l_{switch} — задержка обработки на одном маршрутизаторе (которых было $N_{hops} - 1$ на пути следования пакета).

Сеть типа «звезда» с заторами на маршрутизаторе

$$L_{network}(packet) = \\ l_{media} + Size(packet)/Bandwidth + Length(queue) \cdot l_{switch}, \quad (6)$$

где $Length(queue)$ — среднее число пакетов, ожидающих обработки на маршрутизаторе, на момент прихода нового пакета, получаемое из его Simics-модели.

Следует отметить, что во всех представленных моделях игнорируется факт фрагментации передаваемых сообщений в пакеты сети. Это может привести к заниженным величинам при большом количестве мелких сообщений.

Теперь рассмотрим два примера влияния реализаций MPI-функций на время их выполнения.

MPI Send На рис. 2 приведена схема взаимодействия двух процессов. Первый вызывает неблокирующий `MPI_Send()` продолжает работу, тогда как второй ждёт окончания передачи данных в `MPI_Recv()`.

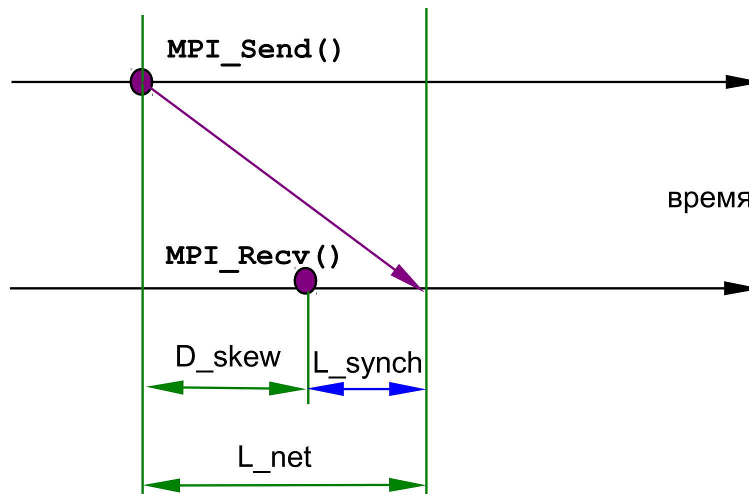

Рис. 2: `MPI_Send()` – `MPI_Recv()` при $L_{network} \geqslant \delta_{skew}$.

Моменты входа в `MPI_Send()` и `MPI_Recv()` сдвинуты на δ_{skew}[1], что частично компенсирует время передачи по сети. Однако, время ожидания не может быть меньше

[1]Отметим, что эта величина может быть отрицательной, что означает, что приёмник стал ожидать данные раньше, чем они начали передаваться.

нуля. Таким образом, формулы для величин задержек такие:

$$L_{Send} = 0 \tag{7}$$

$$L_{Recv} = \max(L_{network} - \delta_{skew}; 0) \tag{8}$$

MPI Barrier На рис. 3 изображена блокировка нескольких процессов до тех пор, пока все они не достигнут барьера.

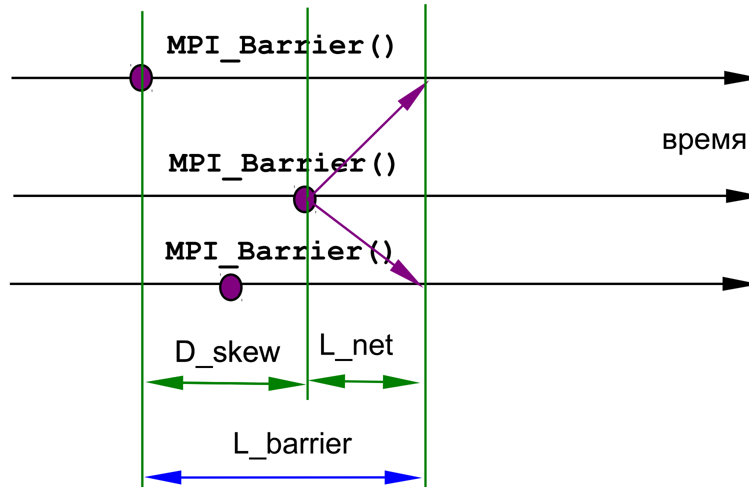

Рис. 3: `MPI_Barrier()`.

В этом случае величина «перекоса» δ_{skew} определяется наибольшим расстоянием между моментами вызовов всех `MPI_Barrier()`. После этот факт должен быть доведён до всех процессов, что вызывает дополнительную задержку. Полное время:

$$\delta_{skew} = \max_{j,k;j \neq k} (t_j - t_k), j,k \in (1; MPI_Size) \tag{9}$$

$$L_{Barrier} = \delta_{skew} + L_{network} \tag{10}$$

Из приведённых примеров видно, что для вычисления данных величин необходимо иметь трассы всех MPI-событий в системе, а не только произошедших в контексте изучаемого процесса.

5.2 Экспериментальная часть

Точности используемого симулятора недостаточно для получения информации о микроархитектурном поведении ядра приложения на вычислительном ядре ЦПУ, т.к. Simics предоставляет лишь функциональную модель процессора. Кроме того, точное потактовое моделирование всех микроархитектурных особенностей нерационально с т.з. времени эксперимента: известно, что такие симуляторы имеют скорости приложений в сотни тысяч раз меньшие, чем непосредственное выполнение на реальной аппаратуре.

По этой причине мы используем измерения на настоящей аппаратуре. Используются встроенные в современные процессоры счётчики микроархитектурных событий, позволяющие собрать информацию о практически всех аспектах работы приложения. При этом замедление изучаемого приложения существенно меньшее, чем было бы при использовании потактовой модели.

Для анализа аппаратных счётчиков использовалась программа Intel VTune [2], дополнительно имеющая возможность находить секции «горячего» кода приложения и демонстрировать результаты дизассемблирования этих участков.

Получаемое с помощью Intel VTune значение τ получается из значений счётчика `CPU_CLK_UNHALTED.REF_TSC`, а N_{event} — из соответствующих счётчиков событий (например, величину FLOPS можно выводить из `FP_COMP_OPS_EXE.SSE_PACKED_-DOUBLE`, `FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE` [14], CPI — из `INST_RETIRED.ANY`). Подробнее об этом в секции 6.

6 Эксперимент

На рис. 4 приведён пример результатов анализа «горячего» кода с помощью Intel VTune для приложения Linpack [15]. Измерения проводились на одном из узлов кластера Gen1.

Sandy Bridge - Cycles, FP events - Hardware Event Sample Counts

Analysis Target | Analysis Type | Summary | Bottom-up | Module Timeline

Grouping: Module / Basic Block

Module / Basic Block	CPU_CLK_UNHALTED. REF_TSC by Package	INST_RETIRED. ANY by Package	FP_COMP_OPS_EXE. SSE_PACKED_DOUBLE by Package	FP_COMP_OPS_EXE. SSE_PACKED_SINGLE by Package
xhpl	1,833	4,373	977	0
0x4946e0	511	1,388	748	0
0x4a8366	204	444	0	0
0x42ec92	179	343	0	0
0x422d41	166	447	0	0
0x42ed6d	125	306	0	0
0x49471a	119	411	160	0
0x4ac428	59	133	0	0

Рис. 4: Экран программы Intel VTune для запуска Linpack.

Полученные данные обнаруживают самый «горячий» участок приложения (начинающийся по адресу 0x4946E0), при этом в нём содержится и наибольшее количество инструкций над значениями с плавающей точкой (векторных команд SSE). Анализ машинного кода этого блока показал, что он соответствует вычислительной части алгоритма решения задачи (все инструкции — это операции сложения или умножения), поэтому в нашей модели он соответствовует ядру приложения.

Второй по времени исполнения блок (начинающийся с адреса 0x4A8366) тоже содержит векторные инструкции, однако анализ его кода показал, что все они выполняют операции чтения и записи в память, а не арифметические действия, т.е. соответствуют коммуникациям.

На рис. 5 приведены результаты серии измерений на Linpack при различных размерах матрицы N. Приведены значения, сообщаемые самим приложением, и измерениями для ядра.

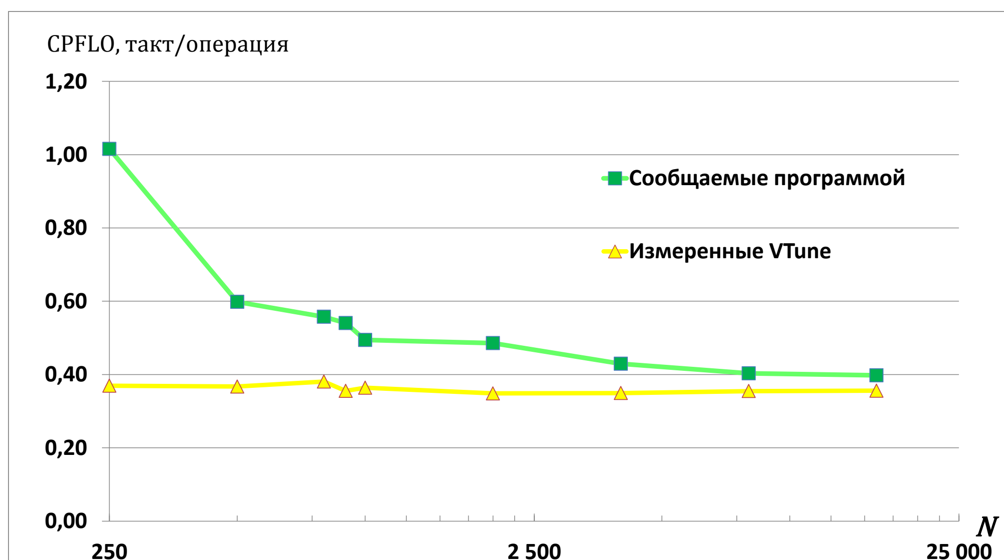

Рис. 5: Сравнение результатов производительности приложения, сообщаемых VTune и Linpack. По оси абсцисс отложены размеры задач приложения, по оси ординат — число циклов, затрачиваемых на одну операцию над числом с плавающей точкой двойной точности. Нижний график — измеренная VTune величина для ядра приложения, верхний — сообщаемая самим тестом по завершении скорость вычислений, приведённая к одному такту процессора.

Комментарии к графикам:

- Ядро всегда работает быстрее, чем в всё приложение. Разница обусловлена необходимостью синхронизации отдельных потоков, т.е. коммуникации.
- Скорость работы ядра практически не зависит от размера задачи. Это подкрепляет гипотезу, что значение CPE_{ideal} нечувствительно к вариациям сетевой подсистемы кластера.
- При росте N потери на коммуникациях уменьшаются, и скорость всего приложения приближается к своему предельному значению, определяемому работой ядра. Такое поведение характерно для Linpack [15].

7 Заключение

7.1 Результаты

К настоящему моменту в рамках работы были получены следующие результаты:

1. На аппаратуре Gen1 построена, запущена и отлажена модель Gen2.
2. Для Linpack произведено изучение применимости предложенной методики анализа производительности.
3. Разработана методика сбора трасс событий MPI с помощью симулятора.

Дальнейшая работа включает в себя применение предложенной методики вычисления производительности MPI-приложений. Основной интерес для нас представляют программы молекулярной динамики, симулирующие задачи биологии [16]. Кроме того, представляет интерес описанное в следующей секции развитие предложенного подхода.

7.2 Уточнение модели

Продемонстрированная в секции 4 модель имеет в своём основании упрощённые предположения о характере взаимодействия MPI-процессов, различий в технологиях поколений ЭВМ и ограничения на структуру приложения.

Ниже представлены предлагаемые расширения модели, призванные повысить степень её соответствия реальной системе. Детальное их рассмотрение выходит за рамки данной статьи, здесь мы лишь обозначим основные идеи.

7.2.1 Система MPI вызовов

Очевидное улучшение здесь — увеличение количества выделенных классов коммуникаций для более точного улавливания характера поведения. Наиболее важным здесь является отделение неблокирующих вызовов от блокирующих. Тогда как длительность первых зависит лишь от реализации вызова и нижележащих механизмов ОС, вторые очевидно подвержены влиянию как величины «перекоса» моментов старта операции в разных процессах. Кроме того, для части вызовов MPI стандарт не определяет, должны ли они быть блокирующими или нет.

7.2.2 Подсистема кэшей

Конфигурации кластеров Gen1 и Gen2 (и последующие за ними Gen3, Gen4 ...) имеют ядра процессоров, совместимые между собою по набору команд, однако иерархия кэшей в них может существенно различаться, что отразится на характеристиках производительности ядра изучаемого приложения. Intel VTune невозможно запустить на ещё не выпущенных процессорах, поэтому данный фактор необходимо будет учитывать с помощью техник симуляции моделей кэшей.

Simics поддерживает подключение моделей кэшей различных конфигураций с задаваемой топологией соединений (включая протокол когерентности MESI), конфигураций ассоциативности, алгоритмов вытенения линий и задержек при промахах.

Для такой модели в формуле (2) появится дополнительный член:

$$CPE = CPE^{\star}_{ideal} + CPE_{caches} + CPE_{synch} \tag{11}$$

Здесь CPE_{caches} — полученная в Simics средняя задержка на системе кэшей, $CPE^{\star}_{ideal}$ — скорость работы ядра, пересчитанная в предположении о «мгновенности» доступов в память.

7.2.3 Перегруженность процессорных ядер

Как было отмечено в предположениях секции 4, число потоков в программе не превышает количество доступных ядер. При невыполнении этого условия будут возникать ситуации, когда один или несколько потоков ожидают освобождения ядра, чтобы исполниться. Это обстоятельство должно находить отражение при построении сети центров обслуживания и показано на рис. 6.

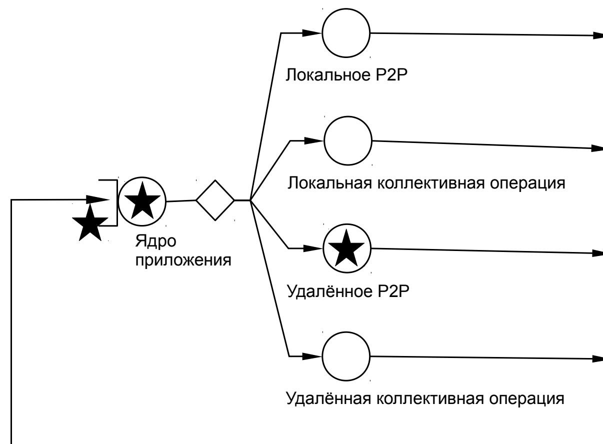

Рис. 6: Диаграмма переходов клиентов в сети, представляющей перегруженную систему. Скобкой обозначена очередью ожидающих процессов. В ситуации, изображённой на рисунке, один процесс исполняется, второй ожидает завершения коммуникаций, третий уже вышел из блока коммуникаций и ожидает освобождения ядра.

7.2.4 Несколько ядер в одном приложении

Для некоторого приложения может оказаться, что нельзя выделить в нём ровно одну область кода, в которой происходят все интересующие нас события. В таком случае будет необходим более детальный анализ такого алгоритма с целью понять, как связаны два или более ядер — выполняются ли они последовательно или же существуют одновременно, при этом конкурируя за доступные вычислительные ресурсы.

Список литературы

[1] *MPI: A Message-Passing Interface Standard. Version 2.2.* Sept. 2009. URL: http://www.mpi-forum.org/docs/docs.html.

[2] *Intel® VTune™ Performance Analyzer 9.1 for Linux* – Documentation.* Intel Corporation. URL: http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-for-linux-documentation/.

[3] Eduardo Rocha Rodrigues Jairo Panetta and Philippe O. A. Navaux. *On Simulation of Massively Parallel Computers.* 2008. URL: http://gppd.inf.ufrgs.br/wsppd/2008/papers/Rodrigues.pdf.

[4] Matt T. Yourst. *PTLsim User's Guide and Reference. The Anatomy of an x86-64 Out of Order Superscalar Microprocessor.* 2007.

[5] Richard Uhlig et al. "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture". In: *Intel Technology Journal* (1999), pp. 112–126.

[6] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *FREENIX Track: 2005 USENIX Annual Technical Conference.* 2005. URL: http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[7] Darek Mihocka and Stanislav Shwartsman. "Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure". In: *ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation* (). URL: http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf.

[8] Edward D. Lazowska et al. *Quantitative system performance: computer system analysis using queueing network models.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984. ISBN: 0-13-746975-6.

[9] Jack J. Dongarra. "Performance of Various Computers Using Standard Linear Equations Software". In: (). URL: ftp://netlib2.cs.utk.edu/benchmark/performance.pdf.

[10] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25. URL: http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps.

[11] Ping Lai, Sayantan Sur, and Dhabaleswar K. Panda. "Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems". In: *Computer Science - R&D* 25.1-2 (2010), pp. 3–14.

[12] Peter S. Magnusson et al. "Simics: A Full System Simulation Platform". In: *Computer* 35 (2 Feb. 2002), pp. 50–58. ISSN: 0018-9162. DOI: 10.1109/2.982916. URL: http://portal.acm.org/citation.cfm?id=619072.621909.

[13] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 2A*. Intel Corporation.

[14] David Levinthal. *Performance Analysis Guide for Intel® Core™i7 Processor and Intel® Xeon™5500 Processors*. 2009. URL: http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[15] *High performance Linpack benchmark*. Netlib. URL: http://www.netlib.org/linpack/.

[16] David Van Der Spoel et al. "GROMACS: Fast, flexible, and free". In: *Journal of Computational Chemistry* 26.16 (2005), pp. 1701–1718. ISSN: 1096-987X. DOI: 10.1002/jcc.20291. URL: http://dx.doi.org/10.1002/jcc.20291.