УДК 004.383.4

Н.С. Поливанов, студ. 5 к.

Г.С. Речистов, м.н.с.

А.А. Абдухаликов, студ. 3. к.

В.М. Пентковский, д.т.н., рук. лаборатории

Лаборатория суперкомпьютерных технологий для биомедицины, фармакологии и малоразмерных структур, факультет радиотехники и кибернетики, Московский физико-технический институт, Москва, Россия

ЗАО «Интел А/О», Москва, Россия

РЕАЛИЗАЦИЯ ИНСТРУМЕНТАРИЯ ДЛЯ ИССЛЕДОВАНИЯ СЕТЕВОЙ ПРОИЗВОДИТЕЛЬНОСТИ MPI-ПРИЛОЖЕНИЙ НА РАСПРЕДЕЛЕННОМ СИМУЛЯТОРЕ

В работе демонстрируется использование распределенного симулятора Simics для сборки трасс вызовов процедур MPI, и последующего их анализа. Приведено описание процесса постановки эксперимента и первые результаты анализа полученных трасс для реализации бенчмарка Linpack HPL.

Ключевые слова: распределённая симуляция, многоядерные системы, трассировка, производительность сети, MPI, Simics, кластер, Linpack.

Введение

При разработке новых распределённых вычислительных машин возникает необходимость оценки производительности тех или иных приложений в зависимости от конфигурации этих машин (конфигурация отдельных узлов, конфигурация сети). Распределенные приложения, как правило, используют библиотеки MPI для обмена данными между узлами.

В качестве метрики производительности приложения используется величина CPI (*англ.* cycles per instruction), она равна среднему количеству циклов, необходимых для исполнения одной инструкции процессора. Эта величина складывается из нескольких факторов [1][2]:

$$CPI = CPI_{core} + CPI_{caches} + CPI_{memory} + CPI_{MPI}$$

Среди этих факторов: количество циклов, затрачиваемое на исполнение непосредственно в ядре процессора (CPI_{core}), задержки при обращении в кэш-память (CPI_{caches}) и оперативную память (CPI_{memory}), коммуникации с помощью MPI (CPI_{MPI}).

Для анализа величины CPI_{MPI} был разработан сборщик трасс для функционального симулятора Simics. Он собирает вызовы MPI-функций с метками времени, что позволяет в дальнейшем проанализировать производительность приложения в целом в зависимости от производительности и топологии сети.

Разработанный нами инструмент анализа имеет следующие возможности:

1. Собранные трассы позволят проанализировать эффективность использования протокола MPI приложениями на кластере.
2. Используя трассы, можно проводить дальнейший анализ производительности приложения для модифицированных конфигураций сети без дополнительных запусков изучаемого приложения, таким образом, экономя время.
3. Возможен запуск приложения на конфигурациях оборудования с числом узлов, превышающих количество, имеющееся в наличии, в несколько раз, позволяет изучать ещё не созданные системы.
4. Симуляция оказывает минимальное возмущение на исполнение изучаемой программы, т.к. сбор и запись событий вносит минимальную и предсказуемую задержку в полное виртуальное время работы приложения – несколько инструкций на каждый вызов.

Обзор литературы

Для исследования MPI приложений в литературе описано несколько сборщиков трасс, а также симуляторов. В [3] описывается подход к симуляции сети с возможностью настройки задержек при передаче сообщений между узлами. При помощи использования профилирующего интерфейса MPI и дополнительного узла-симулятора моделируется виртуальное время процесса (получаемое процессом через **MPI_Wtime**). Управление виртуальным временем позволяет минимизировать его возмущения из-за влияния симуляции. Скорость работы модели сравнима с наблюдаемой на реальном оборудовании. Однако при этом количество симулируемых узлов ограничено

количеством реально доступных узлов. В нашем решении размер моделируемой системы превышал физическую до 16 раз.

В работе [4] реализована специальная библиотека MPI, симулирующая процедуры передачи данных. Изучаемое MPI-приложение преобразуется в многопоточное (с общей памятью потоков) в целях экономии ресурсов системы при коммуникациях. Это требует дополнительных усилий при его компиляции (обработка глобальных переменных как локальных, неявное порождение потоков и т.п.). Адаптация программ, написанных на языках программирования, отличных от Си, потребует дополнительной модификации компилятора. Возможны проблемы с корректностью работы приложения, если используемые сторонние библиотеки не поддерживают многопоточность. Наш подход не требует изменения сопутствующих инструментов или исходного кода приложения, за исключением стадии компоновки.

Третий способ описан в [5]. В работе производится моделирование как сети, так и подсистемы ввода-вывода (MPI IO). Есть возможности работать с распределенной файловой системой. Кроме того, предлагается способ оценки энергопотребления по полученным заранее трассам на существующем оборудовании. Моделируется только одно пользовательское приложение, отсутствие полной симуляции узлов так же ограничивает симуляцию различных конфигураций.

Использование симулятора аппаратных платформ позволяет учесть влияние операционной системы, фоновых процессов и особенностей аппаратуры моделируемой системы, тогда как модели, работающие только с одним приложением, пренебрегают этими эффектами, тем самым снижая достоверность результатов.

Изучаемое приложение и инструменты исследования

В качестве первого исследуемого приложения первоначально был взят тест (бенчмарк) High Performance Linpack [6] (сокращённо **HPL**). Он компилировался с использованием библиотекой MPICH2 [7] версии 1.4, реализующей стандарт MPI2. Данное приложение является стандартным тестом производительности вычислительных машин на операциях с плавающей запятой и представляет собой реализацию алгоритма решения системы линейных уравнений с плотной матрицей. Операции линейной алгебры осуществляются с помощью библиотеки BLAS (реализация ATLAS 3.97).

Следующее изучаемое приложение – **mdrun** из состава пакета молекулярной динамики Gromacs [8]. В данной работе оно было собрано компилятором GCC 4.4.5, двойная точность вычислений.

Все приложения были собраны и запускались под управлением 64-битной операционной системой GNU/Linux Debian 6 “Wheezy”.

Для построения модели был взят Wind River Simics – функциональный симулятор аппаратных платформ ЭВМ. Он поддерживает симуляцию широкого спектра систем, в т.ч. многомашинных конфигураций.

Особенностями Simics являются:

- Детерминированность симуляции, даже в случае распределённых систем.
- Возможность сохранения контрольных точек для восстановления состояния системы без полного перезапуска симуляции.

- Стабильный и хорошо документированный программный интерфейс для написания новых моделей, а также обширная библиотека готовых компонентов.
- Наличие интеграции со скриптовым языком Python для автоматизации процессов исследования.

Симуляция распределённых систем в Simics описано в документации [9][10].

Кроме того, Simics позволяет создавать и привязывать собственные подпрограммы-обработчики, называемые *"Hap"*, для определённых событий в моделях, например, исключения процессора, смена его режима работы, вывод строки на экран и т.п. Среди таких событий есть исполнение «магической» инструкции (*англ.* magic instruction), которая встраивается в исследуемое приложение для отслеживания процесса исполнения.

Данный подход применим для других симуляторов; реализации его присутствуют в Qemu [11] и SoftSDV [12]. Для этого необходима возможность прерывать исполнение по «магической инструкции» и получать данные из памяти виртуальной системы. Описываемая ниже профилирующая библиотека может быть перенесена на другие симуляторы и архитектуры с минимальными изменениями, для этого достаточно выбрать подходящую «магическую» инструкцию и обеспечить перехват её исполнения. Реализация внешнего модуля трассировщика более привязана к предоставляемому программному интерфейсу конкретного симулятора, но решение является достаточно общим и тоже может быть перенесено на другие системы.

Трассировщик

Сборщик трасс состоит из профилирующей библиотеки, подключаемой к исследуемой программе, и модуля для симулятора. Профилирующая библиотека переопределяет процедуры MPI, тем самым перехватывая обращения к ним. Модуль обрабатывает поступающие события, вызванные «магической» инструкцией, собирает и записывает трассы.

Процесс использования реализованного трассировщика выглядит следующим образом:

1. По заголовочным файлам библиотеки MPI, программа на Python генерирует файл с описанием всех найденных прототипов MPI-функций и исходный код профилирующей библиотеки на языке Си, который затем преобразуется в объектный файл с помощью компилятора GCC.
2. Исследуемое приложение компонуется с полученной на предыдущем шаге профилирующей библиотекой.
3. Нужно загрузить модуль **mpi-tracker** в симулятор Simics и затем зарегистрировать с помощью команд **`g-register-mpi-tracker`** или **`g-register-mpi-tracker-delayed`**. Первая команда немедленно регистрирует и включает трассировщик во всех процессах Simics. Вторая делает это с заданной задержкой во времени, а также отключает трассировщик через заданное время. Это позволяет автоматизировать сбор данных только для

интересующих нас участков процесса работы приложения. Обе команды принимают файл с описанием MPI процедур, полученный на 2 этапе.

4. После окончания работы следует отключить трассировщик с помощью **`g-unregister-mpi-tracker`**. Это нужно для корректного и полного сброса данных всех собранных на диск.
5. Для каждого запущенного у участвующего в симуляции процесса Simics создается бинарный файл с трассами в директории запуска модели. Для их последующего чтения написана небольшая библиотека на Python (используемая и в трассировщике). Нами написаны различные обработчики, извлекающие информацию из этих файлов: простое преобразование трасс в текстовый вид; анализ частот возникновения MPI-событий, простой симулятор для проигрывания истории взаимодействия процессов при работе MPI-программы.

Рабочий процесс сбора трасс выглядит следующим образом (

Список **подрисуночных подписей**

1. Общая схема работы MPI-трекера при симуляции.
2. Распределение частот вызовов MPI процедур при работе HPL.
3. Процесс распределения копий Simics при старте симуляции на узлах, выделенных с помощью SLURM.
4. Зависимость среднего гармонического для длины интервала вычислений между двумя соседними вызовами MPI-функций от полного числа MPI-ранков в приложении **mdrun**. Каждый узел содержит 16 ранков, полное число узлов варьируется от 10 до 80.

):

1. Приложение вызывает процедуру MPI, которая переопределена в профилирующей библиотеке.
2. Профилирующая библиотека вызывает настоящую процедуру MPI с теми же аргументами, получает возвращаемое значение.
3. Профилирующая библиотека исполняет «магическую» инструкцию, при ее выполнении Simics создает событие (HAP).
4. Исполнение программы приостанавливается в месте выполнения «магической инструкции», и вызывается обработчик события, находящийся в модуле трассировщика.
5. Обработчик в модуле собирает и записывает в файл:
    a. Аргументы MPI-процедуры.
    b. Полученное возвращаемое значение из процедуры MPI.
    c. Номер (*англ.* rank) MPI-процесса в коммуникаторе MPI_COMM_WORLD.
    d. Текущее виртуальное время (количество циклов текущего процессора).
    e. Текущий номер узла и ядра процессора.
6. Исполнение передается обратно в профилирующую библиотеку, а затем приложение.

Поскольку сбор и запись трассы происходит в симуляторе, а не в гостевой системе, то это время работы не учитывается в виртуальном времени системы, а значит и в полученных трассах. Вся активность проходит незаметно для исполняемой программы, для которой она представлена выполнением одной обычной инструкции.

Профилирующая библиотека

Данная библиотека переопределяет процедуры MPI, что позволяет перехватывать управление от приложения. Программа компонуется с профилирующей библиотекой, для этого компоновщик должен поддерживать слабые ссылки. Объявления процедур MPI в библиотеке должны быть слабыми (т.к. в таком случае их можно переопределить на этапе компоновки). Чтобы вызывать процедуры MPI из профилирующей библиотеки, в стандарте MPI предусмотрен профилирующий интерфейс [13].

«Магическая» инструкция – это команда, которая не влияет на исполнение программы в симуляторе, но позволяет вызвать заранее определенный обработчик в нем. В случае Simics используется инструкция **CPUID** для архитектуры x86-64, при исполнении которой генерируется событие **Core_Magic_Instruction**. Эта инструкция позволяет передавать ограниченный объём информации в симулятор. В оригинальном макроопределении, поставляемом с Simics для x86-64, это 2 байта, в нашем исследовании оно было расширено для передачи информации объемом до 18 байт. В данном случае передается порядковый номер процедуры, MPI rank и указатель на массив с аргументами функции, что позволяет установить возвращаемое значение, количество и тип её аргументов.

В каждой функции профилирующей библиотеки был заведен массив на стеке, в который помещаются возвращаемое значение и аргументы функции, которые потом читаются из модуля трассировщика. Для этого был модифицирован макрос магической инструкции,

чтобы передавать MPI rank и указателя на массив через регистры, задействованные инструкцией **CPUID**.

Для сохранения текущего rank процесса после инициализации изучаемой программы в функции **MPI_Init** дополнительно вызывается **PMPI_Comm_rank** для коммуникатора **MPI_COMM_WORLD**, и полученный номер сохраняется в глобальной переменной (номера процессов в других порождённых коммуникаторах не отслеживаются).

Профилирующая библиотека создается автоматически на основе заголовочных файлов MPI, что позволяет быстро сформировать её заново для изучения иных реализаций MPI.

Модуль трассировщика

По событию (**Core_Magic_Instruction)**, вызванному исполнением симулятором «магической инструкции», вызываются его обработчики. Каждому обработчику доступны регистры и адресное пространство процесса, что используется для считывания аргументов функций из заранее подготовленного массива (как было описано выше).

Стоит учесть, что при симуляции двух и более машин в пределах одного процесса Simics для всех регистрируется лишь один обработчик **Core_Magic_Instruction**. Поскольку симуляция машин происходит в отдельных потоках симулятора, возникает проблема гонки при доступе к общим ресурсам. В данном случае таким ресурсом являлся объект открытого файла, куда записываются результаты. Поэтому была использована взаимная блокировка при записи в файл.

Во время тестовых запусков было замечено большое число (на 3-4 порядка большее, чем других процедур) количество вызовов **MPI_Iprobe** – неблокирующей проверки наличия сообщения в очереди. Запись каждого такого вызова заметно снижала производительность симуляции из-за дисковых операций; это также значительно увеличивало размер генерируемых файлов. Потому было решено реагировать только на каждый десятитысячный её вызов.

Постановка эксперимента

Эксперименты по моделированию проводились на вычислительном кластере МФТИ, содержащем 16 узлов, каждый из которых имеет 2 процессора Intel Xeon E5680 с шестью ядрами; таким образом, полное число ядер равняется 192. Распределение вычислительных ресурсов на нём обеспечивается программой SLURM (*англ*. Simple Linux Utility for Resource Management). Она позволяет выделять ресурсы по запросу. Так, команда «**salloc –N 1 –time=24:00:00 ./myscript.sh»** выделит один узел на 24 часа для выполнения программы **myscript.sh**. Из-за того, что требуется собрать трассы программ для нескольких запусков различных конфигураций, а один эксперимент может выполняться несколько суток, то необходима автоматизация, минимизирующая необходимость вмешательства оператора. Для этого были выполнены следующие шаги:

1. Интеграция Simics со SLURM. Сопутствующие скрипты симулятора были изменены для того, чтобы автоматически определять количество и список выделенных SLURM узлов и распределять задачу по ним (Рис. 3).
2. Для проведения серий из большого числа экспериментов с различными конфигурациями для автоматизации процесса сбора информации была создана

пакетная задача, ставящая отдельные запуски в очередь SLURM и архивирующая их результаты по завершении.

Simics использует язык Python для описания модели системы, что облегчило интеграцию со SLURM, также имеющему интерфейсы к этому языку.

Для каждой запускаемой копии модели можно было установить различные значения конфигурационных элементов, таких как частота процессора, его тип (модель, количество ядер), объём ОЗУ, количество моделируемых узлов.

Автоматизирован ввод команд через управляющий терминал (приглашение Bash) модели. Типичный запуск моделируемого приложения требовал ввода команды вида "**mpiexec.hydra –np 256 –f hosts ./a.out**" в определённый момент.

Длительность процесса сбора полезных данных о поведении MPI-приложения варьировалось в зависимости от масштаба исследуемой системы. Для небольших моделей: 32 ядра на каждом из 5 узлов и 400 симулируемых секундах, в течение которых собирались трассы – оно равнялось 4 часам, при этом сама симуляция помещалась на единственной реальной машине. Для больших систем (более 14 узлов) было необходимо выделять суммарно 30-50 Гб физической памяти, поэтому эксперимент был распределён на несколько узлов в количествах от 2 до 5. Полное время симуляции при этом достигало 12 часов, замедление модели по отношению к реальной скорости работы приложений находилось в диапазоне от 400 до 600 раз.

Результаты

Для HPL были произведены эксперименты на модели 8 двухъядерных систем, при этом они размещались на 2 физических узлах (каждая ЭВМ содержала один процесс Simics, они связывались между собой по сети). Частота вызовов отдельных MPI-функций представлена на

. Согласно этим данным, HPL использует только процедуры коммуникаций «точка-точка», среди них больше всего вызовов **MPI_Iprobe** (на 3-4 порядка больше, чем **MPI_Recv/MPI_Send**). Происходит достаточно много вызовов MPI-процедур, не связанных с коммуникациями: **MPI_Type_free, MPI_Type_struct, MPI_Type_commit**. В настоящее время проводится сбор частот вызовов MPI-функций с помощью сборщика трасс, входящего в пакет [7]. Это позволит сравнить и верифицировать полученные результаты двух различных сборщиков трасс.

После завершения наладки процесса сбора данных все полномасштабные эксперименты проводились для приложения **mdrun**, для которого изучались различные статистические характеристики поведения приложения при разном количестве узлов, входящих в симуляцию. На рис.4 приведена зависимость средней дистанции между MPI вызовами изучаемого MPI-взаимодействия от количества параллельных потоков, участвующих в работе изучаемого приложения. Это позволит проанализировать величину CPI, как было сказано выше и как описано в работах [1, 2], и, таким образом, проанализировать производительность распределенной машины. Максимальный симулируемый кластер в данной серии экспериментов содержал 48 узлов, при этом количество выделенных под него физических узлов равнялось пяти, т.е. «плотность» размещения равнялась около 10 моделируемых ЭВМ на одну физическую.

Заключение

Задача анализа производительности сети распределенных машин представляет большой практический интерес. В связи с растущими потребностями в вычислительных мощностях, важность задачи будет возрастать. В настоящее время способы анализа производительности сети не обладают достаточной гибкостью для анализа еще не созданных, распределенных систем. Трассирование процедур с использованием симуляторов платформ позволяет анализировать сетевую производительность произвольных конфигураций распределённых систем.

Список литературы

1. **Simonson L.J., He L**. Micro-architecture Performance Estimation by Formula // Proceedings of SAMOS'05 – 2005 – Pp. 192–201.
2. **Речистов Г. С., Иванов А. А., Шишпор П. Л., Пентковский В. М.** Симуляционный подход для нахождения производительности параллельных MPI-приложений на вычислительном кластере // Труды 54 научной конференции МФТИ «Проблемы фундаментальных и прикладных естественных и технических наук в современном информационном обществе». 2011. С. 82–83.
3. **Riesen, R.** A Hybrid MPI Simulator. // Barcelona - 2006 IEEE International Conference on Cluster Computing - 2006.
4. **Sundeep P., Rajive L. B.** MPI-SIM: using parallel simulation to evaluate MPI programs. Los Angeles: WSC '98 Proceedings of the 30th conference on Winter simulation, 1998.
5. **Kunkel, J.** HDTrace – A Tracing and Simulation Environment of Application and System Interaction // Hamburg. University of Hamburg – 2011.
6. **HPL** - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. [Электронный ресурс] // Netlib Repository at UTK and ORNL <http://netlib.org/benchmark/hpl/> (26.06.2012).
7. **MPICH2**: High-performance and widely portable MPI. [Электронный ресурс] // Mathematics and Computer Science Division, Argonne National Laboratory <http://www.mcs.anl.gov/research/projects/mpich2/> (26.06.2012).
8. **Van Der Spoel D. et al**. GROMACS: Fast, flexible, and free // Journal of Computational Chemistry. – 2005. – T. 26, No 16. – pp. 1701–1718.
9. **Wind River Systems Inc**. Simics Accelerator Guide. s.l. : Wind River, 2012.
10. **Wind River Systems Inc**. Understanding Simics Timing. s.l. : Wind River, 2012.
11. **Uhlig R., Fishtein R., Gershon O., Hirsh I., Wang H.** SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture // Intel Technology Journal. — 1999. — Pp. 112–126.
12. **QEMU**: Open source processor emulator. [Электронный ресурс] <http://wiki.qemu.org/Main_Page> (26.06.2012)
13. **Dongarra J.** The MPI Profiling Interface. [Электронный ресурс] // MPI: The Complete Reference <http://www.netlib.org/utk/papers/mpi-book/node182.html> (16.04.2012).

Список подрисуночных подписей

14. Общая схема работы MPI-трекера при симуляции.
15. Распределение частот вызовов MPI процедур при работе HPL.
16. Процесс распределения копий Simics при старте симуляции на узлах, выделенных с помощью SLURM.
17. Зависимость среднего гармонического для длины интервала вычислений между двумя соседними вызовами MPI-функций от полного числа MPI-ранков в приложении **mdrun**. Каждый узел содержит 16 ранков, полное число узлов варьируется от 10 до 80.

Рис. 1

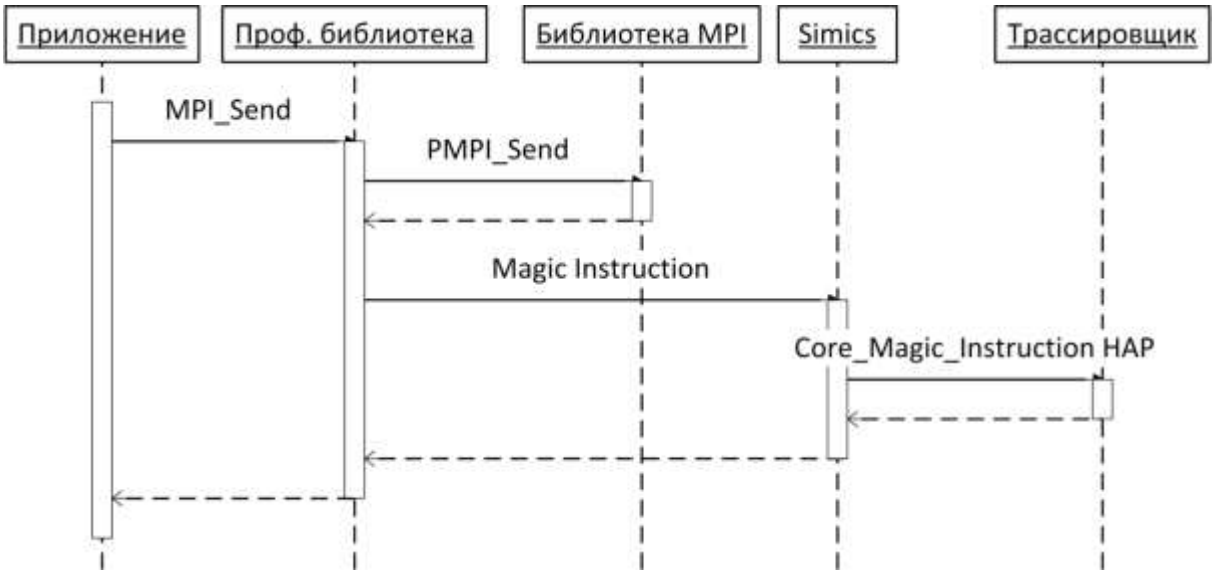

Приложение
Проф. библиотека
Библиотека MPI
Simics
Трассировщик
MPI_Send
PMPI_Send
Magic Instruction
Core_Magic_Instruction HAP

Рис. 2

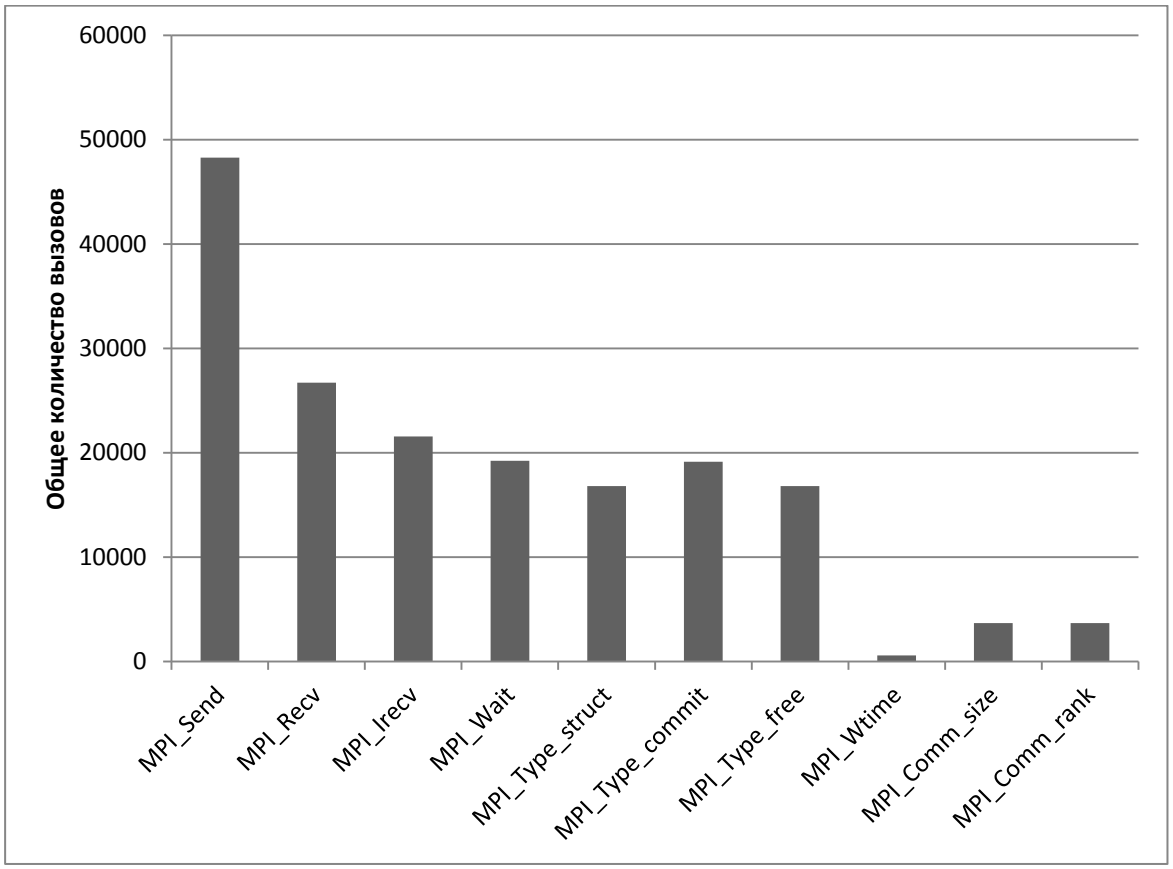

60000
50000
40000
30000
20000
10000
0
Общее количество вызовов
MPI_Send
MPI_Recv
MPI_Irecv
MPI_Wait
MPI_Type_struct
MPI_Type_commit
MPI_Type_free
MPI_Wtime
MPI_Comm_size
MPI_Comm_rank

Рис. 3

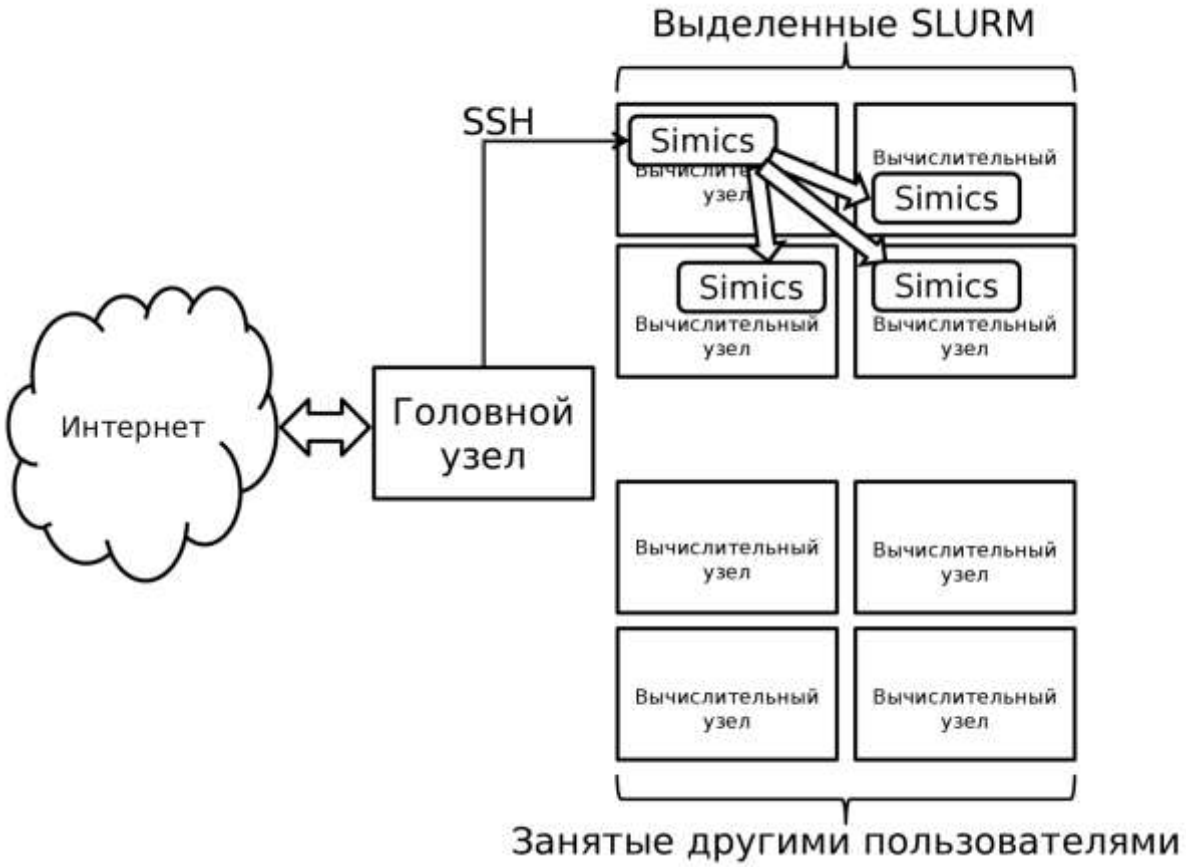
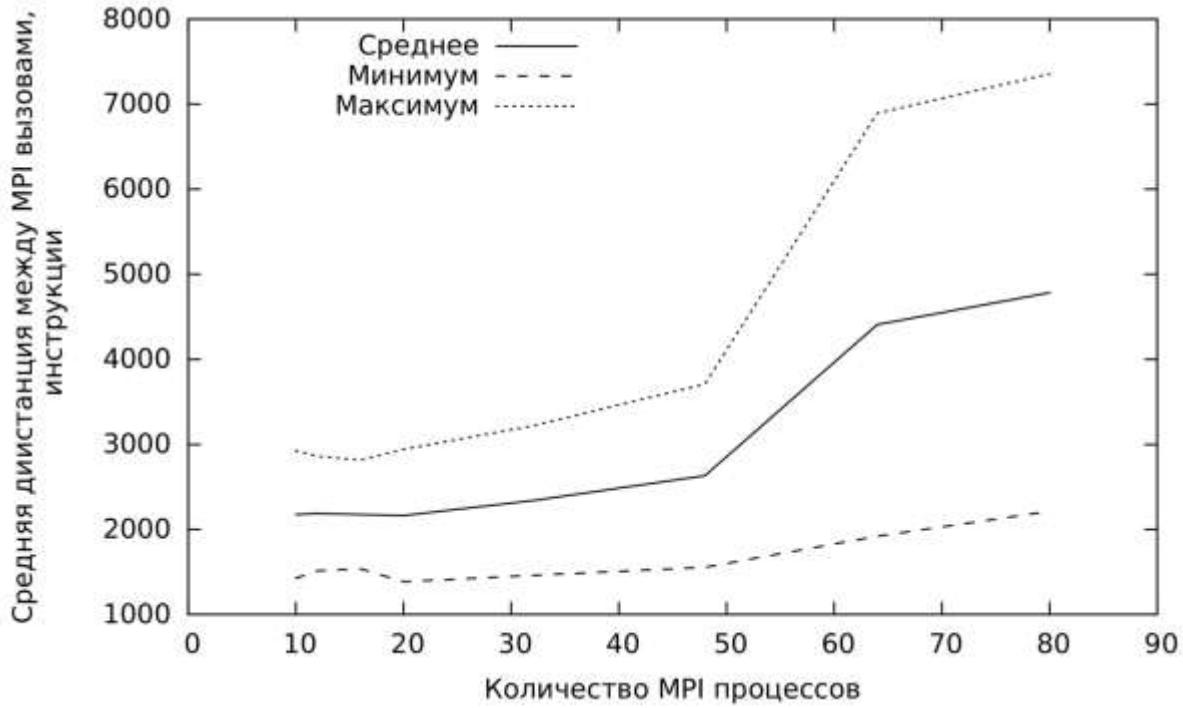

Выделенные SLURM
SSH
Simics
Вычислительный
Simics
Вычислительный
узел
Simics
Simics
Вычислительный
узел
Вычислительный
узел
Интернет
Головной
узел
Вычислительный
узел
Вычислительный
узел
Вычислительный
узел
Вычислительный
узел
Занятые другими пользователями

Рис. 4


Среднее
Минимум
Максимум
Средняя дистанция между MPI вызовами, инструкции
Количество MPI процессов
8000
7000
6000
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90

N. Polivanov

G. Rechistov

A. Abdukhalikov

V. Pentkovskiy

IMPLEMENTATION OF TOOLS FOR RESEARCHING NETWORK PERFORMANCE OF MPI APPLICATIONS ON A DISTRIBUTED SIMULATOR

Abstract

We present an approach to use the distributed full platform simulator Simics for collecting traces of MPI functions for applications for the offline analysis. The experimental setup and results of analysis of traces for the Linpack benchmark implementation HPL are given.

Key words: distributed simulation, Simics, multi core systems, network performance, tracing, cluster, MPI, Linpack.