Федулов Я.А., Федулова А.С. Анализ использования профилировщиков программного продукта для параллельного кода в системе вычислительного кластера // Международный журнал информационных технологий и энергоэффективности. – 2020. – Т.5 № 1(15) с. 30-38

Международный журнал информационных технологий и энергоэффективности

Сайт журнала: http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.77

АНАЛИЗ ИСПОЛЬЗОВАНИЯ ПРОФИЛИРОВЩИКОВ ПРОГРАММНОГО ПРОДУКТА ДЛЯ ПАРАЛЛЕЛЬНОГО КОДА В СИСТЕМЕ ВЫЧИСЛИТЕЛЬНОГО КЛАСТЕРА

[1]Федулов Я.А., [2]Федулова А.С.
Филиал ФГБУ ВО «НИУ «МЭИ» в г. Смоленске, Смоленск, Россия (214013, г. Смоленск, Энергетический проезд,1), e-mail: [1]fedulov_yar@mail.ru, [2]voitsitskay@mail.ru

В работе представлен общий анализ существующих профилировщиков программ, позволяющих отследить время выполнения программного кода. Отражены простые программы с функциями для проверки выполнения профилирования. Использование каждого из способов вывода времени затраченного на выполнение отдельной функции, позволяет сделать вывод о необходимости использования стандартных или специализированных продуктов профилирования программ для оптимизации участков кода, что существенно позволит облегчить работу программисту.

Ключевые слова: профилирование, профилировщик программного продукта, статистические профайлеры, компилятор, кластер, параллельное программирование, динамический анализ.

THE ANALYZE OF A USAGE OF PROFILERS FOR THE PARALLEL CODE IN A COMPUTER CLUSTER SYSTEM

[1]Fedulov Ya.A., [2]Fedulova A.S.
Smolensk Branch of the National Research University "Moscow Power Engineering Institute", Smolensk, Russia (214013, Smolensk, Energetichesky proezd, 1), e-mail: [1]fedulov_yar@mail.ru, [2]voitsitskay@mail.ru

The paper presents a general analysis of existing program profilers, which track the execution time of program code. Simple programs show with functions for checking profiling. The use is demanded for each method of outputting the time spent on performing a separate function, it allows us to conclude that it is necessary to use standard or specialized products for profiling programs to optimize sections of code, which will greatly facilitate the work of the programmer.

Keywords: profiling, software product profiler, statistical profilers, compiler, cluster, parallel programming, dynamic analysis.

В настоящее время используется большое количество многопроцессорных систем для расчетов больших данных и для повышения производительности вычислительных систем [1]. Разработчикам программы важно понять особенность выполнения кода, чтобы выполнять его эффективно и задействовать большинство доступных вычислительных ресурсов. Это особенно актуально для параллельных программ, где требуется решение основных вопросов, таких как: загрузка, балансировка, синхронизация и связь узловых потоков. Многопроцессорные вычислительные системы, использующие в своем составе несколько вычислительных узлов с 2-мя или более процессорами, далее называемые вычислительными кластерами.

При использовании вычислительных кластеров требуется более высокий уровень программирования для использования всех возможностей используемой системы. Одним из важных критериев программного кода является эффективность программного кода, необходимо отслеживать результат распараллеливания программы, чтобы выполнение осуществлялось быстрее и использовалось больше ресурсов предоставляемого оборудования. Производительность параллельной программы зависит от модели, для которой код разработан. Ряд инструментов различной сложности и мощности были разработаны для параллельных языков программирования и систем [2]. Инструменты для сбора данных о производительности сохраняют полученную информацию в виде трасс событий или файле. Трассировка дает более подробный анализ временных характеристик, но является наиболее трудоемкой. Профилирование обладает важным преимуществом по сравнению с трассировкой, так как выдает время, затраченное при вызове каждой из функций программного кода.

Целью исследования является изучение известных методов профилирования параллельных программ и экспериментальное подтверждение преимущества стандартных средств профилирования для удаленного исследования параллельного кода.

Материал и методы исследования: проведение экспериментов на гибридном вычислительном кластере ФГБОУ ВО НИУ МЭИ в г. Смоленске (ГВКС) с использованием технологии распараллеливания передачи информации, которая позволяет обмениваться сообщениями между процессами в параллельных программах для получения данных от установленных профилировщиков.

Результатами исследования будут поставленные эксперименты и отчеты о профилировании параллельных программ.

Описание экспериментов

Для получения оптимального времени выполнения программы, необходимо учитывать работу программного кода, который обладает рядом ограничений, которые необходимо учесть при дальнейшей работе.

Профилирование – это процесс осуществления сбора данных о работе программы, об эффективности выполнения кода, выделение мест, в которых программа выполняется дольше всего, а также выводы о дальнейшем усовершенствовании программного кода на определенных его участках для дальнейшей оптимизации. За анализ работы программы отвечает профилировщик (профайлер), а для получения данных реального времени используется динамический анализ выполнения программы. Динамический анализ программного кода представляет собой анализ программы при ее выполнении. К основным этапам динамического анализа относят следующие: подготовка данных, тестовый запуск программного кода и сбор определенных параметров, анализ полученных данных.

Характеристики параллельной программы – это выделение функций программы, для определения времени выполнения вычислений и обменных операция. Профилировщики сохраняют последовательность событий и временные метки в специальных файлах gmon.out.

К самым известным современным профилировщикам относятся: стандартный профилировщик Linux – Gprof [3], Intel VTune Profiler [4], DTrace [5], Intel Threading Analysis Tools [6], Intel Trace Analyzer and Collector 7.1 [7], Pallas Gmbh [8], EDPEPPS [9].

Выбор профилировщиков, из представленных выше (используемых для определения участков программного кода, где тратится много времени, и которые должны быть преобразованы), зависит в первую очередь от установленного программного обеспечения, а также от установленных программ. Профилирование может проводиться для любых сфер деятельности, где необходимо ускорение работы программного кода за счет распараллеливания и точное определение мест для распараллеливания.

Выделяются наборы базовых показателей, которые учитываются с помощью профилировщиков [10].

1. Общее время выполнения каждого участка программы.

2. Удельное время выполнения каждого участка программы.

3. Выявление причин конфликтов и пенальти.

4. Получение количества вызовов определенного участка программы.

5. Расчет степени покрытия программы.

Общее время рассчитывается для выполнения каждой точки программы, и также отслеживаются времязатратные участки. При этой операции получения времени профилировщик отразит, что 99 % общего времени выполнения программа работает в функции main(). Для упрощения работы и понимания программистов профайлеры выводят время, за которое осуществлялось выполнение дочерних функций.

Удельное время выполнения команд (растактовка). Профилировщик VTune позволяет рассчитать данный показатель.

В статье представлен анализ установленных профилировщиков на гибридном вычислительном кластере ФГБОУ ВО НИУ МЭИ в г. Смоленске (ГВКС), работа профайлеров проверена с помощью написанных программах для параллельных технологий MPI и OpenMP.

Как правило, большинство профилировщиков используются только для вычисления общего времени, но большинство из них могут выполнять и другие функции для удобства понимания времязатратных мест кода, но когда нет необходимости использовать сложные в понимании и установке, поэтому достаточны стандартные ресурсы. Для этого необходимо знать основные особенности выбранных профайлеров, представленные далее. На ГВКС можно использовать большое количество профайлеров, но из-за сложности установки на кластере был выбран стандартный gprof, strace и Intel VTune Profiler для технологии MPI, так как является более сложной и интересной для рассмотрения.

Рассмотрим основные характеристики технологии MPI (Message Passing Interface). MPI – это библиотека функций при работе с параллельными процессами с помощью передачи сообщений. Данная технология на сегодняшний день является самым востребованным стандартом обмена сообщениями для параллельных программ. Множество реализаций MPI разработано для компьютерных платформ. Применяется при написании программного кода для кластеров. Стандарт MPI передачи сообщений важен для корректной работы приложения на платформе, а также на системе пользователя. Данная технология используется в различных языках Fortran, Java, C, C++. MPI ориентирован на работу с распределенной памятью, а OpenMP – на работу с общей памятью.

Для оптимизации времени выполнения программы необходимо выделить участки кода, которые чаще всего используются, выделение времени выполнения функций. Из этого можно сделать вывод, что необходимо точно знать места кода, где больше всего затрачивается время в программе при реальных входных данных, для эффективной ее оптимизации. Для этого будут рассмотрены различные инструменты работы с программным кодом и выделены достоинства и недостатки использования каждого из них. В статье представлены наиболее известные способы получения данных о затраченных ресурсах, а также задействование стандартных ключей компиляторов для других компоновщиков с несвойственными применениями, позволяющих получить требуемые результаты.

1. Gprof

Gprof (GNU-профайлер) – стандартная программа профилирования для компилятора набора GCC(GNU Compiler Collection) GNU(GNU is Not Unix) проекта по разработке свободного программного обеспечения (СПО) для различных аппаратных и программных платформ, а также для высокопроизводительных систем (кластеров), а также осуществляется поддержка большого количества языков C, C++, Objective-C, Fortran, Ada, Go, D, и библиотек данных языков (libstdc ++).

Gprof-профилировщик разработан для компилятора gcc и имеет различные ключи для компилятора такие как –pg, -g, -b, каждый из которых является необходимым для получения

требуемой информации [3]. Главная особенность рассмотренного способа заключается в использовании стандартной конструкции компилятора GCC для компилятора другой технологии MPI (компилятор mpicc), информации о работе данного компилятора большое множество, но об использовании стандартных профилировщиков немного информации [11, 12]. Был сделан вывод, что если mpicc является частью набора GCC, то и ключи компилятора gcc будут выполняться и на mpicc.

Тестирование MPI-профилировщика было проведено на программном коде параллельной программы, с помощью которой осуществляется вычисление времени выполнения обменных операций и вывода адресов переменных, используемых в программе, для этого были применены функций MPI. Программа написана на языке высокого уровня С.

Для проверки работы были использованы опции для профилирования программы –pg, -g при компилировании (получении испольняемого файла):

mpicc -pg main.c -o main.ex –g,

где опция –pg для создания исполняемого файла с настройками для генерации профилирующих данных. В результате будет создан исполняемый файл, который необходимо запустить с помощью функции mpirun

mpirun -n 32 main.ex 33 4,

где –n – число потоков, на которых выполняется программный код и будет получен файл отчета gmon.out, 33 и 4 – числа, которые меняются друг с другом большое количество раз. Профайлер подсчитывает время участков кода, в которых задерживалась программа, совмещает данные с графом вызовов и результат заносит в отчет (файл gmon.out). После выполнения

gprof main.ex gmon.out>main_L.txt

main_L.txt – файл, в котором записано время выполнения всех функций программы на рисунке 1.

```
                              Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.29      0.04     0.04        1    40.12    40.12  obmen
…
             Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 24.93% of
0.04 seconds

index % time    self  children    called     name
                0.04    0.00       1/1           main [2]
[1]    100.0    0.04    0.00       1          obmen [1]
-----------------------------------------------

<spontaneous>
[2]    100.0    0.00    0.04                  main [2]
                0.04    0.00       1/1           obmen [1]
                                   …
```

Рисунок 1 – Результирующий файл с временными замерами

Из результирующего файла видно, что функция obmen() вызвана из функции main() 1 раз и занимает «не все время» выполнения программы.

При добавлении –l будет получен файл с более подробной информацией компановки, представленный на рисунке 2.

```
                        gprof main.ex gmon.out>main_L.txt –l

                                    Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 75.22      0.03     0.03                             obmen (main.c:184 @ 40163b)
 25.07      0.04     0.01                             obmen (main.c:190 @ 4015cf)
  0.00      0.04     0.00        1     0.00     0.00  obmen (main.c:181 @ 401555)
…
                          Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 24.93% of 0.04 seconds

index % time    self  children    called     name
                0.00    0.00       1/1           main (main.c:130 @ 4012ec) [58]
[3]      0.0    0.00    0.00       1         obmen (main.c:181 @ 401555) [3]
Index by function name

   [3] obmen (main.c:181 @ 401555) [2] obmen (main.c:190 @ 4015cf) [1] obmen
(main.c:184 @ 40163b)
```

Рисунок 2 – Результирующий файл с подробными временными замерами

После выполнения был получен результат работы функций с использованием стандартных опций для иного вида технологий компилятора, что удобно и не требует дополнительных временных и денежных затрат.

Недостатком является неточность выдаваемых данных, что может быть важным для определенных задач.

Далее рассмотрим еще один стандартный профилировщик, он также будет использоваться для параллельной программы описанной ранее.

2. Strace

В вычислительной системе и написанных параллельных программах могут возникать ошибки или работа программы некорректна, то можно использовать стандартные средства системы Linux, позволяющие следить за работой каждой программы, системными вызовами, которые используются программой [14].

Команда strace позволят получать все системные вызовы программы, отправленные во время выполнения, а также время каждого системного вызова. Команда является более универсальной для Linux систем и позволяет получить общее время выполнения программного кода, но при этом получаются более громоздкие результаты, что не всегда удобно для восприятия.

Далее представлен краткий отчет, полученный при выполнении команды strace.

Для получения результирующего файла были выполнены следующие шаги компилирование программы с помощью стандартного компилятора MPI:

mpicc main.c -o main.exe.

Запуск программы с помощью команды strace со специальными опциями –ttt (время выполнения системных вызовов в формате UNIX) и –T (время выполнения вызова):

strace -ttt -T -o strace.txt mpirun -n 32 main.exe 12 15.

Все подробно описано, так как мало информации о совместном использовании команды strace и компилятора mpicc (14) на рисунке 3 представлен отчет с временными выводами.

```
1585743740.490193   execve("/usr/lib64/openmpi/bin/mpirun",   ["mpirun",   "-n",   "32",
"main.exe", "12", "15"], [/* 52 vars */]) = 0 <0.000464>
1585743740.491041 brk(NULL)             = 0x22cc000 <0.000017>
1585743740.491141 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7effa3c5b000 <0.000018>
…….
1585743757.531581 close(11)             = 0 <0.000007>
1585743757.531602 rmdir("/tmp/openmpi-sessions-voitsitskay_as@mng1_0") = 0 <0.000030>
1585743757.531647   openat(AT_FDCWD,   "/tmp/openmpi-sessions-voitsitskay_as@mng1_0/14331",
O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC)  =  -1  ENOENT  (No  such  file  or  directory)
<0.000027>
1585743757.531714 munmap(0x7effa15a5000, 2127344) = 0 <0.000018>
1585743757.531760 munmap(0x7effa1bb6000, 2101648) = 0 <0.000013>
1585743757.531811 munmap(0x7effa17ad000, 2110072) = 0 <0.000015>
1585743757.532211 exit_group(0)         = ?
```

Рисунок 1 – Результирующий файл с подробными временными выводами

Команда strace с опцией -c позволяет получить отчет о системных вызовах и определить самые вызываемые функции на рисунке 4.

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- -------- 30.73      0.039963
17      2287       304 openat
 12.77    0.016609           7      2353        32 read
 11.04    0.014360          15       974       959 newfstatat
  9.02    0.011723           5      2462           close
  5.81    0.007553         229        33           clone
  5.62    0.007305          22       326       139 open
  3.24    0.004214          13       329           write
  3.22    0.004187           6       663           getdents
  2.19    0.002845          38        75         1 wait4
  2.08    0.002699           4       746           fcntl
  2.00    0.002604           7       381        63 stat
  2.00    0.002599          31        84           poll
  1.05    0.001369           4       351           fstat
  0.78    0.001016           9       114           munmap
  0.77    0.001004           8       130           pipe
  0.77    0.001002           5       209           mmap
  0.75    0.000981          30        33           sched_setaffinity
…….
  0.00    0.000000           0         2           lseek
  0.00    0.000000           0         1           dup
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           getpgrp
------ ----------- ----------- --------- --------- --------
100.00    0.130035                 12540      1551 total
```

Рисунок 4 – Отчет о системных вызовах

Команда strace позволяет получить большое количество системных данных и является стандартной для Linux, но описание использования ее только для последовательных программ затрудняет работу и понимание, поэтому проведенные эксперименты на реальном вычислительном кластере позволили сделать выводы о возможности ее использования также для параллельных программ.

Основным недостатком является сложность получения информации для работы с параллельными программами. Дальнейшая работа с данной командой и технологиями распараллеливания позволит получить необходимые данные о совместной работе технологий.

Существуют специализированные пакеты для профилирования параллельных программ к ним относятся Intel VTune Profiler, Intel Parallel Studio XE, Intel System Studio [15], они являются платными и сложны в установке.

3. Intel VTune Profiler

Intel VTune Profiler – специализированный профилировщик, используемый для повышения производительности программы на высокотехнологичном оборудовании. Для получения эффективных улучшений необходимо обладать корректными данными для повышения производительности. Intel VTune Profiler отражает основные данные профилирования в понятном для пользователя интерфейсе, который позволяет легче проводить анализ и интерпретацию.

Профилировщик может быть использован для различных направлений и позволит оптимизировать программное обеспечение. К основным можно отнести высокопроизводительные вычисления прогнозирования погоды и биоинформатике, приложения для транспортировки и производства, медиа-программы для транскодирования видео и обработки изображений, облачные приложения, драйверы устройств, игровые движки.

Профайлер используется для Linux. Используется Intel VTune Profiler для профилирования последовательных и многопоточных приложений, которые выполняются на различных аппаратных платформах (CPU, GPU, FPGA), графический интерфейс позволяет отследить работу программы и проанализировать полученные результаты, рисунок 5 [4].

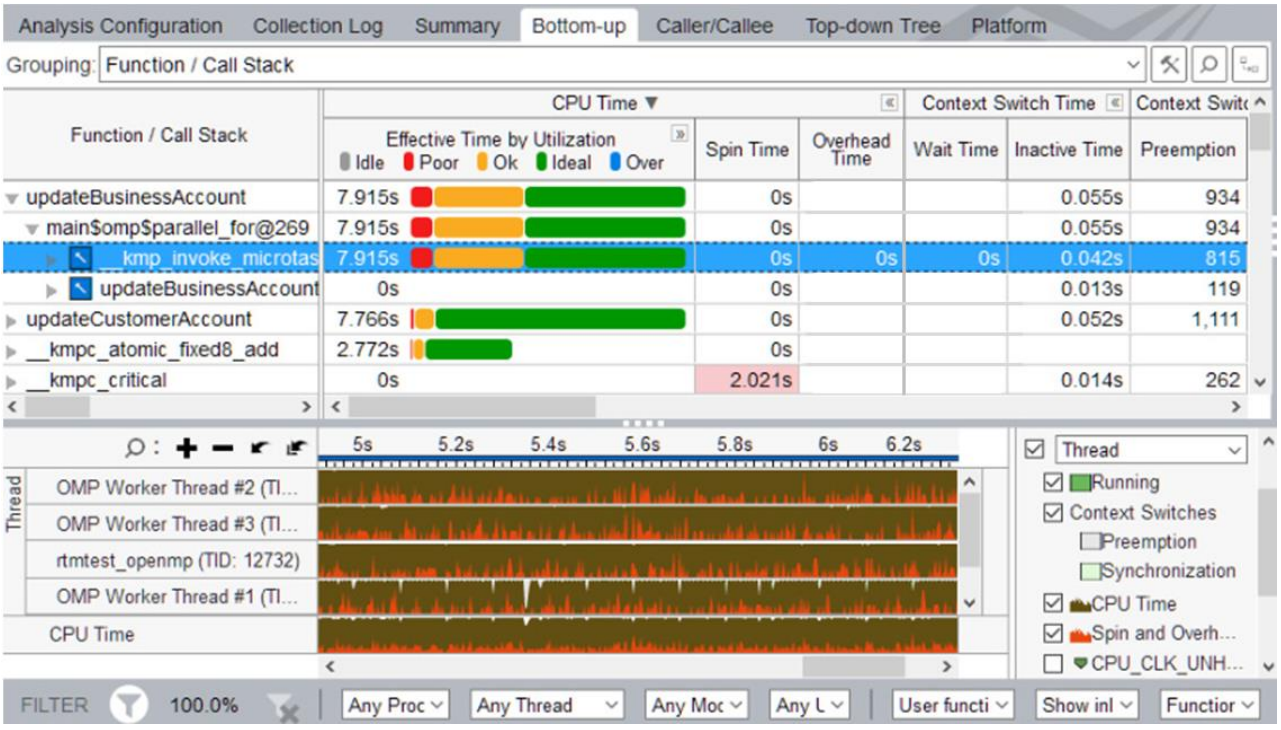

Рисунок 5 – Вывод результирующего файла при наличии графического интерфейса на вычислительном кластере

Данная программа удобна, если она закуплена и есть специалист, умеющий работать с данным обеспечением. Недостатком можно считать, что профайлер имеет графический интерфейс исключительно на машине вычислительного кластера, при удаленном подключении через PuTTY (клиент для различных протоколов удалённого доступа, таких как SSH, Telnet, rlogin) нет возможности видеть все графики, работать сложно и неудобно.

Поэтому для профилирования программ удобен и прост в использовании стандартный профилировщик компилятора Linux, который выдает результаты в файл, который можно

прочитать, разобрать и понять, что очень важно для удаленной работы с многопроцессорной вычислительной системой.

Заключение

В статье представлены результаты экспериментального исследования стандартных средств профилирования, а также их соединение с компиляторами, которые только теоретически могли использоваться совместно, но на практике были проведены необходимые тесты и получены требуемые результаты, были проверены на практике и работа стандартных средств профилирования для Linux, были отражены в отчетах о выполнении профилирования для анализа производительности параллельных программ.

Описаны сильные и слабые стороны исследованных стандартных и специализированных профилировщиков, сделаны выводы по наиболее приемлемому использованию вышеописанных инструментов анализа производительности высокотехнологичных систем в зависимости от технических и экономических возможностей, а также возможности удаленного контроля для исследования работы программ.

Список литературы:

1. Чайковский, Д.С.. Анализ современных средств профилирования параллельных программ/ Д.С. Чайковский, Н.А. Гулевич // Информационные технологии (Вестник СГТУ) – Саратов, 2014. – 109-113.
2. Matthias S. Müller. OpenMP Shared Memory Parallel Programming: International Workshops / Matthias S. Müller, Barbara M. Chapman // IWOMP 2005 and IWOMP – USA. 2006.
3. GPROF Tutorial [Сайт]. URL: https://www.thegeekstuff.com/2012/08/gprof-tutorial/, 27.03.2020
4. Intel VTune Profiler [Сайт]. URL: https://software.intel.com/en-us/vtune, 28.03.2020
5. Кляус С.М. Инструменты динамической трассировки DTrace и SystemTap, 2017, 220 с. URL: http://www.unilib.neva.ru/dl/local/407/oe/oe.ppt (дата обращения: 01.04.2020)
6. Intel® Threading Building Blocks [Сайт]. URL: https://software.intel.com/en-us/articles/intel-threading-building-blocks-threading-performance-and-correctness-analysis, 28.03.2020
7. Getting Started with Intel® Trace Analyzer and Collector on Linux* OS [Сайт]. URL: https://software.intel.com/en-us/get-started-with-itac-for-linux, 28.03.2020
8. Компании-разработчики программных средств [Сайт]. URL: https://parallel.ru/tech/software_vendors.html, 28.03.2020
9. Средства создания и проектирования параллельных программ [Сайт]. URL: https://parallel.ru/tech/tech_dev/build_par.html#edpepps, 29.03.2020
10. Касперски К. Техника оптимизации программ. Эффективное использование памяти. - СПб.: БХВ-Петербург, 2003. - 464 с.
11. Значимые результаты от gprof для кода MPI [Сайт]. URL: https://stackoverflow.com/questions/53794093/how-do-i-get-meaningful-results-from-gprof-on-an-mpi-code, 29.03.2020
12. Евсеев, И.. Использование MPI на компьютерах ЦСТ [Сайт]. URL: https://www.opennet.ru/docs/RUS/MPI_intro/using.html, 29.03.2020
13. Команда Strace в Linux [Сайт]. URL: https://losst.ru/komanda-strace-v-linux, 29.03.2020
14. Using Strace to Examine Your IO [Сайт]. URL: https://www.clustermonkey.net/FileSystems/using-strace-to-examine-your-io/Page-2.html, 29.03.2020
15. Choose the Best Option [Сайт]. URL: https://software.intel.com/en-us/vtune/choose-download, 01.04.2020

References

1. Tchaikovsky, D.S. Analysis of profiling parallel programs modern means / D.S. Tchaikovsky, N.A. Gulevich // Information Technologies (Bulletin of SSTU) - Saratov, 2014 .- 109-113.

2. Matthias S. Müller. OpenMP Shared Memory Parallel Programming: International Workshops / Matthias S. Müller, Barbara M. Chapman // IWOMP 2005 and IWOMP - USA. 2006.
3. GPROF Tutorial [Site]. URL: https://www.thegeekstuff.com/2012/08/gprof-tutorial/, 03/27/2020
4. Intel VTune Profiler [Site]. URL: https://software.intel.com/en-us/vtune, 03/28/2020
5. Klyaus S.M. Dynamic Tracing Tools DTrace and SystemTap, 2017, 220 p. URL: https://www.tune-it.ru/documents/10136/828092/dtrace_stap_book_b10a.pdf/66cae909-5677-40d2-a306-ca265392d142, 04.01.2020
6. Intel® Threading Building Blocks [Site]. URL: https://software.intel.com/en-us/articles/intel-threading-building-blocks-threading-performance-and-correctness-analysis, 03/28/2020
7. Getting Started with Intel® Trace Analyzer and Collector on Linux * OS [Site]. URL: https://software.intel.com/en-us/get-started-with-itac-for-linux, 03/28/2020
8. Software companies [Site]. URL: https://parallel.ru/tech/software_vendors.html, 03/28/2020
9. Tools for creating and designing parallel programs [Electronic resource]. URL: https://parallel.ru/tech/tech_dev/build_par.html#edpepps, 03/29/2020
10. Kaspersky K. Technique for optimizing programs. Efficient use of memory. - SPb .: BHV-Petersburg, 2003 .- 464 p.
11. Significant gprof results for MPI code [Site]. URL: https://stackoverflow.com/questions/53794093/how-do-i-get-meaningful-results-from-gprof-on-an-mpi-code, 03/29/2020
12. Evseev Il. Using MPI on computers TsST [Site]. URL: https://www.opennet.ru/docs/RUS/MPI_intro/using.html, 03/29/2020
13. Strace command in Linux [Site]. URL: https://losst.ru/komanda-strace-v-linux, 03/29/2020
14. Using Strace to Examine Your IO [Site]. URL: https://www.clustermonkey.net/FileSystems/using-strace-to-examine-your-io/Page-2.html, 03/29/2020
15. Choose the Best Option [Site]. URL: https://software.intel.com/en-us/vtune/choose-download, 04/01/2020.