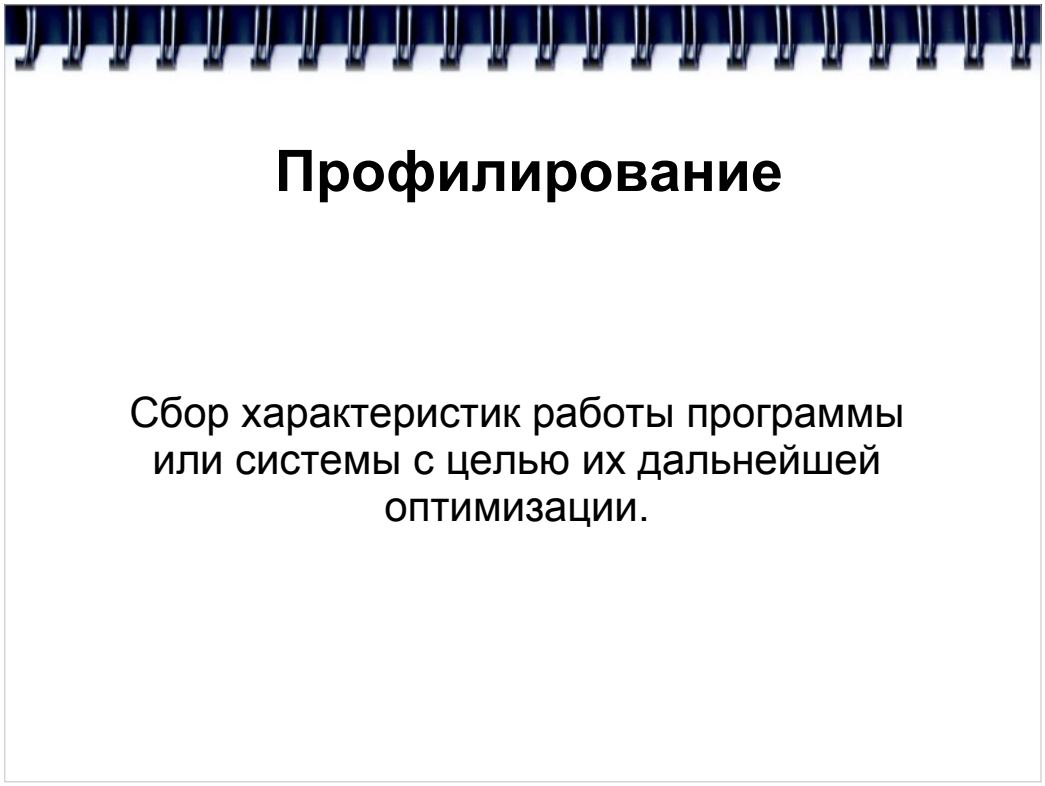
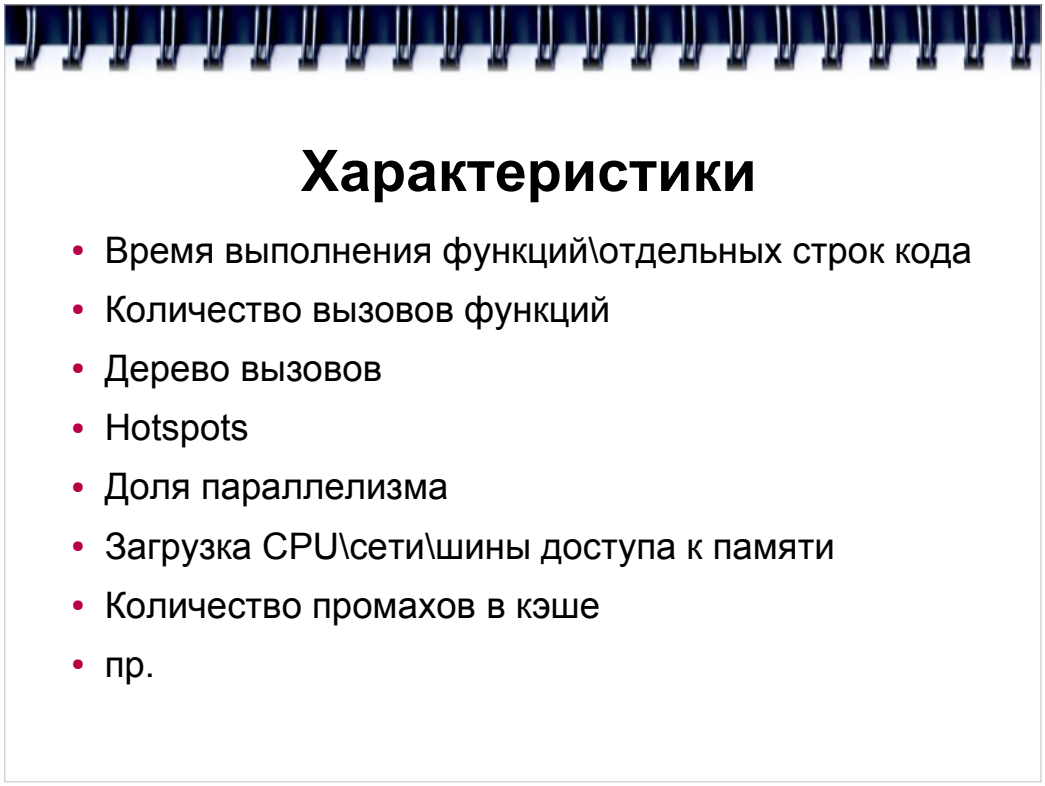
Профилирование программ

Алексей А. Романенко

arom@ccfit.nsu.ru



Сбор характеристик работы программы с целью понять на сколько эффективно работает программа и какие шаги и на каких участках программы стоит предпринять для дальнейшей оптимизации.



К интересующим характеристикам относятся такие как время выполнения отдельных фрагментов программы (от функций до отдельных ассемблерных инструкций), число верно предсказанных условных переходов, число кэш промахов и т. д.

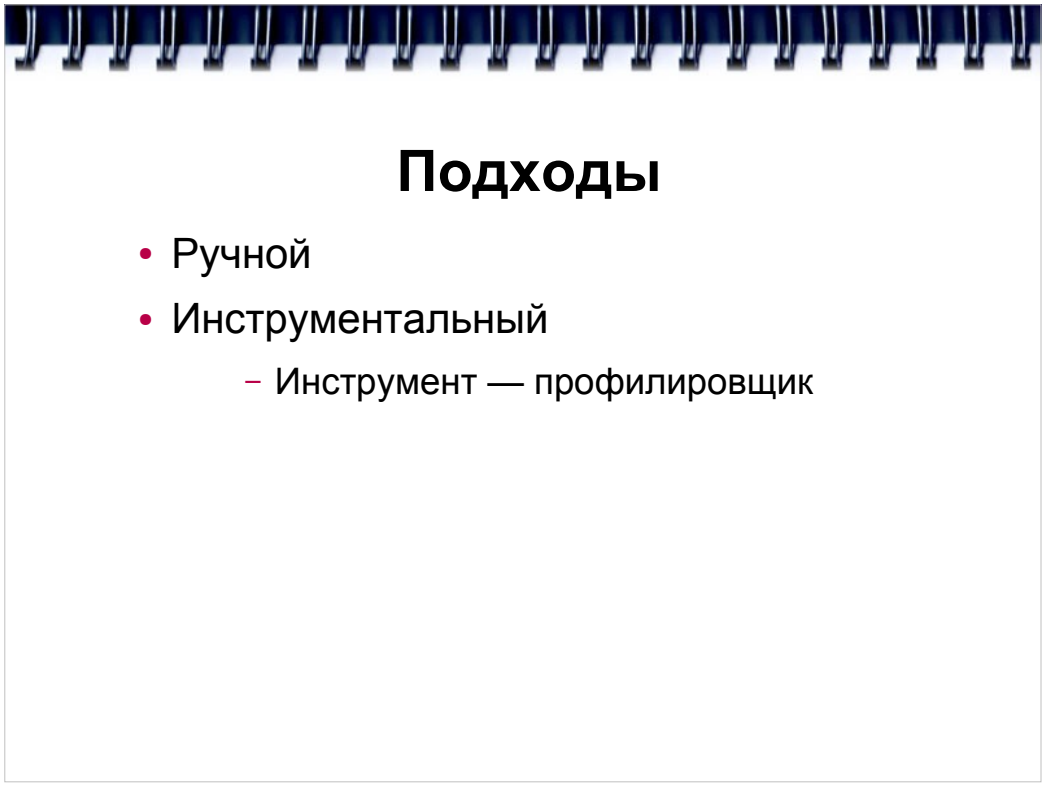
Время выполнения

```
Bash# date; wc -l /etc/passwd; date
Срд Июл 28 17:43:29 NOVST 2010
    564 /etc/passwd
Срд Июл 28 17:43:29 NOVST 2010

Bash# time wc /etc/passwd
    564    1348   42335 /etc/passwd

real    0m0.011s
user    0m0.000s
sys     0m0.010s
```

Простейший способ оценить эффективность работы программы — замерить время ее выполнения на разных тестах. Для этого в Unix системах есть утилита time, которая выдает время потраченое на выполнение программы пользовательское, системное и общее.

Системное время — время на чтение данных из файла, синхронизацию потоков, переключение контекста и пр., т.е. все то, что выполняется ядром ОС.

Пользовательское время — расчеты, мат. операции.



Ручной способ — расстановка замеров времени вручную.

Инструментальные средства анализа программы универсальны и позволяют собирать обширную информацию о поведение программы. Проектировщики ПО нуждаются в таких инструментальных средствах, чтобы оценить, как хорошо выполнена работа. Программистам они нужны, чтобы проанализировать их программы и идентифицировать критические участки программы.

Оба способа оказывают влияние на поведение программы.

Недостатки ручного способа профилирования

- Вставка дополнительного кода в программу → новые ошибки и необходимость чистки кода
- Анализ результата в текстовом виде
- Большой объем выходных файлов
- Отсутствие дерева вызовов

Макросы для ручного профилирования

```
#define newT(name)   int name ## _timer_linef; \
                     struct timeval name ## _tv_1; \
                     struct timeval name ## _tv_2;

#define startT(name) gettimeofday(&name ## _tv_1, NULL); \
                     name ## _timer_linef = __LINE__;

#define printT(name) gettimeofday(&name ## _tv_2, NULL); \
                printf("%s:%d through %d takes %f usec \n", \
                   __FILE__, name ## _timer_linef ,__LINE__, \
   (name ## _tv_2.tv_sec - name ## _tv_1.tv_sec)*1e6 + \
   (name ## _tv_2.tv_usec - name ## _tv_2.tv_usec));
```

Если программа небольшая и подозрения в одном-двух ее участках, то возможно проще вставить в код несколько строк для замера времени.

На сладе три макроса для объявления таймера, сброса и распечатки времени.

Пример использования

```
int main(...){
   ...
   newT(my_work);
   startT(my_work);
   do_something();
   printT(my_work);
   ...
}
```

```
test.c:23 through 31 takes 20.386 usec
```

Инструменты

какая часть программы нуждается в оптимизации*?*

- Профилировщик — инструмент анализа производительности программы. Сбор информации о поведение программы во время ее выполнения.
- Первые инструменты - начало 1970s
  - Использование прерываний по времени, сохоранение регистра PSW для поиска “hot spots” на IBM/360, IBM/370
- Вывод:
  - Trace
  - Sampling

Методы сбора данных

http://en.wikipedia.org/wiki/List_of_performance_analysis_tools

- Event based profilers
  - .NET, Java, Python, Ruby
- Statistical profilers
  - gprof
  - Oprofile
  - CodeAnalyst
  - VTune
  - Valgring
  - ...
- Hypervisor/Simulator
  - SIMMON, OLIVER

Инструментальных средств профилирования много и для сбора информации о работе программы они могут использовать разные подходы.

Пожалуй наиболее распространенным подходом является семплирование. Это подход, при котором через определенные промежутки времени профилировщик смотрит где находится программа и фиксирует это положение отождествляя его с исходным кодом.

DTrace

- Разработан Sun Microsystems
- Трассировка в режиме реального времени
- Более 50 000 точек проверки
- Язык программирования «D»

- http://en.wikipedia.org/wiki/DTrace

DTrace может использоваться для наблюдения за количеством потребляемой памяти, процессорным временем, файловыми системами и сетевыми ресурсами, используемыми активными процессами, на работающей системе. Также можно получить более детальную информацию, например, список аргументов, с которыми вызывается каждая функция, или список процессов, использующих определенный файл.

GProf

- GNU profiler
- Информация
  - Вызовы функций (количество/время)
  - Дерево вызовов (Call graph)
- Сборка программ
  - gcc -g -pg <прочие опции>
- Запуск программы → на выходе файл gmon.out

Gprof — один из простых профилировщиков, свободно доступных разработчикам ПО. Предназначен для профилирования программ на Си, Паскаль и Фортран

Программа ведет подсчет времени, которое программа провела в той или иной функции, накладывает информацию на граф вызовов и результат пишет в файл отчета (gmon.out)

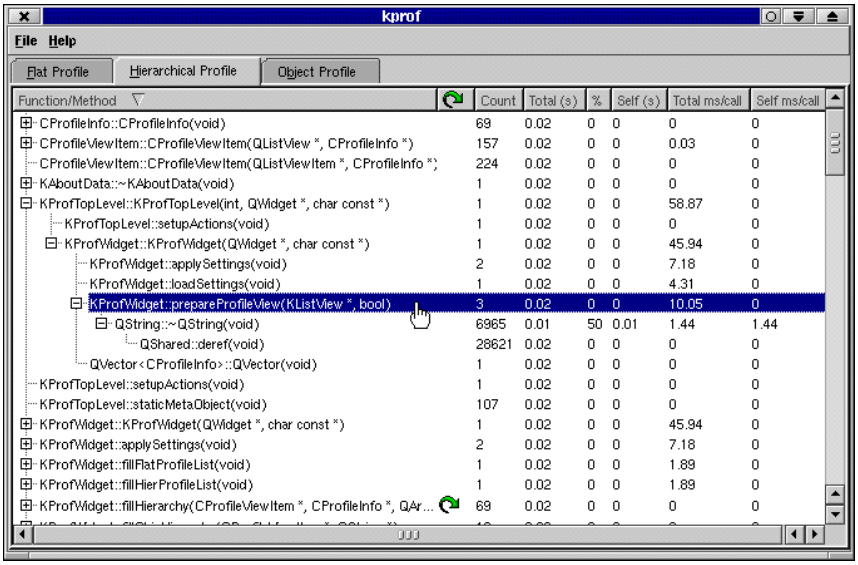
GProf. Пример

```
bash # gcc -O3 -g -pg test.c
bash # ./a.out
bash # ls
a.out    gmon.out    test.c
bash # gprof
...
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00      7.93     7.93        1     7.93     7.93  calculate
  0.00      7.93     0.00        1     0.00     0.00  init_depth
  0.00      7.93     0.00        1     0.00     0.00  init_h1_and_h2
  0.00      7.93     0.00        1     0.00     0.00  init_mareographs
...
index % time    self  children    called     name
                7.93    0.00       1/1           main [2]
[1]    100.0    7.93    0.00       1          calculate [1]
-----------------------------------------------
                0.00    0.00       1/1           init_params [6]
[3]      0.0    0.00    0.00       1          init_depth [3]
...
```

На результате профилирования видно, что функция calculate() вызывается из функции main() и занимает «все время» выполнения программы.

Немного более подробную информацию можно получить, если добавить ключ «-l» - вывод информации по строкам кода.

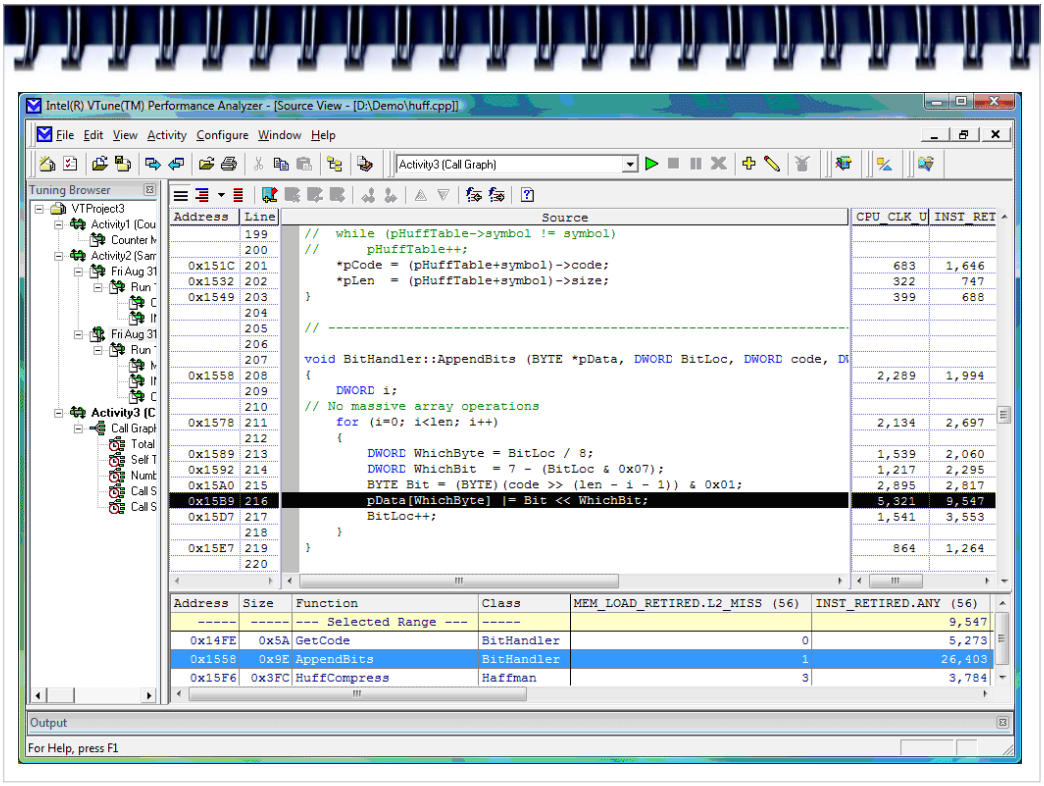
Kprof - GUI обертка для gprof



В KDE есть программа, которая способна разбирать текстовые файлы от gprof и представлять ее в удобном для пользователя виде.

VTune

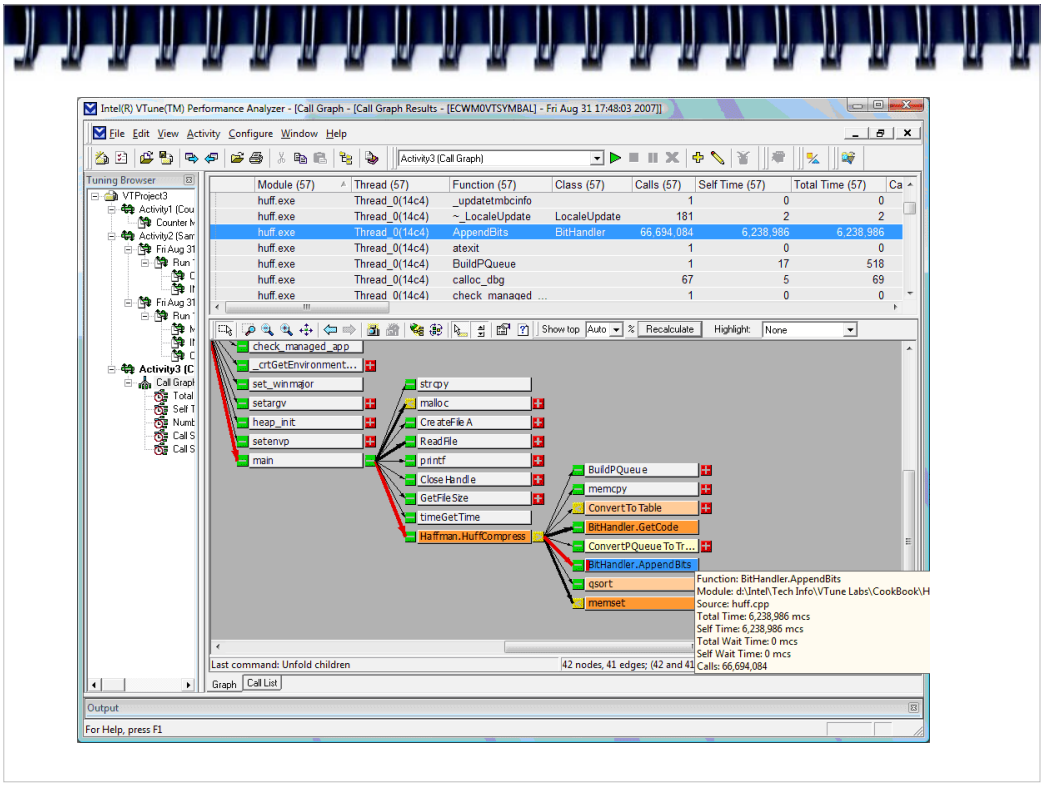
- Коммерческий продукт от Intel
- Информация
  - Дерево вызовов
  - Семплирование
  - Просмотр исходного кода
  - Мониторинг показаний
  - Профилирование потоков
- http://software.intel.com/en-us/intel-vtune/



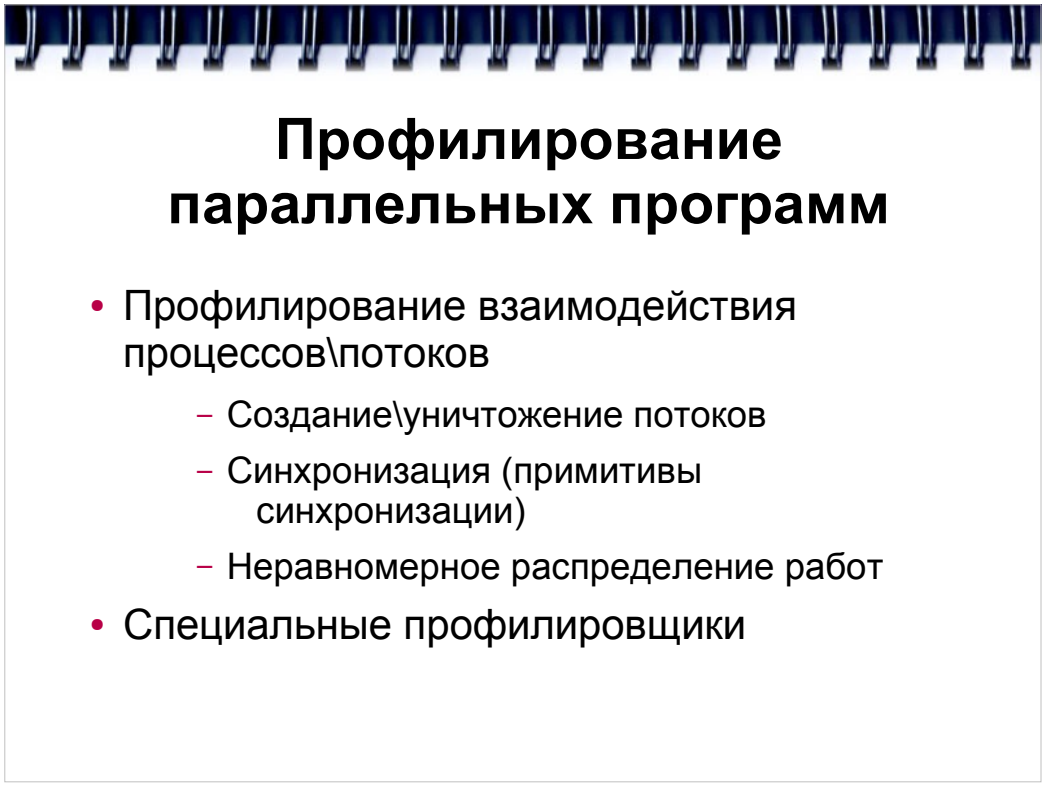
Присутствует возможность навигации как по исходному коду программ так и по ассемблерному коду. Для каждой из строчек кода присутствует информация о времени.

При выделении нескольких строчек можно получить суммарную информацию по ним.

В VTune встроен помошник, который после анализа кода может давать советы, как код можно оптимизировать.



Отображение графа вызовов сопровождается указанием критического пути (красные жирные стрелки).



Ситуация с многопоточными приложениями существенно другая, поскольку в силу вступают новые аспекты производительности, связанные с взаимодействием потоков, затратами на их создание и синхронизацию. Ускорение отдельных участков кода может не дать никакого прироста производительности, если, например, разработчик построил свое приложение таким образом, что основную часть времени потоки ожидают освобождения разделяемого ресурса. Также можно упомянуть такие распространенные в многопоточных приложениях проблемы как неравномерное распределение нагрузки и неэффективное использование примитивов синхронизации. Эти факторы могут привести к катастрофически низкой производительности приложения, вплоть до того, что многопоточная версия будет медленнее последовательной. Для анализа многопоточных приложений необходимы специальные инструменты, ориентированные на обнаружение проблемных ситуаций и последующей помощи программистам в их разрешении

Инструменты профилирования и отладки параллельных программ (MPI)

- HeNCE
- TRAPPER
- EDPEPPS
- GRADE
- AIMS (An Automated Instrumentation and Monitoring System)
- Vampir, VampirTrace
- Pablo Performance Analysis Toolkit Software
- Paradyn
- Jumpshot, Nupshot
- Puma
- CXperf

Профилировщики параллельных программ (SMP)

- Intel® Thread Profiler
  www.intel.com/cd/software/products/asmo-na/eng/286749.htm
- Intel® VTune Performance Analyzer
  www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm
- Intel® Threading Analysis Tools
  www.intel.com/cd/software/products/asmo-na/eng/threading/219785.htm
- Intel® Trace Analyzer and Collector 7.1
  http://www.intel.com/cd/software/products/asmo-na/eng/306321.htm