Лекция 4. Инструменты Intel для оптимизации отладки приложений на платформе Windows

Процесс разработки приложений на платформе Windows. Возможности Parallel Studio XE. Краткий обзор возможностей оптимизации компилятора Intel для повышения приизводительности нативного кода. Ключи компилятора для оптимизации кода под Atom Производительность Атома.

4.1. Введение

В настоящее время разработчикам приложений необходимы мощные и удобные инструменты для адаптации существующих или написания новых приложений, максимально использующих производительность многоядерных процессоров. Понятно, что часто графический интерфейс приложений написан с использованием Java или .Net языков. Тем не менее, те части приложений, которые требовательны к производительности (фильтры, кодеки, и т.д.), в большинстве случаев реализованы именно на C/C++, и именно в них важно добиться задействования всех возможностей процессора.

Сейчас уже нет сомнений, что дальнейшее увеличение производительности приложений будет достигаться за счет того, насколько хорошо эти приложения распараллелены и как хорошо они масштабируются с ростом процессоров в системе. Очевидно, что фактическим стандартным набором C/C++ разработчика является Microsoft Visual Studio. Intel Parallel Studio – это набор из нескольких инструментов, который является расширением Microsoft Visual Studio, и позволяющий за счет удобства использования и понятного интерфейса добиваться хорошей эффективности параллельных программ на многоядерных системах. Несмотря на то, что этот набор является plug-in'ом к Visual Studio, он полностью покрывает все этапы разработки приложения программистом, от создания скелета будущей параллельной программы до оптимизации релизной версии проекта (Рис. 4.1.1). В состав этого набора входят четыре отдельных продукта, каждый их которых используется на определенном этапе разработки, и каждый может быть установлен и интегрирован в Visual Studio как отдельно, так и всем пакетом сразу.

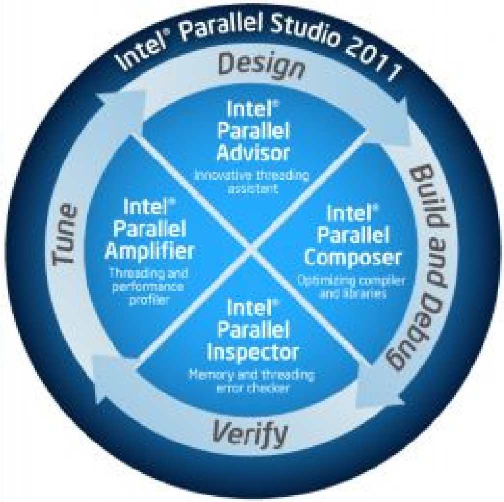

Рис. 4.1.1. Цикл разработки параллельной программы

В состав пакета входят:

- **Intel Parallel Advisor** помогает найти возможности распараллеливания кода с самого начала разработки приложения;
- **Intel Parallel Composer** предназначен для генерирования параллельного кода, т.е. создания программ с помощью компилятора и широкого набора библиотек для многопоточных алгоритмов;
- **Intel Parallel Inspector** проводит проверку параллельного приложения на корректность и находит ошибки работы с памятью;
- **Intel Parallel Amplifier** находит «узкие места» в программе, которые мешают масштабируемости и увеличению производительности на многоядерных платформах.

Так как цикл разработки может быть непрерывен и повторяем, важно, чтобы все эти инструменты слаженно работали в одной среде и могли взаимодействовать друг с другом. Именно поэтому они глубоко интегрированы в среду Microsoft Visual Studio, и когда нужно найти потоковые ошибки или оптимизировать программу, не приходится запускать отдельные профилировщики и создавать отдельные эксперименты. Весь функционал доступен в текущем проекте, из панели инструментов или меню Visual Studio (Рис. 4.1.2), а результаты анализа размещаются там же, где и файлы проекта. Хотя пользователь может и сам выбрать, где ему хранить файлы с результатами.

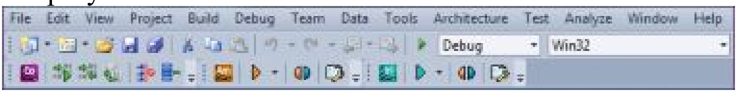

Рис. 4.1.2. Панель инструментов Intel Parallel Studio в среде Visual Studio

Весь функционал инструментов Intel Parallel Studio как бы является естественным продолжением Visual Studio и расширением ее возможностей, которых раньше недоставало для разработки хороших параллельных программ.

4.2. Intel Parallel Advisor

Существует два похода к написанию параллельных программ. Первый – это распараллеливание, частичное или полностью, уже готовых последовательных приложений для ускорения работы некоторых алгоритмов. В этом случае разработчик просто анализирует приложение и определяет участки программы, которые потребляют максимальное количество ресурсов процессора. Второй подход предполагает изначальный дизайн с учетом требований параллельного выполнения нагрузки. И если по существу проект можно разделить на участки, которые должны выполняться одновременно, то начать его реализацию в виде программы часто является сложной задачей для новичков. Особенно трудно написать проект так, чтобы потом не пришлось прибегать к первому подходу.

Здесь на помощь приходит Parallel Advisor. Это класс инструментов, который несет в себе методологию создания параллельных программ «с нуля» с использованием правильных подходов к их реализации, в том числе с использованием параллельных библиотек. Не каждую программу или алгоритм легко и просто распараллелить. Advisor находит, из-за чего именно параллельная реализация может оказаться неэффективной, и выдает нужные решения. Кроме того, все знания по применению параллельных библиотек собраны здесь в виде образцов и шаблонов, для того чтобы максимально облегчить начальный этап их использования. Однако и в случае с распараллеливанием готовой последовательной программы Advisor предоставляет «путеводную нить» (workflow) по распараллеливанию, проверке корректности и оптимизации приложения, если разработчик пока еще не в силах «уложить» в голове эту методологию. Но через какое-то время работы с проектом надобность в Advisor отпадает, так как принципы разработки параллельной программы становятся понятными.

4.3. Intel Parallel Composer

Intel Parallel Composer – это не только компилятор C++ от Intel. Он интегрирован в Visual Studio вместе с библиотекой производительности IPP (Integrated Performance Primitives) и параллельной библиотекой TBB (Threading Building Blocks), что значительно облегчает процесс разработки параллельного кода.

Наличие сразу нескольких компонент в пакете позволит сразу же начать оптимизировать свою программу с использованием параллельных технологий, которые содержит Composer:

- Вычислительные примитивы, реализованные в виде функций в библиотеке IPP, гарантируют высокую производительность алгоритмов;
- Поддержка новой версии стандарта OpenMP 3.0 позволяет использовать multitasking, недоступный в предыдущих версиях;
- Тип данных Valarray упрощает код, реализующий векторные операции, а компилятор генерирует эффективный бинарный код, задействующий для увеличения производительности SIMD-инструкции;
- Ну и наконец, встроенный параллельный отладчик Parallel Debugging Extention. Он является расширением к отладчику

Microsoft (Рис. 4.3.1) и позволяет анализировать данные, разделяемые между потоками OpenMP, акцентируя внимание на конфликтах доступа и возможных проблемах с корректностью параллельных вычислений. Подобной информации может быть собрано очень много, поэтому богатый набор фильтров позволяет отсеять не интересующие нас события.

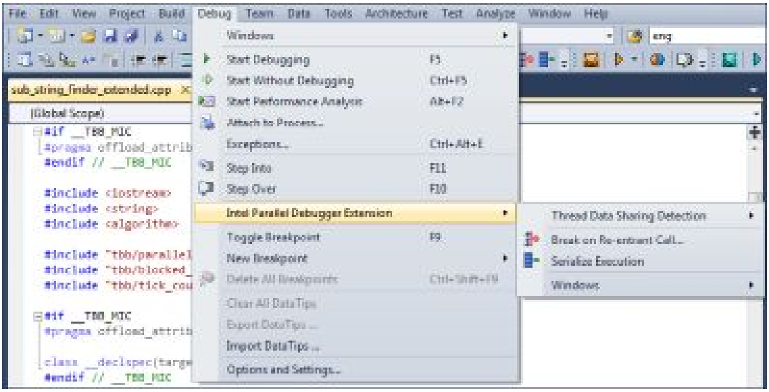

Рис. 4.3.1. Intel Parallel Debugger Extention.

Для того, чтобы быстро определить насколько корректны результаты вычислений, выполняемых в параллельном регионе, в отладчике можно выполнить любой регион последовательно, и сравнить потом результаты. Причем перекомпилировать ни приложение, ни исследуемый модуль для этого не потребуется.

4.4. Intel Parallel Inspector

Это, пожалуй, самый востребованный инструмент на сегодняшний день, так как он помогает избавиться от ошибок в многопоточной программе на этапе верификации, повышая корректность и стабильность ее исполнения. Несмотря на свой характерный функционал, Inspector применяется не только командами тестировщиков. Нормальная инженерная практика предполагает проверку программы на наличие ошибок и самим разработчиком, хотя бы на уровне юнит-тестов.

Parallel Inspector помогает обнаружить два класса ошибок: ошибки многопоточности и ошибки работы с памятью, причем анализ для каждого класса запускается отдельно. Последний класс ошибок хорошо известен программистам, которые до последнего времени использовали различные инструменты, чтобы найти утечки памяти, нарушение целостности стека или доступ по несуществующим адресам. Второй класс ошибок связан с многопоточной природой программ. Они неизбежно возникают при разработке параллельных приложений, и их чрезвычайно сложно отловить, особенно если они проявляются нерегулярно и только при совпадении определенных условий.

4.4.1. Ошибки работы с памятью

Механизм обнаружения ошибок памяти основан на анализе абсолютно всех инструкций чтения/записи и их адресов на уровне бинарного кода с помощью бинарной инструментации. Понятно, что всесторонний анализ исполняемого кода не может быть проведен без существенных накладных расходов, поэтому он разделен на уровни, отражающие глубину и сложность анализа. Чем выше уровень, тем больше потребуется времени для проверки приложения.

- **Уровень mi1** – позволяет обнаруживать только утечки памяти, выделенной в куче (heap). Глубина стека функций равна 7, что даст достаточно информации для определения местонахождения ошибки и структуры вызовов функций, выделявших память.
- **Уровень mi2** – позволяет обнаруживать все остальные ошибки работы с памятью в куче. Однако для снижения накладных расходов и ускорения анализа глубина стека равна единице. То есть на этом уровне можно узнать, есть ли в принципе ошибки в программе. А где эти ошибки, помогает определить следующий уровень.
- **Уровень mi3** – отличается от предыдущего тем, что глубина стека увеличена до 12-ти. Плюс добавлен функционал поиска утерянных указателей. На этом уровне мы доступен наиболее полный анализ корректности работы с памятью в куче, но накладные расходы увеличат время выполнения программы от 20 до 80 раз по сравнению с оригиналом.

- **Уровень mi4** – высший уровень, дополнен анализом ошибок доступа к памяти, выделенной на стеке, которые не обнаружены на стадии компиляции или с помощью run-time check опций компилятора. Уровень вложенности функций – 32. Как и все остальные уровни, 4-й является инклюзивным, то есть включающим в себя все виды анализа на предыдущих уровнях. Соответственно, накладные расходы будут максимальными.

В зависимости от выбранного уровня анализа, Parallel Inspector способен обнаружить следующие виды ошибок работы с памятью:
- **Memory Leak** – возникают при выделении программой памяти в куче и неосвобождении ее по окончании программы;
- **Invalid Memory Access** – возникают при чтении/записи по недействительным адресам памяти, в куче или в стеке;
- **Invalid Partial Memory Access** – возникают при чтении/записи по частично недействительным адресам памяти;
- **Missing Allocation** – возникают при попытке освободить память по несуществующему адресу;
- **Mismatched Allocation/Deallocation** – возникают при попытке освободить память с помошью функций, не соответствующих функции выделения памяти;
- **Uninitialized Memory Access** – возникают при попытке чтения неинициализированной памяти, в куче или в стеке;
- **Uninitialized Partial Memory Access** – возникают при попытке чтения частично неинициализированной памяти.

По окончании анализа приложения предоставляется лог событий или ошибок, которые были выявлены. Причем на всех уровнях, кроме mi1, можно начать анализировать логи еще до окончания анализа, так как результаты уже будут доступны.

Результаты структурированы таким образом, чтобы вначале был обзор списка проблем, от которых потом можно перейти к деталям или в окно анализа исходного кода. В списке с найденными ошибками доступна вся информация относительно процесса, модуля и функции, в которой эта ошибка произошла. При значительном списке ошибок, в случае достаточно большого проекта, удобно воспользоваться функционалом фильтрации по типу ошибки, по имени функции или модуля. Исходный код сопровождается стеком вызовов функций.

4.4.2. Ошибки многопоточности

Коварство ошибок многопоточности заключается в том, что при отладке или тестировании приложения они могут и не проявляться. Но стоит измениться некоторым временным параметрам, например, запустить приложение на системе с другой тактовой частотой процессора, как они тут же себя обнаруживают.

Для Inspecrtor не имеет никакого значения, произошла ошибка во время исполнения анализируемого приложения или нет. Он инструментирует все инструкции потоковых API, вызовов функций синхронизации и обращения к разделяемым между потоками объектам, на уровне бинарного кода. Затем производится анализ исполняемого пути приложения и выявление даже гипотетической возможности одновременного доступа потоками к незащищенным данным. Вот почему важно чтобы были пройдены все критические пути исполнения кода приложения, то есть тест должен обладать свойством полноты.

Тем не менее, технология бинарного инструментирования несет с собой значительные накладные расходы на выполнение приложения. Поэтому оно будет исполняться дольше, чем его оригинал. В связи с этим пользователю предоставлена возможность управления сложностью анализа. Для этого перед началом анализа он выбирает один из четырех уровней:

- **Уровень ti1** – позволяет обнаруживать взаимные блокировки потоков. Глубина стека функций равна единице;
- **Уровень ti2** – дополнительно позволяет обнаруживать конкуренцию доступа к незащищенным данным;
- **Уровень ti3** – отличается тем, что глубина стека увеличена до 12-ти;
- **Уровень ti4** – высший уровень, позволяющий определять все типы ошибок многопоточности. Уровень вложенности функций – 32. Накладные расходы будут максимальными.

В дополнение к ошибкам взаимной блокировки Inspector способен обнаруживать следующие ошибки многопоточности:

- **Lock Hierarchy Violation** – возникает при захвате нескольких объектов синхронизации, состоящих в иерархии или уже захваченных данным потоком;
- **Potential Privacy Infringement** – возникает при попытке доступа к стековой памяти другого потока.

После окончания исполнения приложения Inspector выведет список ошибок и диагностических сообщений о событиях, связанных с существованием потоков в процессе выполнения. Каждому сообщению Inspector сопоставит строку исходного кода, в которой найдена причина того или иного события или ошибки, а также стек вызовов функций и адрес памяти. Так как потоковые ошибки сопряжены с выполнением программы несколькими потоками одновременно, а иногда доступ к незащищенным данным может происходить из разных функций, то для удобства представления и обнаружения причин ошибки Inspector отображает сразу два окна исходного кода с функциями, выполнявшимися разными потоками. Из любого окна очень легко перейти в редактор Visual Studio, для редактирования программы. Для того, чтобы проанализировать приложение, не требуется никаких специальных ключей компиляции. Достаточно просто в режиме Debug запустить инструмент на исполнение.

4.5. Intel Parallel Amplifier

Профилировщик производительности предназначен для того, чтобы выяснить, насколько эффективно используется приложением многопроцессорная платформа, и где находятся те узкие места в программе, которые мешают ей масштабироваться и увеличивать производительность с ростом вычислительных ядер в системе. Методология профилировки приложения предельно проста: необходимо ответить себе на три основных вопроса, каждый из которых соответствует своему типу анализа и отражает суть, место и причины проблем с производительностью.

- **Hotspot-анализ.** «На что программа тратит вычислительное время процессора?» Необходимо знать те места в программе, Hotspot-функции, где больше всего тратится вычислительных ресурсов при исполнении, а также стек вызовов (тот путь, по которому мы в эти места попали).

- **Concurrency-анализ.** «Почему программа плохо параллелится?» Бывает так, что независимо от того, насколько продвинута параллельная структура приложения, ожидаемый прирост производительности при переходе, например, от 4-ядерной системе к 8-ядерной так и не достигается. Поэтому тут нужна оценка эффективности параллельного кода, которая дала бы представление о том, насколько полно используются ресурсы процессора.
- **Lock & Wait - анализ.** «Где программа простаивает в ожидании синхронизации или операции ввода-вывода?» Поняв, что программа плохо масштабируется, хочется найти, где и какие именно объекты синхронизации встали на пути к хорошей параллельности. Возможно, необходимо пересмотреть реализацию алгоритмов, а может, и всю параллельную структуру приложения.

Каждый из этих видов анализа запускается по отдельности и имеет собственное окно представления результатов. Наличие встроенного функционала сравнения результатов позволяет отслеживать влияние изменения кода программы на ее производительность.

4.5.1. Hotspot-анализ

Результатом запуска приложения на Hotspot-анализ будет окно со списком «горячих» функций, напротив каждой из которых будет представлена ее временная характеристика как в числовом, так и в графическом представлении. По умолчанию результаты отсортированы так, что самая «горячая» функция оказывается в списке первой (Рис. 4.5.1). Кроме того, имя каждой функции может быть «раскрыто» для представления всех стеков вызовов этой функции. А в окне Stack View можно определить, какой именно путь внес наибольший вклад (в смысле влияния на производительность).

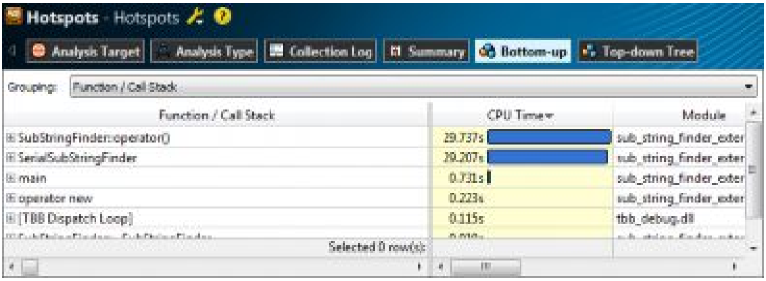

Рис. 4.5.1

Необходимо отметить, что поиск Hotspot-функций не несет в себе существенных накладных расходов. То есть анализ не влияет на время исполнения анализируемого приложения, что достигается путем использования технологии Stack Sampling (временное сэмплирование стеков), благодаря которой мы получаем трассу с данными, содержащими статистически значимые временные показатели функций, их стеки, а также контекст исполнения. В постпроцессинге трасса раскрывается в имя функции, потока, модуля и процесса, и выстраиваются список Hotspot-функций и статистическое дерево вызовов.

Естественно, что определив имя интересующей Hotspot-функции, хотелось бы проанализировать ее исходный код, поняв какие конструкции чрезмерно потребляют время процессора. Двойной клик мыши по имени функции открывает Source View закладку (Рис. 4.5.2), где затраченное функцией время на исполнение распределено по строкам исходного кода, а самое «горячее» место перемещено в центр окна – это очень удобно, особенно когда листинг функции занимает несколько экранов.

При переключении на закладку статистического дерева вызовов появится Top-Down представление для данной функции, которое помогает определить все пути ее вызовов и критический путь. Важными временными характеристиками функций здесь являются время, затраченное на выполнение самой функцией (Self-Time), и время, затраченное самой функцией и всеми функциями, вызванными из нее (Total-Time).

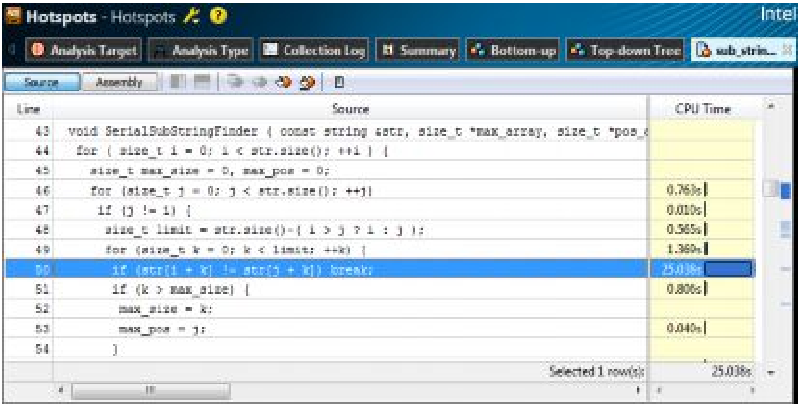

Рис. 4.5.2

Без такого представления, например, очень трудно определить, каким образом очень часто исполняемые функции (например, memcpy) влияют на производительность, если они попали в список самых «горячих» в Hotspot-листе.

4.5.2. Concurrency-анализ

В результатах Concurrency-анализа главной метрикой является интегральная характеристика параллельности всего приложения, которая дает представление о том, насколько хорошо распараллелено приложение в целом. Эта характеристика представлена гистограммой распределения времени исполнения приложения по уровням параллельности. Под уровнем параллельности понимают долю времени, в течение которого программа выполнялась в N ядрах процессора. Уровень N=1 означает, что программа выполнялась последовательно, N=2 – в двух потоках, и так далее. В идеальном случае график должен иметь пик на уровне N, равному числу вычислительных ядер в системе, и незначительные показатели на всех остальных уровнях. Другим важным свойством такого представления является то, что по графику можно оценить потенциал роста производительности программы, в случае разрешения проблем с масштабируемостью.

В главном окне появится список функций, при выполнении которых программа недоиспользовала возможности микропроцессора, и уровень параллельности был низкий. Чем больше времени выполнялась функция в неэффективном с точки зрения производительности режиме, тем выше она в списке. То есть это не список Hotspot-функций, хотя он и может с ним пересекаться. Здесь напротив каждой функции представлена графическая характеристика параллельности программы в виде «градусника», где зеленый цвет обозначает порцию времени исполнения в идеальном для данной системы режиме, а красным – время, в течение которого ресурсы системы недоиспользовались (Рис. 4.5.3). Есть еще синий цвет, который обозначает время исполнения программы в потоках, количество которых превышает число ядер процессора – переиспользование ресурсов. При незначительном превышении числа потоков идеального количества ничего страшного с производительностью обычно не происходит, так как влияние избыточного количества потоков в очереди системы начинает быть заметным со значительным ростом этого количества.

Рис. 4.5.3

Таким образом, в параллельной программе важно найти не только Hotspot-функции, но и понять, насколько хорошо они распараллелены и нужно ли над ними работать в плане более эффективной балансировки нагрузки или оптимизации объектов синхронизации. Если же Hotspot-функции распараллелены хорошо, то они не окажутся в списке с красной зоной «градусника», и тогда работа по увеличению производительности разделится на две задачи: оптимизация Hotspot-функций на архитектурном уровне и улучшение параллельности тех функций, которые есть в списке и недоиспользуют ресурсы процессора.

4.5.3. Locks & Waits - анализ

Профилировка приложения с помощью Locks & Waits - анализа помогает найти причину, почему не масштабируется приложение, что мешает ему выполняться с большим количеством потоков. В суммарной характеристике появляется метрика заблокированных потоков, то есть порция времени, когда потоки находились в заблокированном состоянии и не нагружали процессор (Рис. 4.5.4). Главное окно отображает список функций, исполнение которых было заблокировано в ожидании каких-либо объектов синхронизации. Чем больше влияние такой функции на время исполнения программы, тем выше она в списке.

Получив список «виновников» плохой масштабируемости, разработчик уже сам принимает решение, каким образом он изменит реализацию данного алгоритма или всю потоковую структуру в целом. Важно то, что он имеет объективную картину состояния с блокировками в программе и может оценить, сколько необходимо затратить усилий для переписывания приложения, и что это в результате даст.

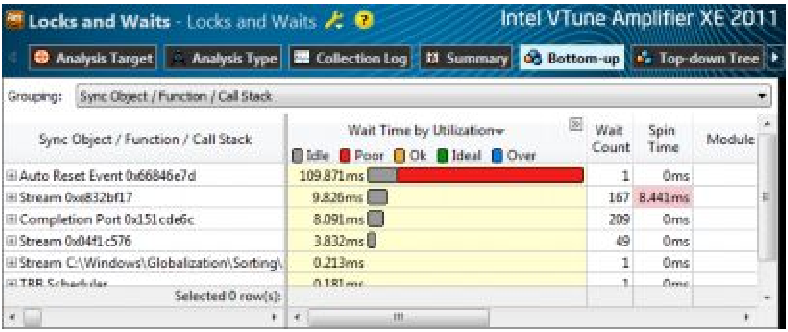

Рис. 4.5.4.

4.6. Выводы

В этой главе мы познакомились с основами процесса разработки приложений под Windows. Intel Parallel Studio – комплексное решение для разработки параллельных программ, которое облегчает старт новичкам и делает работу опытных разработчиков на C/C++ более продуктивной. Ведь разработка масштабируемых программ автоматически увеличивает производительность приложений на будущих многоядерных платформах.

Список литературы

1. Intel Parallel Studio home page
http://software.intel.com/en-us/forums/intel-parallel-studio

2. Intel Parallel Composer – расширение для отладки параллельных программ
http://www.ixbt.com/soft/intel-parallel-composer.shtml

3. Intel Parallel Amplifier – профилировщик многопоточных приложений
http://www.ixbt.com/soft/intel-parallel-studio.shtm