УДК 004.432.2

ПРИМЕНЕНИЕ ЯЗЫКОВЫХ СРЕДСТВ C# ДЛЯ НАУЧНЫХ ВЫЧИСЛЕНИЙ.

Рыжих Д.А.
БГУИР, Минск, Беларусь, ryzhykh.dmitry@gmail.com

Язык C# довольно успешно использовался в проектах многих типов, в том числе для разработки Web-приложений, баз данных, GUI и т. д. Одним из последних рубежей применения C# кода вполне могут стать научные вычисления. Но может ли C# сравниться с FORTRAN и C++ в научных и математических проектах? Ответим на этот вопрос, исследуя общеязыковую исполняющую среду (common language runtime, CLR) в .NET, чтобы определить, как JIT компилятор, промежуточный язык Microsoft (Microsoft intermediate language, MSIL) и сборщик мусора (garbage collector) влияют на производительность. Также рассмотрим типы данных C#, включая массивы и матрицы, и другие языковые средства, играющие важную роль в приложениях для научных вычислений.

Появление компьютеров позволило ученым легче доказывать теории, решать комплексные уравнения, моделировать трехмерные среды, прогнозировать погоду и выполнять многие другие задачи, требующие интенсивных вычислений. С годами были разработаны буквально сотни высокоуровневых языков, помогающих использовать компьютеры в данных областях (некоторые из этих языков были узко специализированными для параллельных вычислений, например, Ada и Occam, другие вроде Eiffel или Algol предлагали широкий набор средств для научных расчетов). Но лишь немногие из них стали выдающимися языками научного программирования, в том числе C, C++ и FORTRAN, каждый из которых сыграл важную роль в мире научных вычислений.

Одним из главных отличий среди языков, применяемых в научном программировании, была производительность. Компиляторы и генераторы кода часто считаются факторами, ограничивающими производительность, но это утверждение не вполне корректно. Например, наиболее распространенные компиляторы C++ хорошо справляются с генерацией и оптимизацией кода. Здесь есть некоторые тонкости, но обычно они не так важны, как эффективность кода. Скажем, в C++ следует избегать создания слишком большого количества временных объектов, тем более что в этом языке очень легко неосознанно создать массу безымянных временных объектов. Поэтому лучше использовать шаблоны выражений, позволяющие отложить реальные вычисления математического выражения до его присвоения. В результате вы избежите больших потерь из-за абстракций в период выполнения.

Нельзя сказать, что единственный фактор, влияющий на производительность, − особенности языка. При сравнении быстродействия языков на самом деле сравнивается мастерство создателей компиляторов, а не сами языки. Если вы можете получить приемлемое быстродействие от языка, исполняющей среды или платформы, то выбор становится делом вкуса.

JIT компиляция − значимая часть технологии, открывающей возможности для широкого спектра оптимизаций. Хотя текущая версия по временным рамкам ограничена в количестве выполняемых оптимизаций, теоретически она может превзойти все существующие статические компиляторы. Конечно, причина в том, что динамические свойства критичного к быстродействию кода или его контекст не полностью известны или проверены до периода выполнения. JIT компилятор способен использовать собранную информацию, генерируя более эффективный код, который теоретически может заново оптимизироваться при каждом запуске. Сейчас компилятор генерирует машинный код лишь раз для каждого метода. После генерации машинный код выполняется с той скоростью, на которую способен данный компьютер.

В научном программировании это может быть удобно. Код научных расчетов в основном выполняет операции над числами. Чтобы осуществлять такие вычисления за приемлемое время, следует корректно использовать некоторые аппаратные ресурсы. Хотя

многие статические компиляторы хорошо оптимизируют код, динамическая природа JIT компилятора позволяет ему оптимизировать использование ресурсов с помощью таких методов, как распределение регистров на основе приоритета (priority-based register allocation), «ленивая» выборка кода (lazy code selection), подстройка кэша (cache-tuning) и специфичные для процессора оптимизации (CPU-specific optimizations). Эти методы также открывают возможности для более тонкой оптимизации вроде разложения сложных команд (strength reduction), подстановки значений констант (constant propagation), избыточной загрузки после записи (redundant load-after-store), исключение общих подвыражений (common sub-expression elimination), исключение проверки границ массивов (array bounds check elimination), подстановки тел методов (method inlining) и т. д. Хотя для JIT компилятора такие продвинутые виды оптимизации возможны, в текущей версии .NET они не поддерживаются.

JIT компилятор, поставляемый с .NET Framework 1.1, значительно усовершенствован в сравнении со своим предшественником версии 1.0. График на рис. 1 иллюстрирует сравнение производительности между CLR версии 1.0 и 1.1 при выполнении комплекса эталонных тестов SciMark 2.0 на двух платформах. Для тестирования использовалась машина с процессором Pentium 4 с тактовой частотой 2.4 ГГц и 256 МБ оперативной памяти.

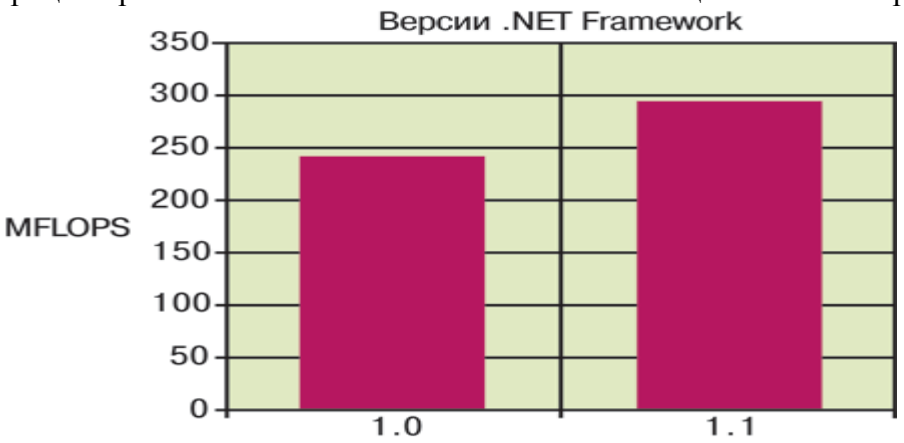

Рисунок 1 – сравнение производительности CLR

Думаю, что следующая версия JIT компилятора будет работать еще лучше, и способность JIT компиляторов генерировать более быстрый код, чем создаваемый статическими компиляторами, − лишь вопрос времени.

С точки зрения реализации, автоматическое управление памятью, наверное, − лучший подарок CLR разработчикам. Память выделяется относительно быстро (указатель кучи просто перемещается на следующий свободный слот) − по сравнению с более медленным и расточительным просмотром списка свободных страниц в вызовах malloc или new в C/C++. Более того, в период выполнения память управляется автоматически, освобождая и уплотняя незадействованное пространство. Программистам больше не надо гоняться за указателями, преждевременно освобождая блоки памяти или не освобождая их вовсе (хотя такие языки, как C# и Visual C++ по-прежнему дают такую возможность).

Реакция многих разработчиков на мысль об использовании сборщика мусора вполне предсказуема. Хотя сборщики мусора действительно приводят к некоторым издержкам в интенсивно работающих с памятью приложениях, они все же берут на себя все запутанные детали отслеживания утечек памяти и очистки «брошенных» указателей. Сборщики мусора постоянно управляют кучей, сжимают ее и обеспечивают повторное использование.

Недавние исследования и эксперименты показали, что в приложениях с интенсивными вычислениями, где объекты выделяются и освобождаются чаще, сбор мусора способен реально увеличить производительность за счет сжатия кучи. Кроме того, часто используемые объекты, которые иначе были бы случайным образом распределены в памяти, собираются вместе, что повышает локальность и эффективность работы кэша. Это намного увеличивает общую производительность приложения. В то же время один из недостатков сборщиков мусора — их непредсказуемость, из-за которой трудно проводить сбор мусора именно тогда, когда это нужно. В этой области ведутся исследования, и сборщики мусора постоянно

совершенствуются. Со временем появятся улучшенные алгоритмы с более предсказуемым поведением.

C# − объектно-ориентированный язык. Поскольку реальный мир состоит из тесно взаимосвязанных объектов с динамическими свойствами, объектно-ориентированное программирование (ООП) зачастую лучше подходит для решения задач, связанных с научными расчетами. Более того, структурированный объектно-ориентированный код легче модифицировать, заменяя внутренние части в соответствии с изменениями в научных моделях.

Однако не все научные проблемы можно выразить через объекты и их отношения — в таких случаях ориентация на объекты приводит к лишнему усложнению. Кроме того, между объектами возможно сложное взаимодействие, которое в программной реализации приведет к еще большему усложнению или нежелательным издержкам. Для примера возьмем молекулярную динамику. Молекулярная динамика широко используется в вычислительной химии, физике, биологии и материаловедении. Она была одной из первых областей научного применения компьютеров: в 1957 году Элдер (Alder) и Уэйнрайт (Wainwright) смоделировали движения примерно 150 атомов аргона. В молекулярной динамике ученых интересует моделирование взаимодействия атомов через парные потенциалы (pairwise potentials), аналогичного тому, как гравитация влияет на взаимодействие между солнцем, планетами, их спутниками и звездами. Моделирование взаимодействия между двумя атомами с использованием ООП может быть оправданным. Но представьте воображаемый куб, содержащий N^3 атомов, где N − очень большое число. В итоге уравнения для вычисления всех парных сил и энергий могут оказаться настолько сложными, что для расчетов придется отказаться от ориентации на объекты и отдать предпочтение традиционному процедурному подходу. Но и процедурный код может оказаться менее производительным. Все зависит от способа хранения данных и применяемых алгоритмов.

Все это можно сделать на C#. Вероятно, это не самое оптимальное применение языка с мощными объектно-ориентированными средствами, но хорошее начало для консервативного научного программиста. Единственный класс может содержать все переменные и методы для выполнения вычислений и выдачи результатов. Тем не менее, для многих задач научного программирования ООП может быть ценным инструментом, поскольку оно обеспечивает модульность и целостность данных, облегчающие распространение и повторное использование кода.

В научном коде нельзя игнорировать точность и корректность преобразований. Даже самые мощные современные компьютеры могут обеспечить точность лишь конечным числом разрядов. Значимость точности трудно переоценить — она подтверждается катастрофами, которые происходили из-за арифметических ошибок; вспомните, например, известный взрыв непилотируемой ракеты «Ариан 5» в 1996 году (64-битное число с плавающей точкой в системе инерциальной ориентации было неправильно преобразовано в 16-битное целое со знаком… и последовал взрыв!). Конечно, вы вряд ли разрабатываете программное обеспечение для ракет, но важно понимать, насколько и почему во многих научных приложениях столь важна высокая точность, а также почему стандарт IEEE 754 для двоичной арифметики с плавающей точкой может оказаться скорее сковывающим, чем полезным.

C# позволяет использовать в арифметических операциях с плавающей точкой более точные типы данных на аппаратных платформах с соответствующей поддержкой, например 80 битный формат Intel двойной точности. Этот тип имеет большую точность, чем double, и неявно задействуется нижележащим оборудованием при выполнении всех вычислений с плавающей точкой, давая абсолютно корректные или очень близкие к тому результаты.

C# также поддерживает тип decimal − 128-битный тип данных, пригодный для финансовых и валютных расчетов.

В C# существуют две основные категории типов: ссылочные (reference types) и типы значений, или значимые типы (value types).

Память под ссылочные типы всегда выделяется из кучи (единственное исключение − использование ключевого слова stackalloc); они создают дополнительный уровень абстракции, т. е. требуют доступа через ссылку на место их хранения. Поскольку к этим типам нельзя обращаться напрямую, переменная ссылочного типа всегда хранит ссылку на реальный объект (или null), а не сам объект. А так как память под них выделяется из кучи, исполняющая среда должна быть уверена в правильности каждого запроса на выделение.

Значимые типы хранятся непосредственно в стеке (хотя есть исключение, о котором я вскоре расскажу). Таким типам не требуется дополнительный уровень абстракции и поэтому переменные значимых типов всегда хранят само значение, а не ссылки на другие типы (и поэтому не могут содержать null). Главное преимущество значимых типов по сравнению со ссылочными в том, что создание их экземпляров не приводит к большим издержкам. Память под них выделяется в стеке простым приращением указателя стека, и они не управляются диспетчером памяти. Эти объекты никогда не инициируют сбор мусора. Более того, для значимых типов барьер записи не генерируется.

Примеры значимых типов в C# − элементарные типы данных (int, float, single, byte), перечислимые и структуры. Кстати, сказав ранее, что значимые типы хранятся прямо в стеке, я не употребил слово «всегда», как для ссылочных типов.

Загвоздка в том, что можно случайно внести значимые типы в объект и они окажутся в куче, − этот процесс известен как упаковка (boxing). Убедитесь, что значения в вашем коде не упаковываются без необходимости, иначе вы потеряете изначально достигнутое быстродействие. Учтите также, что массивы значимых типов (например массивы типа double или int) хранятся в куче, а не в стеке. В стеке хранится лишь значение, содержащее ссылку на такой массив. Причина в том, что все типы массивов неявно наследуют от System.Array и являются ссылочными типами.

Для научных приложений значимые типы быстрее, эффективнее и предпочтительнее ссылочных типов.

Многие разработчики тратили время и ресурсы на создание библиотек или решений на основе технологий и языков, предшествовавших .NET Framework. Хотя в переносе кода на C# много преимуществ, это не всегда приемлемо, если у вас уже есть протестированный код, на который можно опереться в дальнейшей работе. К счастью, благодаря спецификациям Common Language Specification (CLS) и Common Type System (CTS) в .NET встроена поддержка взаимодействия языков. Она позволяет существующему коду, написанному на любом CLS-совместимом языке, эффективно взаимодействовать с кодом на C#. В результате ваши любимые библиотеки для численных расчетов или код, написанный, скажем, на FORTRAN, можно интегрировать непосредственно в приложения на C#. Вы даже можете писать критичные к быстродействию части кода на неуправляемом C и использовать их в управляемом приложении на C#. .NET Framework обычно делает переходы между управляемым и неуправляемым кодом незаметными.

Это приведет к революции в научных разработках, поскольку теперь программисты могут одновременно писать код на нескольких языках, а затем легко интегрировать его или использовать совместно.

Таким образом, если у вас есть опыт программирования на C# и вы подумываете о выполнении серьезных научных расчетов, прибегать к другому языку не нужно − C# более чем достаточно.

Список литературы:

1. Microsoft developer network
   URL: https://msdn.microsoft.com/ru-ru/library/dd335957.aspx (12.11.2016)
2. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд. − СПб.: Питер, 2013. − 896 с.