ПРОГРАММИРОВАНИЕ

УДК 004.4'6

«ЭФФЕКТИВНОСТЬ» НИТЕЙ В МНОГОПРОЦЕССОРНЫХ СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

М.О. Бахтерев

Традиционно предполагается, что вычисление, разбитое на несколько нитей определённым образом, выполняется в системах с общей памятью (SMP или NUMA) быстрее, чем это же вычисление, но разбитое на несколько процессов. В представляемой работе высказана гипотеза о том, что такое предположение может быть неверным для вычислений с большими объёмами данных, главным образом по двум причинам. Во-первых, поддержка единого адресного пространства для нитей может быть существенно более накладной, чем суммарные затраты на переключение контекста выполнения между процессами. Во-вторых, даже если вычисление не требует интенсивного управления памятью, естественное ограничение на объём хранимого в TLB описания рабочего множества страниц, и в случае нитей приводит к необходимости частого обновления этого кэша трансляций. В статье описаны эксперименты и их результаты, которые подтверждают адекватность этой гипотезы.

Ключевые слова: общая память, производительность, нити, процессы.

Введение

При разработке некоторых компонент системного программного обеспечения для поддержки высокопроизводительных вычислений таких, как ядра ОС, языки программирования, среды распределённых вычислений, важно выбрать или выработать абстракции для представления задач и управления ими в многозадачном режиме работы. От особенностей этих абстракций зависят и удобство программирования, и уровень накладных расходов во время исполнения, а также некоторые важные свойства системы, например, устойчивость вычисления к сбоям.

Первым делом в поиске подходящего набора абстракций необходимо определить, какой тип задач — нити или процессы — должен быть для них базовым. И раз речь идёт о высокопроизводительных приложениях, то в выборе этого типа основным критерием должна быть бóльшая эффективность, то есть бóльшая скорость проведения вычисления в многозадачном режиме с задачами соответствующего типа.

Среди программистов распространено предубеждение, согласно которому проведение вычисления на многопроцессорной машине с общей памятью с разбиением на n ($n \in \mathbb{N}^{>0}$; где $\mathbb{N}^{>l} = \{m \in \mathbb{N} \mid m > l\}$ для $l \in \mathbb{N}$) нитей является более эффективным, чем с разбиением на n процессов. Обычно предложения произвести декомпозицию задачи на несколько процессов для параллельного расчёта на такой машине отвергаются с примерно таким комментарием: «дополнительные затраты на переключение контекста исполнения процессора с одного процесса на другой существенно снизят производительность».

При этом редко уточняется, о каких именно дополнительных накладных расходах идёт речь. А когда уточняется, то указывается на необходимость сбрасывать и загружать TLB

(Translation Lookaside Buffer; специальный кэш в структуре современных процессоров общего назначения, хранящий *трансляции* адресов, то есть, пары значений, описывающих соответствие виртуальных адресов физическим [1]). Общепринятая логика такова: так как в адресных пространствах различных процессов одинаковым виртуальным адресам могут соответствовать разные адреса физические, то после переключения процессора с выполнения процесса p_0 на выполнение процесса p_1 в TLB не должно остаться трансляций, загруженных в ходе выполнения p_0. Это мнение подкрепляется результатами хитроумных тестов, направленных на измерение скорости переключения контекста исполнения с нити на нить и с процесса на процесс [2, 3].

Однако, такая логика может вызвать сомнения, если обратить внимание на следующее. Для основных целевых процессоров SunOS 4.0 проблема с перезагрузкой TLB не стояла так остро, как для процессоров семейств IA-32 или AMD64. Приведённые рассуждения можно применить к последним, но внедрение современного варианта нитей с поддержкой на уровне ядра (kernel threads или LWP — Light Weight Processes) в широкую практику в 1998 году началось именно с операционной системы (ОС) SunOS 4.0 [4]. Для этой системы основной аппаратной платформой являлись рабочие станции Sun-4 с процессорами архитектуры SPARCv7 или SPARCv8 [5]. Техника трансляции виртуальных адресов в таких процессорах подразумевает использование идентификаторов адресных пространств (АП-идентификаторы), которые благодаря некоторому пересчёту виртуальных адресов позволяют одновременно хранить в TLB трансляции для нескольких различных процессов с гарантией от нежелательного пересечения их адресных пространств [6]. Более правдоподобной причиной, побудившей компанию SUN внедрить технологию LWP, можно считать другую архитектурную особенность этих процессоров — кэши с доступом по виртуальным адресам. С одной стороны, такие кэши позволяют эффективнее выполнять однозадачные вычисления, и это делало их популярным решением [7]. Но с другой, в них могут накапливаться данные, которые составляют контекст исполнения процесса (одному и тому же виртуальному адресу в разных процессах могут соответствовать разные данные). Для изменения этого контекста в определённых, достаточно вероятных обстоятельствах, может потребоваться долгая операция копирования большого объёма данных из кэша в память. Вероятность возникновения таких обстоятельств прямо зависит от количества различных адресных пространств в системе. А нити являются естественныс способом снизить количество таких пространств в многозадачном режиме работы.

Кроме этого, можно сказать и следующее. Без дополнительных специальных действий ОС общее адресное пространство для нитей формируется только при выполнении этих нитей на одном процессоре. При их исполнении в многопроцессорной системе необходимо создавать такое общее адресное пространство при помощи согласованного управления модулями доступа в память (MMU — Memory Management Unit [1]), в состав которых и входят TLB. Очевидно, что это требует некоторых затрат процессорного времени. Однако, найти работы, в которых производилась бы теоретическая или экспериментальная оценка влияния, оказываемого таким согласованным управлением на скорость проведения тех или иных вычислений, не удаётся. Не упоминается об этом и в классическом обзоре современно уровня развития технологий ОС Таненбаума [8].

Не удаётся отыскать и такие работы, в которых бы напрямую сравнивались эффективности декомпозиции некоторого вычисления на нити и на процессы. Исследователи обычно ограничиваются синтетическими тестами для измерения накладных расходов именно на переключение контекста, в которых обрабатывается незначительное по меркам высокопроизводительных приложений (а именно для них особо важны многозадачные режимы расчётов) объёмы данных.

Для получения более точных представлений об эффективности нитей как средства де-

композиции вычисления на параллельные задачи и была проделана работа, результаты которой составляют содержание этой статьи. В разделе 2 описываются возможные накладные расходы на организацию общего адресного пространства для выполнения нитей в многопроцессорной системе с общей памятью. И высказывается гипотеза о том, что они могут быть сравнимыми с расходами на переключение контекста исполнения с процесса на процесс. В разделе 3 рассказывается о результатах экспериментов, которые были поставлены для оценки потерь производительности из-за указанных расходов в некоторых видах вычислений: умножение матриц, управление памятью, обработка данных с интенсивными обменами.

1. Накладные расходы на общее адресное пространство

Безусловно, нити являлись бы таким инструментом, если бы речь шла об исполнении программы на однопроцессорной, или, как вернее говорить в наше время, одноядерной системе[1]. В этом случае любые изменения, вносимые в адресное пространство многонитевого процесса P одной из его нитей t_P сразу же распространяются и на другие нити P. Ведь, к моменту передачи управления на одну из них все необходимые изменения уже внесены и в управляющие адресным пространством P структуры, и в TLB. Иначе обстоят дела в случае исполнения P на наборе ядер C_P в многоядерной системе, а тем более в многопроцессорной, современные варианты которых являются системами NUMA, с неоднородным доступом в память. И утвердительный ответ на вопрос: «являются ли нити средством оптимизации под работу с TLB?» — теряет свою очевидность.

Дело в том, что семантика исполнения нитей в общем адресном пространстве требует от операционной системы поддерживать это общее адресное пространство для всех ядер в C_P. А это означает, что каждое изменение адресного пространства, запрашиваемое t_P, выполняемой на некотором ядре из C_P, должно происходить с согласованием буферов трансляции всех ядер в C_P, даже если это изменение неявное и связанное, например, с выделением новой страницы под стек t при его разрастании во время вызова некоторой подпрограммы. И это согласование нужно производить с обменом информацией по относительно медленным внешним для системы «ядро + TLB + кэш L1» шинам данных.

Наиболее сложной при этом является операция удаления страниц из адресного пространства. Выполняя её, ОС должна гарантировать, что TLB каждого ядра в C_P после завершения этой операции не будет содержать трансляций с адресами удалённых страниц. Обеспечение же этих гарантий требует очистки буферов трансляции адресов на всех ядрах из множества C_P. Можно различными способами оптимизировать эту процедуру, но в любом случае она потребует организации дополнительных критических секций для совместного доступа с различных ядер к некоторым управляющим данным, а также приостановки или даже прерывания исполнения нитей процесса P.

Прерывания же и выполнение дополнительных циклов синхронизации при входе в критические секции могут значительно увеличить время выполнения операций с адресным пространством. И чем больше ядер входит в C_P, тем подобные расходы процессорного времени существеннее. Синхронизация может оказаться особенно затратной по времени в системах NUMA. В последних необходимость поддерживать единое адресное пространство для всех нитей влечёт ещё один тип издержек.

Он связан с тем, что операционной системе необходимо размещать в памяти данные, служащие источником для заполнения буферов трансляции значениями. В данном случае не важно, автоматическое это заполнение или нет. При этом, можно хранить в некотором

[1]При существующем сейчас архитектурном разнообразии, не очевидно, какую часть процессора следует называть ядром. Для определённости в этой статье ядром называется та часть процессора, которая для исполнения программы использует определённый TLB.

модуле памяти одну, общую для всех ядер в C_P, копию таких данных. Но тогда увеличивается время доступа к этим структурам с ядер, находящихся в процессорных модулях далёких по расстоянию в графе связей NUMA системы от модуля памяти, содержащего эту копию. Соответственно, уменьшается и скорость доступа выполняющихся на этих «далёких» ядрах нитей к данным, требующим загрузки новых трансляций в TLB. Если же хранить некоторые полные или частичные копии этих данных о трансляциях в нескольких различных модулях памяти, то придётся тратить на поддержание их в согласованном состоянии некоторое процессорное время. И это согласование так же требует выполнения коллективных операций с синхронизацией и прерыванием исполнения нитей.

В противоположность этому, управление адресным пространством процесса с одним потоком управления не требует коллективного управления TLB многих ядер. В NUMA-системе необходимо поддерживать только одну копию данных, описывающих адресное пространство такого процесса, а решение о том, в каком модуле памяти их следует размещать, обычно, очевидно. Синхронизации и коллективных взаимодействий ядер могут потребовать операции управления сегментами общей для взаимодействующих процессов общей памяти, но эти операции обычно составляют небольшую часть от общего числа операций, меняющих адресное пространство.

Итак, можно провести декомпозицию некоторого вычисления на процессы или на нити. В первом варианте с процессами возникнут накладные расходы на перезагрузку TLB новыми значениями при переключении контекста исполнения процессора с одного процесса на другой. Во втором – накладные расходы будут связаны с необходимостью поддерживать общее адресное пространство для нитей через коллективное управление TLB множества ядер, на которых исполняются нити вычисления. Какой из этих двух типов накладных расходов оказывает большее негативное влияние на скорость проведения вычисления, априори не ясно. Дополнительную неясность в вопрос вносит и то, что работа с относительно большим объёмом данных, не помещающихся в то небольшое количество страниц, трансляции для которых можно одновременно хранить в TLB современных процессоров, как при выполнении нитей, так и при выполнении процессов, требует достаточно частого обновления записей в буферах трансляции.

Не понятно заранее и то, какую математическую модель можно предложить для оценки тех и других издержек. Следовательно, необходимо оценить их хотя бы экспериментально.

2. Эксперименты

Итак, необходимо оценить, как влияют на скорость расчёта два типа накладных расходов: на поддержку общего адресного пространства для нитей в многопроцессорной системе и на смену контекстов исполнения процессоров в такой системе с одного процесса на другой. Сделать это можно прямым методом. То есть, выполнить некоторое вычисление двумя способами: с разбиением его как на несколько нитей, так и на несколько процессов. Зависимость скоростей исполнения этих вариантов расчёта от количества участвующих в них задач позволит получить некоторые интегральные оценки влияния обсуждаемых издержек на производительность.

При этом, само вычисление должно обладать несколькими свойствами, помимо возможности провести его при помощи большого количества задач — нитей или процессов. Во-первых, время работы программ должно быть достаточным, чтобы каждый участвующий в этой работе процессор большое число раз переключился с выполнения одной задачи на другую. Это необходимо, чтобы потраченное на каждое из таких переключений время, сложилось в заметную величину. Во-вторых, вычисление должно интенсивно обращаться к большому объёму памяти, достаточному для создания высокой нагрузки на TLB. Счёт

должен идти на мегабайты, а не на килобайты, как в традиционных измерениях скорости переключения контекста исполнения [2, 3]. В-третьих, желательно, чтобы варианты кода с нитями и с процессами были максимально похожими.

В качестве подходящих были выбраны следующие вычислительные нагрузки. **1.** Управление памятью — выполнение большого количества операций резервирования (allocate) и освобождения (free) памяти. **2.** Умножение матриц, хранящихся в памяти по строкам. **3.** Умножение матриц, хранящихся по «плиткам» (tiled representation). **4** Обмены — вычисление, в ходе которого задачи обрабатывают массивы чисел и интенсивно обмениваются результатами обработки.

Дерево исходных текстов [11] содержит коды реализаций соответствующих вычислений для ОС на базе GNU/Linux. Выбор такой платформы для экспериментов сделан по двум причинам. Во-первых, под управлением Linux работают суперкомпьютеры ИММ УрО РАН, и выяснение особенностей поведения многозадачных приложений в этой среде важно для практики. И, во-вторых, исходные тексты Linux позволяют убедиться, что при переключении контекста ядра процессора с исполнения нити некоторого процесса P на исполнение нити того же процесса P сброс TLB не производится (например, [10]). Не во всех ОС ядра запрограммированы именно так.

Результаты измерений, проведённых на различных системах с разными настройками аффинности задач записаны в директорию `results` дерева исходных текстов. В статье использованы не все собранные данные, а только те, что представляют классы характерного поведения тестовых программ. Таблица 1 (T — число логических процессоров в системе; C — количество ядер; P — число физических процессоров) описывает системы, на которых получены приводимые в тексте данные.

Установки аффинности незначительно влияли на скорость вычислений, но большие скорости достигались при «чередовании»: задача с номером n назначается на один из N логических процессоров с номером $n \bmod N$. Linux по-умолчанию назначает номера логическим процессорам так, что «процессоры» с номерами t_1 и t_2 разделают ресурсы одного из C ядер системы, только если $t_1 \equiv t_2 \bmod C$. В тексте приведены времена исполнения тестовых нагрузок именно с такой схемой аффинности.

Таблица 1

Некоторые платформы, использованные в экспериментах, и их обозначения

Обозначение	CPU	T/C/P	Linux	GCC	libc
A640	Athlon II X4 640	4/4/1	3.3.4	4.7.0	2.15
X5675	Xeon X5675	24/12/2	2.6.32	4.4.6	2.12
I7	i7-2006K	8/4/1	3.3.5	4.7.0	2.15

Времена выполнения тестов измерялись утилитой `time`. Статистическая обработка измерений не дала дополнительных сведений об эффектах исследуемых накладных расходов, поэтому в тексте указаны «сырые» значения — просто время работы теста при одном из запусков с некоторыми параметрами. Это не искажает полученной качественной картины и позволяет всем желающим воспроизвести результаты, используя коды [11], за приемлемое время.

2.1. Управление памятью

Эксперименты с управлением памятью нацелены на определение накладных расходов, связанных с управлением общим адресным пространством.

Тестовые программы этого эксперимента в каждом раунде запускают N задач (нитей или процессов) для выполнения в сумме примерно K итераций: на задачи приходится по

$\lfloor K/N \rfloor$ итераций. На каждой итерации совершается одна из псевдослучайно выбранных операций. Задача инициирует своим логическим номером в общем вычислении собственный локальный генератор псевдослучайных чисел. Поэтому задача с номером k, как в варианте с нитями, так и в варианте с процессами, выполняет одну и ту же последовательность действий.

Возможные действия таковы. **1.** С вероятностью примерно 3/8 резервируется блок памяти, размер которого псевдослучайно выбирается из множества $\{2^i \cdot 64 \mid i = \overline{1,7}\}$. После резервирования блок размещается в структуре данных «кольцо», которая закреплена за задачей. **2.** С приблизительно такой же вероятностью выполняется операция прохода по кольцу на длину (измеряемую в звеньях кольца), которая так же псевдослучайно выбирается из множества $D = \{2^i \mid i = \overline{1,7}\}$. В каждое просмотренное при выполнении прохода звено кольца записывается некоторое значение. Так делается, чтобы с некоторой степенью достоверности симулировать работу программ, работающих с динамическими структурами данных. **3.** Наконец, с вероятностью около 2/8 выполняется операция освобождения памяти, занимаемой звеном «кольца», находящимся на псевдослучайном расстоянии (из D) от условного «начала» кольца. Применяемые в этом эксперименте многонитевой и многопроцессный варианты программы (директория `alloc` [11]) отличаются лишь кодом запуска и ожидания завершения задач.

Таблица 2

Результаты измерений тестом на управление памятью

N	1	2	4	8	16	32	64	128	256	512
T	42,157	19,804	10,865	7,455	8,025	13,540	14,529	13,429	11,773	11,768
P	40,521	18,220	8,931	4,474	2,504	1,838	1,112	.680	.449	.364

Несколько неожиданные результаты эксперимента с такой вычислительной нагрузкой приведены в таблице 2 (система X5675 (на прочих системах картина аналогичная); число итераций $K = 2^{24}$; T (P) — результаты выполнения K итераций теста N нитями (процессами)). То, что 512 процессов со всеми переключениями контекста могут выполняться на 12-ядерном процессоре быстрее 8 нитей более чем в 5 раз, выполняя примерно «такой же» объём работы, не соответствует общепринятым убеждениям.

Апостериори понятно, что меньшая скорость исполнения нитей в серии этих тестов лишь отчасти обусловлена активностью ОС, связанной с поддержкой единого адресного пространства. Основные же источники накладных расходов здесь — это алгоритмическая сложность освобождения некоторого блока адресного пространства и поиска в нём свободного участка, а также необходимость согласования нитей при выполнении процедур `malloc` и `free`. На это указывают следующие факты. **1.** Монотонное уменьшение времени работы многопроцессного варианта программ с ростом числа процессов, то есть, с уменьшением $\lfloor K/N \rfloor$ и, как следствие, числа операций с адресным пространством одного процесса. **2.** Значительное увеличение времени работы многонитевого варианта, когда количество нитей N превышает количество логических процессоров T в системе. Такое увеличение можно объяснить ростом времени доступа к описывающим адресное пространство структурам. Циклы синхронизации нитей для доступа к ним могут удлиниться из-за попадающих на эти циклы переключений контекста исполнения при $N > T$. **3.** Постепенное уменьшение и стабилизация времени исполнения многонитевого варианта программы при большом количестве нитей свидетельствуют как раз о временной сложности операций управления общим большим адресным пространством, высокой для каждой нити.

Процессы, как средство разделения адресных пространств на меньшие, как показала серия данных тестов, позволяют эффективно управлять этой сложностью.

2.2. Умножение матриц, хранящихся по строкам

Изначально предполагалось, что эксперименты с умножением матриц позволят сравнить накладные расходы на переключение контекста исполнения с нити на нить и с процесса на процесс. Наивное умножение матриц в соответствии с формулой $R^i{}_j = \sum_{k=1}^{m} A^i{}_k B^k{}_j$ не требует сложного управления адресными пространствами, однако требует множества обращений к находящимся в различных страницах памяти ячейкам, чем создаёт высокую нагрузку на TLB. При этом вычисление легко разбивается на участки, которые можно выполнять независимо как несколькими нитями, так и несколькими процессами, используя один и тот же код. В предлагаемых тестах каждая задача рассчитывала блок из строк матрицы $R^i{}_j$.

Таблица 3

Времена выполнения тестов с умножением матриц вида `double[K][K]`

A640										
N	1	2	4	8	16	32	64	128	256	512
T-M	295,464	153,986	84,631	84,389	84,390	85,028	85,224	86,667	85,587	85,544
P	294,966	153,913	84,668	84,461	84,623	85,000	85,339	85,760	86,139	86,785
T	292,140	149,852	82,307	82,184	81,578	80,746	81,390	81,543	80,994	81,487
HP.P	292,120	150,423	82,187	82,175	82,185	82,218	82,306	82,401	82,491	82,550
HP.T-M	291,978	150,453	82,277	82,201	82,233	82,266	82,330	82,438	82,590	82,885
X5675										
N	1	2	4	8	16	32	64	128	256	512
T-M	87,240	56,165	25,152	9,435	8,631	7,222	7,792	8,086	8,897	11,875
P	87,135	54,011	18,366	9,405	9,555	8,867	9,508	9,517	9,831	8,221
T	88,701	51,218	32,850	17,752	17,294	20,918	22,961	22,747	26,625	26,461
HP.P	104,325	74,665	44,272	21,033	19,118	19,726	20,916	19,863	22,270	19,674
HP.T-M	105,438	53,715	46,181	21,347	20,694	21,843	21,412	22,591	22,899	25,167
T-N	86,009	52,536	18,458	9,190	8,748	7,189	7,895	7,875	8,762	10,591
P-N	87,982	48,420	18,467	9,598	9,512	8,323	8,892	8,353	8,055	7,499
I7										
N	1	2	4	8	16	32	64	128	256	512
T-M	83,155	36,494	18,266	19,613	27,531	25,885	25,793	32,698	38,980	39,379
P	83,144	36,414	18,321	26,447	32,891	32,817	32,710	34,065	32,168	31,457
T	78,155	57,541	46,084	38,952	69,976	69,216	65,063	70,211	69,196	68,534
HP.P	81,336	57,728	44,532	22,737	44,757	44,436	39,475	40,880	43,288	43,011
HP.T-M	81,321	57,630	43,829	22,926	45,270	43,983	39,349	41,594	44,474	44,548

Случай, когда матрицы хранятся по строкам, то есть, когда элемент матрицы $A^i{}_k$ находится в ячейке памяти, задаваемой выражением `A[i][k]` (в семантике языка Си), интересен следующим. Если строка матрицы $B^k{}_j$ превосходит своим объёмом одну страницу памяти, то вычисление скалярного произведения $\sum_{k=1}^{m} A^i{}_k B^k{}_j$ требует для каждого k своей трансляции в TLB. Эти трансляции необходимо будет загружать на каждой итерации цикла вычисления произведения, если число строк матрицы $B^k{}_j$ превосходит вместительность TLB. Можно ожидать, что умножение на «широкую» и «высокую» матрицу $B^k{}_j$, хранящуюся по строкам, будет одинаково неэффективным в случаях выполнения этого умножения как несколькими нитями, так и несколькими процессами. Поэтому результаты этого эксперимента должны были послужить базовой для анализа тестов с более оптимальным использованием TLB, когда матрицы хранятся плиткой. Приведённые в таблице 3 ($\mathtt{K} = 2^{11}$; T (P) — умножение N нитями (процессами); HP отмечает варианты, когда под матрицы отводится адресное пространство, отображаемое большими (2MiB) страницами; M — общая

память для нитей резервируется вызовом `mmap`; N — в ядре ОС отключён механизм прозрачных больших страниц; время работы варианта программы p с процессами выделено, если оно меньше времени работы многонитевой программы с такими же параметрами, что и p, и со сходной p политикой использования страниц) времена исполнения тестов рисуют неожиданно сложную картину.

Отчасти это может быть связано с тем, что современные версии Linux (в том числе и 2.6.32, используемая на X5675) реализуют механизм ПБС — прозрачных больших страниц (transparent huge tlb), который старается зарезервированные при помощи `malloc` участки адресного пространства отображать в физическую память большими страницами, если их поддерживает процессор. Делается это в предположении о снижении нагрузки на TLB: чем больше страницы, тем реже необходимо загружать новые записи в TLB. Так, большая страница систем A640, I7 и X5675 размером 2MiB вмещает 128 строк матриц, использованных для получения данных в таблице 3. То есть, при использовании больших страниц в цикле подсчёта $\sum_{k=1}^{n} A^i{}_k B^k{}_j$ для этих матриц из $2^{11} \times 2^{11}$ значений `double` потребуется, на $.99n$ (как минимум) меньше загрузок в TLB новых трансляций, чем при использовании небольших страниц (4KiB для рассматриваемых систем).

«Естественная» реализация многонитевой программы память под матрицы резервирует, конечно, процедурой `malloc`. В многопроцессном же варианте общую память для записи матриц необходимо формировать вызовом `mmap` с такими «естественными» в этом случае параметрами, которые выводят сформированную область памяти из-под действия механизма ПБС, и ядро отображает её в физическую память страницами небольшими.

Вышесказанное объясняет существенную разницу во временах работы многонитевого и многопроцессного вариантов вычисления с «естественным» управлением памятью (строки T и P в таблице 3), которая вопреки общепринятому мнению не всегда говорит в пользу нитей. Поэтому в сравнительном анализе эффективности использования TLB нитями и процессами на эти времена полагаться нельзя. Linux позволяет как отключать поддержку ПБС, так и явно параметрами `mmap`, запрашивать отображение адресного пространства в физическую память большими страницами. Благодаря этому, можно организовать вычисления с декомпозицей на различные виды задач, но с одинаковой политикой использования больших и небольших страниц. Результаты экспериментов, проведённые с такими одинаковыми политиками для нитей и процессов, указаны в парах строк (T-M; P), (T-N; P-N) и (HP.T-M; HP.P) таблицы 3.

Если анализировать только одну из пар (T-M; P) или (T-N; P-N), то можно сказать, что результаты в некоторой степени предсказуемы. Пока количество задач не превышает количества ядер (ядер, а не логических процессоров), в большинстве из представленных случаев времена выполнения обоих вариантов вычислений примерно одинаковы. С ростом числа задач и возникновением вследствие этого конкуренции за TLB вычисление с нитями в большинстве представленных случаев выполняется за меньшее время. Это можно объяснить накладными расходами на переключение контекста TLB при исполнении многопроцессного варианта вычисления.

Однако, если смотреть на картину в целом, возникает множество вопросов, не позволяющих однозначно определить более эффективный метод декомпозиции вычислений на задачи. Вот некоторые из них. Почему в системах с процессорами Intel при увеличении числа задач многопроцессные варианты вычисления исполняются существенно быстрее, чем многонитевые? Почему в этих системах отображение памяти большими страницами существенно снижает скорость вычисления произведения матриц, сохранённых по строкам? Почему в них же механизм ПБС замедляет выполнение этого вычисления почти в 2 раза? Почему при малом числе задач в системе X5675 вычисление с процессами выполняется существенно быстрее?

Поиск ответов на эти вопросы требует более детального исследования особенностей умножения хранящихся по строкам матриц на процессорах с конкретной микроархитектурой, которое выходит за рамки этой статьи. Здесь же необходимо отметить важные для практики выводы, расходящиеся с общепринятыми представлениями. **1.** Реализация параллельного алгоритма «естественным» многонитевым способом может быть существенно менее эффективной по сравнению с «естественной» многопроцессной реализацией. **2.** Использование больших страниц не приводит автоматически к повышению скорости исполнения кода и даже может существенно её снизить. **3.** Кроме этого, ожидания того, что из-за общей неэффективности использования TLB этот вид нагрузки будет нечувствителен к типу задач, используемого для декомпозиции вычисления, не оправдались.

2.3. Умножение матриц, хранящихся по плиткам

Плиточное представление матрицы в памяти означает, что элемент ${A^i}_k$ записан по адресу `A[i/tr][k/tc][i%tr][k%tc]`, где `tr` и `tc` — число строк и столбцов в плитке, соответственно. Подобное представление позволяет более эффективно использовать TLB в вычислении произведения $\sum_{k=1}^{m} {A^i}_k {B^k}_j$. При любых (согласованных, конечно, для произведения) размерах матриц ${A^i}_k$ и ${B^k}_j$ увеличение k на единицу в цикле подсчёта этого произведения, если оно не выводит за пределы текущей плитки, не требует загрузки новых трансляций в TLB. Трансляция для матрицы ${A^i}_k$ может быть использована в `tc` итерациях этого цикла, а для матрицы ${B^k}_j$ — в `tr`. Бóльшая эффективность использования TLB в таком алгоритме позволяет априори ожидать, что многонитевой вариант вычисления будет выполняться быстрее.

Таблица 4

Времена выполнения тестов с умножением матриц вида `double[K/32][K/16][32][16]`

A640										
N	1	2	4	8	16	32	64	128	256	512
T-M	120,204	64,166	38,648	37,703	39,657	38,972	39,217	40,147	39,959	39,981
P	120,712	65,771	38,631	39,266	37,940	38,757	40,094	38,663	40,472	40,732
T	107,344	64,338	41,670	41,134	40,842	41,075	41,285	41,568	41,505	41,749
HP.P	108,030	64,724	41,995	41,632	41,739	42,046	42,230	42,587	42,851	43,110
HP.T-M	108,040	65,158	41,745	41,719	41,724	42,113	42,096	42,337	42,473	42,905
X5675										
N	1	2	4	8	16	32	64	128	256	512
T-M	65,810	23,925	12,285	20,067	5,909	5,702	5,894	5,845	5,860	6,120
P	70,085	23,605	12,284	19,819	5,954	5,640	5,907	5,932	5,892	6,275
T	93,167	60,820	35,291	34,178	21,887	20,808	26,122	28,632	26,900	26,614
HP.P	94,549	62,508	37,761	26,099	23,757	21,066	22,377	22,733	23,419	21,367
HP.T-M	94,303	62,388	39,985	24,918	24,490	21,417	24,890	25,407	26,767	24,918
T-N	80,874	24,286	12,586	7,762	6,057	5,781	6,051	5,997	5,877	5,878
P-N	76,925	23,415	12,323	7,559	5,941	5,712	5,888	5,708	5,564	5,443
I7										
N	1	2	4	8	16	32	64	128	256	512
T-M	87,770	17,638	8,931	9,100	9,138	9,227	9,374	9,535	9,500	10,481
P	88,129	17,636	8,930	9,025	9,181	9,289	9,455	9,343	9,358	9,445
T	98,246	52,992	55,307	99,366	93,864	92,852	93,817	93,634	93,108	94,258
HP.P	80,502	62,358	53,411	60,839	61,917	61,567	61,278	60,676	59,840	58,728
HP.T-M	80,510	63,525	58,200	61,928	61,875	61,488	61,346	61,100	60,721	60,315

Для проведения тестов с умножением матриц, хранящихся по плиткам, использовались те же программы, что и для тестов в предыдущем разделе, с единственным отличием в коде вычисления адреса элемента матрицы.

Времена исполнения этих программ с различными параметрами приведены в таблице 4 (обозначения и параметр K такие же, что и в таблице 3). Полученные результаты расходятся с ожиданиями. Во-первых, потому что теперь во всех случаях использование больших страниц снижает производительность, в том числе и на показавшей их эффективность в предыдущей вычислительной нагрузке системе A640. Во-вторых, потому что многопроцессные программы выполняются быстрее многонитевых во многих случаях, в том числе и при малом числе задач. Из полученных данных нельзя сделать вывод о существенном негативном влиянии смены сохранённого в TLB контекста исполнения при переключении именно процессов (контекст меняется и при переключении нитей, как сказано в конце раздела 1). на скорость проведения вычисления.

Для выяснения причин подобного поведения нужны дальнейшие детальные исследования, которые выходят за рамки этой статьи.

2.4. Обмены

Даже если подойти к представленным в 2.2. и 2.3. результатам наивно и подсчитать для каждого вида задач случаи, когда он обеспечивает меньшее время расчёта при прочих одинаковых параметрах теста, то выйдет, что в 58,56% случаев процессы являются более выгодным решением. Такое соотношение не позволяет с уверенностью сделать вывод о том, на какие задачи следует производить декомпозицию вычислений. Поэтому было решено провести ещё одну серию тестов, в которых бы сочетались арифметическая нагрузка с интенсивными доступами в память, нагрузка управления адресными пространствами и нагрузка, связанная с обменами между задачами блоками данных не определённого заранее размера (можно считать, что случай, когда общие для вычисления данные имеют фиксированный размер, разобран в двух предыдущих параграфах). В рамках интерфейса POSIX процессы могут совершать подобный обмен, либо передавая сообщения, либо копируя данные во внешние по отношению к своим адресным пространствам объекты и из них. Роль этих объектов могут играть файлы специально созданной для этого `tmpfs` [12]. Как при передаче сообщений, так и при использовании внешних объекты Linux может осуществлять копирование при записи (copy on write). В подобных условиях можно ожидать, что в частых на практике ситуациях, когда такие обмены необходимы, несмотря на накладные расходы управления общим адресным пространством и «невнятное» влияние на производительность смены контекста TLB, нити с их возможностью доступа к произвольным полученным в расчёте данным окажутся более эффективным средством декомпозиции вычисления на параллельные задачи.

Тесты, представленные в этом разделе, по своей псевдослучайной структуре похожи на тесты параграфа 2.1. Отличие заключается в трёх процедурах, выполняемых на разных итерациях с некоторыми вероятностями. Все эти процедуры работают с массивом чисел типа `double`. **1.** С вероятностью примерно 3/8 выполняется процедура увеличения. Массив увеличивается с конца на псевдослучайное число элементов, до $W = 2^{10}$ штук, и заполняется псевдослучайными значениями. **2.** С той же примерно вероятностью выполняется процедура сокращения массива. Суммируется псевдослучайное число элементов, до W штук, расположенных в начале массива. Суммирование производится алгоритмом подсчёта суммы с максимальной точностью, то есть, из накопленной суммы и оставшихся элементов выбираются два наименьших значения и складываются в следующую накопленную сумму. Полученная сумма записывается в конец массива, вместимость которого для этого увеличивается на 1 элемент. **3.** С вероятностью примерно 2/8 выполняется процедура обмена рабочими массивами. Выполняющая эту операцию задача t отправляет по стандартному в POSIX каналу `pipe` соседней, условно, «справа» задаче номер массива значений, который t обрабатывала до обмена, и ожидает от задачи «слева» соответственно, номер массива, который t должна

обрабатывать после. Каналы объединяют задачи в ориентированное «слева направо» кольцо. Выполнившая все свои итерации задача посылает «правой» соседке специальное значение, которое передаётся дальше по кольцу и служит сигналом к общему прекращению работ.

В отличие от пар программ, использованных в описанных ранее экспериментах, программы для этого, существенно разнятся по своему коду. Многонитевой и многопроцессный варианты кода, используемые здесь, отличаются существенно.

Многонитевая реализация на C++ использует для представления и операций с массивами чисел обобщённый тип `vector` из библиотеки STL. Операции, описанные выше, операции увеличения и сокращения массива естественным образом укладываются в набор операций для этого типа. Считается, что `vector` реализован эффективно, и эту структуру часто рекомендуют, как наилучший вариант для представления динамических массивов. Для выполнения операции обмена нитям необходимо всего лишь передавать индексы сохранённых в общем статическом массиве структур, описывающих состояние объектов типа `vector`.

Многопроцессная реализация размещает массив чисел в области памяти, управляемой системными вызовами `mremap` и `munmap` при увеличении и сокращении массива соответственно. Для осуществления обмена процессы так же обмениваются номерами, но каждый из них соответствует размещённому в `tmpfs` файлу. Перед передачей массива данных дальше по кольцу процесс записывает накопленный массив в файл, номер которого он передаст «правому» соседу, системным вызовом `pwrite`. Получив номер файла, содержащего массив для дальнейшей обработки, от «левого» соседа, процесс размещает данные из этого файла в своём адресном пространстве при помощи `pread`. Используется именно такой метод обмена, а не пересылка сообщений, чтобы схема взаимодействия процессов была похожа на схему взаимодействия нитей, общие данные которых существуют не только между парами задач.

Таблица 5

Результаты измерений тестом на обмены

N	1	2	4	8	16	32	64	128	256	512
T	80,621	62,323	37,922	20,562	12,790	8,963	7,179	6,388	6,924	FAIL
P	90,608	65,855	39,451	22,577	13,821	10,316	7,397	6,317	5,637	5,126

Результаты измерений вышеописанными тестами приведены в таблице 5 (система X5675 (на прочих системах картина аналогичная); T (P) — результаты выполнения $K = 2^{18}$ итераций теста N нитями (процессами); FAIL означает, что программа не запустилась из-за ограничений на использование ресурсов). И снова, вопреки ожиданиям, результаты не позволяют сделать однозначный вывод об эффективности того или иного вида задач. Видно, что доля накладных расходов на передачу данных между адресными пространствами велика. С увеличением числа задач время работы уменьшается, так как уменьшается число итераций, выполняемых первой из завершающихся задач. Доля издержек на поддержание общего адресного пространства приближается при этом к доле издержек на смену контекста TLB и передачу данных между адресными пространствами. И даже в некоторый момент – 128 задач для указанных K и W – накладные расходы в многопроцессном варианте программы становятся *ниже*, чем расходы в многонитевом. Конечно, при создании высокопроизводительных приложений не рекомендуется запускать на системе с общей задач больше, чем число доступных в ней процессоров. Но системы с общей памятью (и даже чипы), содержащие 128 и более логических процессоров существуют [13].

3. Заключение

Прежде всего стоит отметить «побочный» результат проведённых исследований: отображение виртуального адресного пространства на физическую память большими страницами при выполнении некоторых вычислительных нагрузок на некоторых процессорах под управлением Linux может снижать производительность. Такой эффект можно наблюдать не только на тестах, описанных в разделе 2.Так на системе X5675 OpenMP вариант программы `lu.D` из пакета NPB-3.3.1 при включении механизма ПБС теряет в скорости работы $13,6\%$, выполняясь за $6157,79$ секунды вместо $5422,67$. Поэтому имеет смысл в средствах запуска задач на суперкомпьютерах предусматривать возможность отключения механизм прозрачных больших страниц, чтобы пользователи могли выбирать оптимальный для своих приложений режим работы подсистемы виртуальной памяти.

Ответ на основой вопрос о том, являются ли нити более эффективным инструментом организации многозадачных вычислений в системе с общей памятью в сравнении с процессами, является неоднозначным. По результатам проведённых измерений видно, что разная скорость переключения контекста выполнения не есть адекватный критерий для выбора между нитями и процессами. Значимым фактором в этом выборе является соотношение между интенсивностью выполнения приложением операций управления ресурсами (не только адресными пространствами) и интенсивностью выполнения им операций обмена данными, которые формируются в независимо резервируемых задачами областях памяти. Нити позволяют достичь большей эффективности, если требуется сделать относительно много подобных обменов, в том числе и большей эффективности в использовании памяти, позволяя не создавать копии данных. Важным аргументом в пользу нитей может быть и то, что создание общих для задач данных при программировании с нитями требует существенно меньших усилий от программиста. Однако, причиной этому являются не микроархитектурные детали исполнения разных видов задач, а особенности интерфейсов ядер современных популярных операционных систем, и ориентированность большинства языков на программирование в общем адресном пространстве.

Между тем, процессы являются более эффективным и удобным средством управления ресурсами и, что не менее важно, эффективным инструментом изоляции областей сбоя (failure domains). К тому же, процессы — необходимый для организации распределённых вычислений на неоднородных системах тип задач. Это существенные преимущества, использование которых в полной мере ограничено недостаточным развитием операционных систем и средств программирования. И отчасти оно сдерживается именно общепринятым мнением о том, что «нити эффективнее процессов». Проведённые в ходе работы над этой статьёй измерения показывают, что такое мнение далеко от реального положения дел. Полученный результат подтверждает верность идей, на которых строятся операционные системы и средства программирования, опирающиеся именно на абстракцию процессов, как на основное средство организации многозадачных вычислений [14, 15].

Работа выполнена в рамках программы Президиума РАН № 18 «Алгоритмы и математическое обеспечение для вычислительных систем сверхвысокой производительности» при поддержке проекта УрО РАН 12-П-1-1034.

Литература

1. AMD64 Architecture Programmer's Manual Volume 2: System Programming – Advanced Micro Devices, 2011.

2. Appleton, R. Understanding a Context Switching Benchmark // Linux J. – 1999. – № 57. – P. 1–6.

3. Sigoure, B. How long does it take to make context switch? – 2010. – URL: http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html (дата обращения: 16.10 2012).

4. SunOS Multi-thread Architecture / M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks // Proceedings of the Winter USENIX Conference. – Dallas, TX, 1991. – P. 65–80.

5. The SPARC Architecture Manual. Version 8. – Menlo Park, CA: SPARC International Inc, 1991.

6. Performance of Address-Space Multiplexing on the Pentium / V. Uhlig, U. Dannowski, E. Skoguld, A. Haeberlen, G. Heiser. – 2002.

7. Jacob, B. Virtual Memory in Contemporary Microprocessors / B. Jacob, T. Mudge // IEEE Micro. – 1998. – V. 18, № 4. – P. 60–75.

8. Таненбаум, Э. Современные Операционные системы. – 3-е изд. / Э. Таненбаум. – СПб: Питер, 2010.

9. An Overview of the Singularity Project / G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and Brian D. Zill. – Microsoft Research, 2005.

10. Исходные тексты Linux 3.3.4. Функция `switch_mm`. – URL: http://lxr.linux.no/#linux+v3.3.4/arch/x86/include/asm/mmu_context.h#L33 (дата обращения: 16.10.2012).

11. Бахтерев, М.О. Набор тестов Thread proc benchmark. – URL: `https://github.com/coda/thread-proc` (дата обращения: 16.10.2012).

12. Snyder, P. tmpfs: A Virtual Memory File System // Proceedings of the European USENIX Conference. – France, Nice, 1990. – P. 241–248.

13. A 40nm 16-Core 128-Thread CMT SPARC SoC Processor / J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, A. Strong // ISSCC Digest. – 2010. – № 56. – P. 98–99.

14. Your computer is already a distributed system. Why isn't your OS? / A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, R. Isaacs // Proceedings of the 12th HotOS Workshop. – Monte Verità, 2009. – P. 19.

15. Методика распределенных вычислений RiDE / М.О. Бахтерев, П.А. Васёв, А.Ю. Казанцев, И.А. Альбрехт, // Параллельные вычислительные технологии (ПаВТ'2011): тр. междунар. науч. конф. – М., 2011. – С. 418–426.

Михаил Олегович Бахтерев, младший научный сотрудник отдела системного обеспечения Института математики и механики Уральского отделения Российской Академии Наук (г. Екатеринбург, Российская Федерация), mike.bakhterev@gmail.com.

MSC 68M20

Thread «Efficency» on the Shared Memory Multiprocessors

M.O. Bakhterev, Institute of Mathematics and Mechanics, Ural Branch of the Russian Academy of Sciences (Yekaterinburg, Russian Federation)

It is tradition to assume that computation decomposed in the certain way into several threads is executed on the systems with shared memory (SMP or NUMA) more efficiently than the same computation but decomposed into several processes. In the presented work we hypothesize that this assumption may be false for the computations with big data volumes, mainly by two reasons. Firstly, the support of common shared address space for the treads may introduce substantially more overhead than aggregate expenses on the execution context switching between processes. Secondly, even when computation does not require intensive memory management, the natural limitation for the memory workset description volume stored in TLB results in necessity to frequently renew that translation cache in the case of using threads too. Experiments and their results which prove our hypothesis correctness are described later in the article.

Keywords: shared memory, performance, threads, processes.

References

1. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* Advanced Micro Devices, 2011.
2. Appleton R. Understanding a Context Switching Benchmark. *Linux Journal*, 1999, no. 57, pp. 1–6. Available at http://www.linuxjournal.com/article/2941 (accessed 16.10.2012).
3. Sigoure B. *How Long Does it Take to Make Context Switch?.* 2010. Available at http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html (accessed 16.10.2012).
4. Powell M.L., Kleiman S.R., Barton S., Shah D., Stein D., Weeks. M. SunOS Multi-thread Architecture. *Proceedings of the Winter USENIX Conference*, Dallas, TX, 1991, pp. 65–80.
5. *The SPARC Architecture Manual. Version 8.* Menlo Park, CA, SPARC International Inc, 1991.
6. Uhlig V., Dannowski U., Skoguld E., Haeberlen A., Heiser G. *Performance of Address-Space Multiplexing on the Pentium.* 2002. Available at http://www.cs.rice.edu/~ahae/papers/smallspaces.pdf (accessed 16.10.2012).
7. Jacob B., Mudge T. Virtual Memory in Contemporary Microprocessors. *IEEE Micro,* 1998, vol. 18, no. 4, pp. 60–75.
8. Tanenbaum A. *Modern Operating Systems, 3rd Edition.* New Jersey, Pearson, 2009.
9. Hunt G., Larus J.R., Abadi M., Aiken M., Barham P., Fahndrich M., Hawblitzel C., Hodson O., Levi S., Murphy N., Steensgaard B., Tarditi D., Wobber T., Zill B.D. *An Overview of the Singularity Project.* Microsoft Research, 2005. Available at http://research.microsoft.com/pubs/52716/tr-2005-135.pdf (accessed 16.10.2012).
10. *Linux 3.3.4 source code,* `switch_mm` *function.* Available at http://lxr.linux.no/#linux+v3.3.4/arch/x86/include/asm/mmu_context.h#L33 (accessed 16.10.2012).
11. Bakhterev M.O. *Thread proc benchmark.* 2012. Available at https://github.com/coda/thread-proc (accessed 16.10.2012).
12. Snyder P. tmpfs: A Virtual Memory File System. *Proceedings of the European USENIX Conference.* France, Nice, 1990, pp. 241–248.
13. Shin J., Tam K., Huang D., Petrick B., Pham H., Hwang C., Li H., Smith A., Johnson T., Schumacher F., Greenhill D., Leon A., Strong A. A 40nm 16-Core 128-Thread CMT SPARC SoC Processor. *ISSCC Digest,* 2010, no. 56, pp. 98–99.

14. Baumann A., Peter S., Schüpbach A., Singhania A., Roscoe T., Barham P., Isaacs R. Your Computer is Already a Distributed System. Why isn't Your OS? *Proceedings of the 12th HotOS Workshop*. Monte Verità, 2009, p. 19.

15. Bakhterev M.O., Vasev P.A., Kazantsev A.Y., Albrekht I.A. RiDE: The Distributed Computation Technique [Metodika Raspredelennyh Vychislenij RiDE]. *Trudy Konferencii PaVT'2011* [Proceedings of PaCT'2011 International Conference]. Moscow, 2011, pp. 418—426.

Поступила в редакцию 29 июня 2012 г.