Многоуровневый параллелизм в программах, использующих параллельные библиотеки

В.А. Бахтин [1], Катаев Никита Андреевич[1], А.С. Колганов[1], Д.А. Захаров[1], А.А. Смирнов[1], А.А. Малахов[2]

[1]ИПМ им. М.В. Келдыша РАН

[2]Университет НЕЙМАРК

DVM SYSTEM

Как добиться эффективных вычислений…

Аппаратная составляющая

- Прирост производительности отдельно взятого ядра от года к году - незначителен.
- Основной источник роста производительности всей системы – увеличение числа процессорных ядер и аппаратных потоков, выполняемых одним ядром.

Программная составляющая

- Растет число потоков, участвующих в вычислениях.
- Каждый поток получает меньшую порцию работы, чем раньше.
- Возрастает влияние накладных расходов на планирование и синхронизацию потоков.
- Доступный параллелизм в программе ограничен законом Амдала.

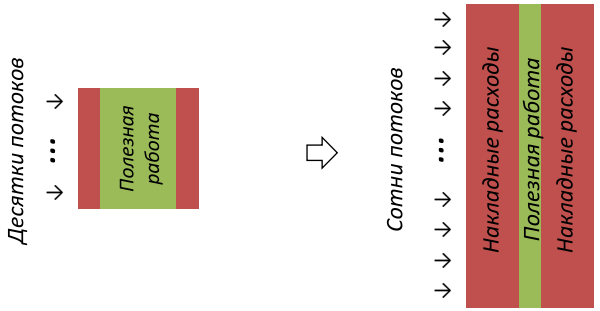
! ***Таким образом, увеличивается разрыв между достигаемой производительностью прикладной программы и пиковыми возможностями современных мультипроцессоров.***

✓ ***Нужно больше параллелизма!***

Возможные источники дополнительного параллелизма:

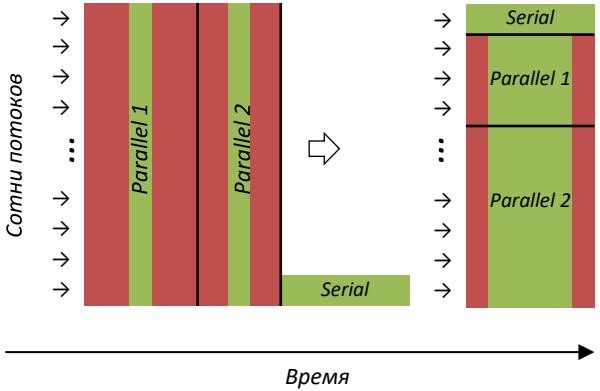
- Вложенный параллелизм в рамках одной программы.
- Совместное выполнение нескольких параллельных программ..

! ***Однако, использование потоков уровня ОС напрямую для исполнения дополнительного параллелизма часто ведет к существенном падению производительности из-за превышения доступных аппаратных возможностей (oversubscription).***



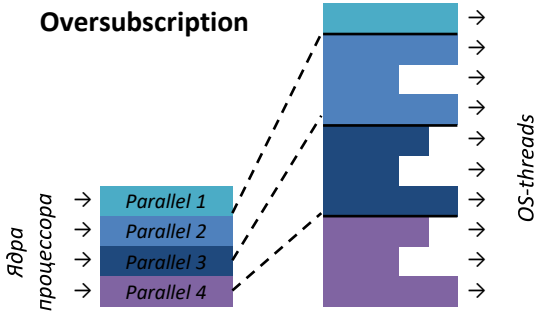
Kunpeng 920 (до 192 потоков)
Intel (до 288 потоков)
AMD (до 768 потоков)

Частота в среднем ~3.0 GHz



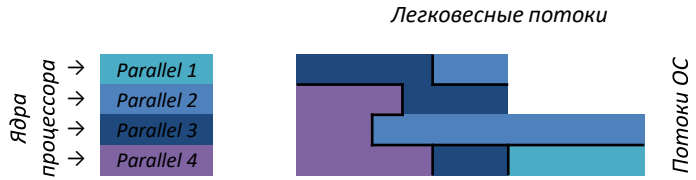
Многоуровневый параллелизма (composability)

Oversubscription



Два уровня параллелизма в худшем случае ведут к использованию P*P потоков уровня ОС при доступных P вычислительных ядрах.

Переупорядочивание вычислительной работы (space-time composition)



Наше исследование направлено на поиск путей прозрачной замены потоков уровня ОС на легковесные потоки с минимальным участием прикладного программиста.

В исследовании было рассмотрено влияние вложенного параллелизма на эффективность существующих программ и, в том числе, программ, использующих библиотеки для ускорения отдельных этапов вычислений.

- **oneAPI Threading Building Blocks (TBB)** – высокоуровневая C++ библиотека, которая предоставляет разработчику средства разработки параллельных программ, основанные на явном использовании параллелизма задач, однако ограничивает возможности ручного управления исполнением программы,
- **Argobots** - легковесная библиотека пользовательских потоков и задач с низкоуровневыми примитивами синхронизации, доступными пользователю, но требующая тонкой ручной настройки параллельной программы для достижения желаемой эффективности,
- **явное использование задач OpenMP (конструкции task и taskloop)** – стандартизованное использование, но возможности не достаточно гибкие для реализации различных стратегий исполнения потоков в сравнении с TBB и Argobots.

Различные пути реализации легковесных потоков на основе существующих моделей программирования.

Прототип системы компиляции

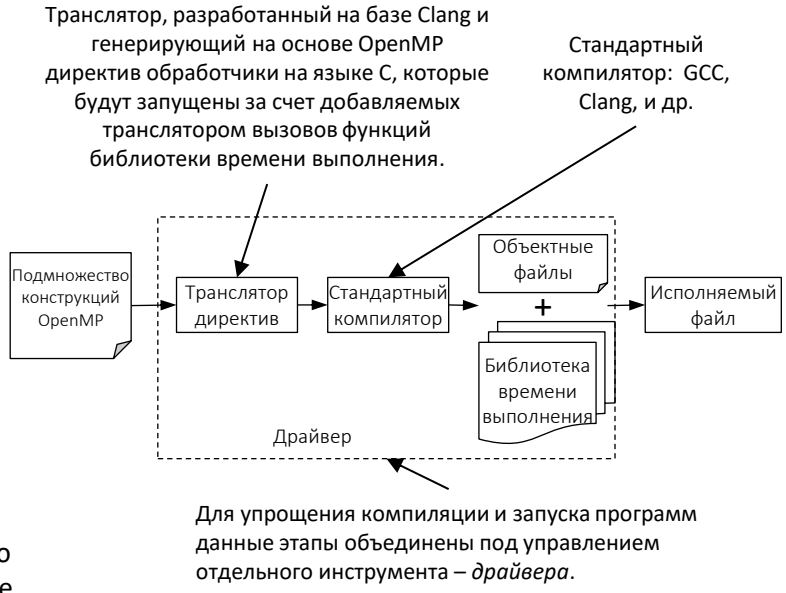
Для описания параллелизма используется подмножество стандарта OpenMP:

- ✓ OpenMP – широко распространенный стандарт позволяющий выражать доступный в программе параллелизм за счет ограниченных изменений в исходном коде.
- ✓ Выбор используемой реализации библиотеки времени выполнения происходит на этапе компиляции и не требует изменения исходного кода программы.
- ✓ Исходная программа также может компилироваться напрямую стандартными компиляторами, использующими стандартную реализацию OpenMP.

Транслятор преобразует конструкции OpenMP для компоновки с разработанными библиотеками времени выполнения:

- ✓ Единый интерфейс для разных реализаций библиотеки времени выполнения (транслятор не зависит от целевой библиотеки).
- ✓ Статический анализ на этапе компиляции позволяет получить дополнительную информацию о программе. Например, возможны разные подходы к обработке конструкций `parallel` и `parallel for` с целью оптимизации числа легковесных потоков, порождаемых для выполнения итераций цикла.

Переменные окружения позволяют управлять количеством потоков уровня операционной системы, задавать границы изменения числа порождаемых легких потоков, управлять размером стека не поток, управлять стратегиями планирования потоков и правилами распределения нагрузки между NUMA узлами.



- **`./cpt c –tbb file.c –o file.tbb.out`**
- **`CP_NUM_STREAMS=48`**
  `CP_PARTITIONER_KIND=2 #affinity partitioner`
  `CP_NUMAS=0,1 CP_STREAM_BIND=1 #enable binding`
  `./cpt run file.tbb.out`

Прототип системы компиляции

Для описания параллелизма используется подмножество стандарта OpenMP:

- ✓ OpenMP – широко распространенный стандарт позволяющий выражать доступный в программе параллелизм за счет ограниченных изменений в исходном коде.
- ✓ Выбор используемой реализации библиотеки времени выполнения происходит на этапе компиляции и не требует изменения исходного кода программы.
- ✓ Исходная программа также может компилироваться напрямую стандартными компиляторами, использующими стандартную реализацию OpenMP.

Транслятор преобразует конструкции OpenMP для компоновки с разработанными библиотеками времени выполнения:

- ✓ Единый интерфейс для разных реализаций библиотеки времени выполнения (транслятор не зависит от целевой библиотеки).
- ✓ Статический анализ на этапе компиляции позволяет получить дополнительную информацию о программе. Например, возможны разные подходы к обработке конструкций `parallel` и `parallel for` с целью оптимизации числа легковесных потоков, порождаемых для выполнения итераций цикла.

Переменные окружения позволяют управлять количеством потоков уровня операционной системы, задавать границы изменения числа порождаемых легких потоков, управлять размером стека не поток, управлять стратегиями планирования потоков и правилами распределения нагрузки между NUMA узлами.

Множество поддерживаемых конструкций OpenMP в зависимости от выбранного способа реализации библиотеки времени выполнения

Construct	TBB	Task	Taskloop	ABT	Taskflow
`parallel for`	+	+	+	+	+
`parallel`	+	+	+	+	+
`for`	+			+	
`critical`	+	+	+	+	
`master`	+	+	+	+	+
`single`	+	±	±	+	±
`barrier`	+			+	
`flush`	+	+	+	+	+
`nowait`	+	+	+	+	+
`num_threads`	+	+	+	+	+
`private`	+	+	+	+	+
`reduction`	+	+	+	+	+
`omp_set_num_threads`	+	+	+	+	+
`omp_get_num_threads`	+	+	+	+	+
`omp_get_max_threads`	+	+	+	+	+
`omp_get_thread_num`	+	+	+	+	+
`omp_in_parallel`	+	+	+	+	+
`omp_get_num_procs`	+	+	+	+	+
`omp_init_lock`	+	+	+	+	
`omp_destroy_lock`	+	+	+	+	
`omp_set_lock`	+	+	+	+	
`omp_unset_lock`	+	+	+	+	
`omp_tesk_lock`	+	+	+	+	
`omp_get_wtime`	+	+	+	+	+

Особенности хранения контекста выполнения потока

Чтобы иметь возможность в любой точке программы идентифицировать пользовательский поток, который ее обрабатывает, необходимо сохранить информацию о потоке в глобальной области видимости – **контекст выполнения потока**.

Одновременно могут выполняться несколько пользовательских потоков (для каждого – свой поток уровня ОС): для хранения контекста выполнения используется **thread-local-storage (TLS)**.

! Стандартный компилятор может некорректно оптимизировать следующий фрагмент кода:

```
#pragma omp parallel for
for ( int i = 0; i < N; ++i) {
  // Некоторая работа внутри родительского легковесного потока.
  // Пусть поток уровня ОС, исполняющий родительский легковесный поток в этой точке, имеет идентификатор TID_1.
  #pragma omp parallel for
  for ( int j = 0; j < N; ++j) {
    // Некоторая работа внутри вложенного дочернего легковесного потока.
  }
  // Пусть поток уровня ОС, исполняющий родительский легковесный поток в этой точке, имеет  идентификатор TID_2.
  // Рабочий поток уровня ОС мог измениться после выхода из внутреннего параллельного цикла: TID_1 может не совпадать с TID_2.
```

! Если объявление TLS переменной видимо для стандартного компилятора в момент оптимизации кода, то компилятор может предположить, что данное значение осталось неизменным, так как в коде программы отсутствуют явные указания на возможность его изменения (стандартный компилятор не знает о пользовательских потоках и о том, что они могут менять принадлежность к потоку операционной системы, который их исполняет).

✓ Вынести обращение к TLS переменным в специальные функции-обертки, расположенные в отдельной единице компиляции.

✓ В момент сборки программы отключить межмодульную оптимизацию и подстановку тел функций-оберток в точках вызова до и после момента исполнения внутреннего цикла.

Особенности реализации

! Динамические библиотеки могут выполнять инициализацию до начала выполнения функции `main`, в том числе обращаясь к функциям OpenMP. В этом случае библиотека времени выполнения OpenMP должна быть инициализирована ранее.

✓ Управлять порядком вызова функций в момент загрузки динамической библиотеки, а также порядком освобождения ресурсов при завершения использования библиотеки можно, указав специальные атрибуты функций:

```
static void __attribute__ ((constructor(101))) inititialize() { /* Инициализация библиотеки времени выполнения */; }
static void __attribute__ ((destructor(101))) free() { /* Освобождение выделенных ресурсов */; }
```

! Следование стандарту OpenMP требует реализовать возможность барьерной синхронизации между выполняющимися легковесными потоками. Синхронизация может быть востребована для реализации особенностей некоторых алгоритмов, например для организации конвейера. Приостановка легковесного потока не должна вызывать приостановку исполняющего его потока уровня ОС, чтобы гарантировать отсутствие взаимной блокировки потоков, когда все исполняющие потоки заблокированы, а невыполненные задачи еще остаются в очереди на выполнение.

✓ При использовании задач OpenMP такая реализация достаточно проблематична.
Некоторые специальные библиотеки (Argobots) поддерживают такой механизм синхронизации явно.
В случае TBB для реализации необходимой функциональности нами был использован механизм приостановки задач (resumable tasks).

```
oneapi::tbb::task::suspend([&] (tbb::task::suspend_point tag) { /* Запомнить идентификатор точки приостановки (tag) */ }
oneapi::tbb::task::resume(tag)  // Возобновить приостановленную задачу
```

✓ В случае TBB был реализован механизм принудительного распределения пользовательских потоков по NUMA узлам.

```
tbb::task_arena // Для каждого узла будет создана отдельная арена для выполнения части порождаемых пользовательских потоков.
```

Эксперименты с OpenBLAS

Многократный запуск различных вычислительных ядер, выполняющих операции над матрицами разных типов данных (*float*, *double*, *complex float*, *complex double*), с указанием числа потоков на внешнем (*num_threads_out*) и внутреннем (*num_threads_in*) уровнях, стратегии распределения работ по потокам (*schedule*) и количества выполняемых однотипных вычислительных ядер.

```
// Задание числа потоков для внутреннего уровня параллелизма,
реализующего параллельное выполнение отдельного вычислительного ядра.
omp_set_num_treads(num_threads_in)

total_time = omp_get_wtime();

// Внешний уровень параллелизма – многократное выполнения одного и того
// же вычислительного ядра над входными данными разных размеров (m x m).
// Количество и сложность решаемых задач задается при помощи параметров
// цикла from, to и step.
#pragma omp parallel for num_threads(num_threads_out) \
                         schedule(runtime) private ...
for (m = from; m <= to; m+= step) {
  // Одно из вычислительных ядер SYMM, HEMM, TRSM
}

total_time = omp_get_wtime() – total_time;
```

Преобразование матриц SYMM и HEMM

$$1400 \le m \le 1800$$

$$C = \alpha * A * B + \beta * C$$

или

$$C = \alpha * B * A + \beta * C$$

Решение уравнения TRSM

$$1000 \le m \le 1200$$

$$A * X = \alpha * Y$$

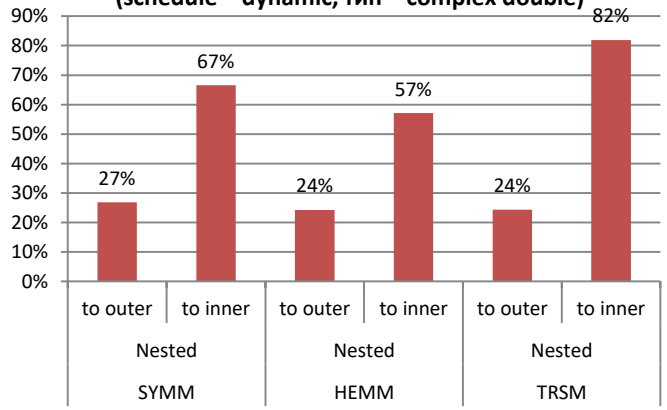
или

$$X * A = \alpha * Y$$

Эксперименты с OpenBLAS

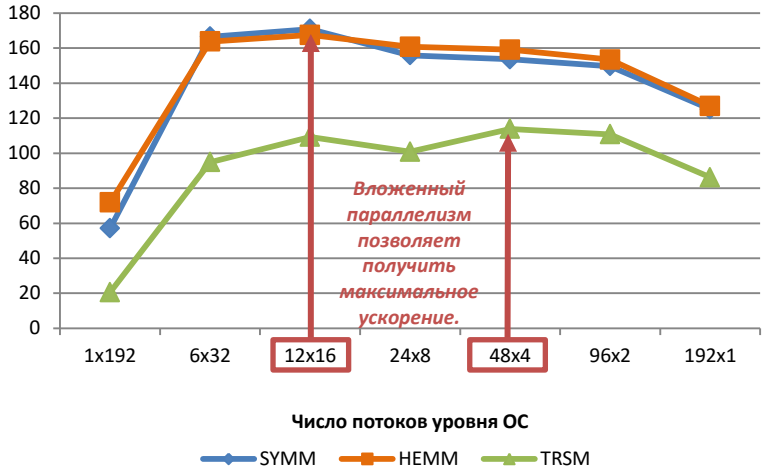
- Kunpeng CPU @ 2.6 GHz (архитектура aarh64), 192 ядра, 8 NUMA узлов, 24 ядра на узел, 1 поток уровня ОС на ядро, 503 GB, Ubuntu 18.04.6 LTS.
- Библиотека OpenBLAS собиралась с опциями по умолчанию, заданными в настройках библиотеки для конкретных целевых архитектур.
- Стандартная реализация GCC 13.0.2 OpenMP с включенной поддержкой вложенного параллелизма и подбором числа потоков на разных уровнях.
- Программы собирались с опцией –O3.

Прирост производительности, получаемый за счет использования вложенного параллелизма (schedule – dynamic, тип – complex double)



$1400 \le m \le 1800$ $1000 \le m \le 1200$

Ускорение относительно последовательного исполнения (schedule – dynamic, тип – complex double)



Источники несбалансированной нагрузки на внешнем уровне параллелизма:

- Разные размеры матриц при нескольких запусках одного вычислительного ядра.
- Несоответствие числа запусков ядра числу используемых вычислительных ядер.

Эксперименты с HPL

- Kunpeng 920 CPU @ 2.6 GHz (архитектура aarch64), 48 ядер, 2 NUMA узла, 24 ядра на узел, 1 поток уровня ОС на ядро, 512 GB, CentOS Linux 8
- Дистрибутив Miniconda, TBB 2021.11, LLVM 17.0.6, GCC 13.2.0 с указанием опции –O3 при компиляции приложений.
- Библиотека OpenBLAS собиралась с опциями по умолчанию, заданными в настройках библиотеки для конкретных целевых архитектур.
- Разработанная версия библиотеки времени выполнения на основе TBB.

! Два уровня параллелизма:

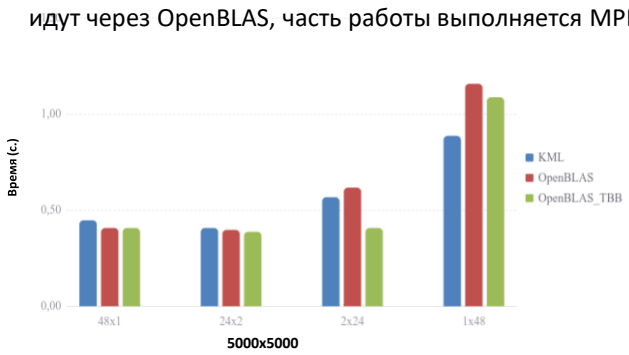
- MPI - внешний уровень параллелизма.
- OpenBLAS (OpenMP) - внутренний уровень параллелизма.

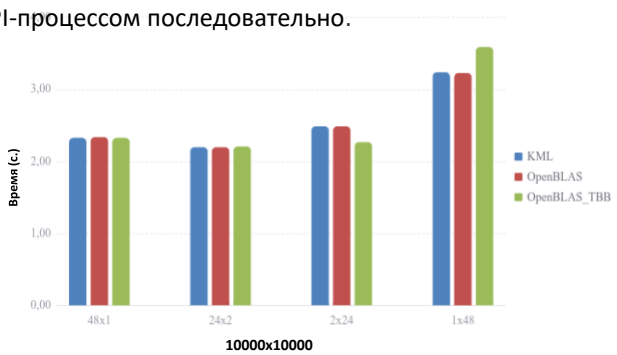
✓ В вычислительных ядрах OpenBLAS может использоваться как стандартная версия OpenMP, так и реализованная на основе библиотеки TBB.

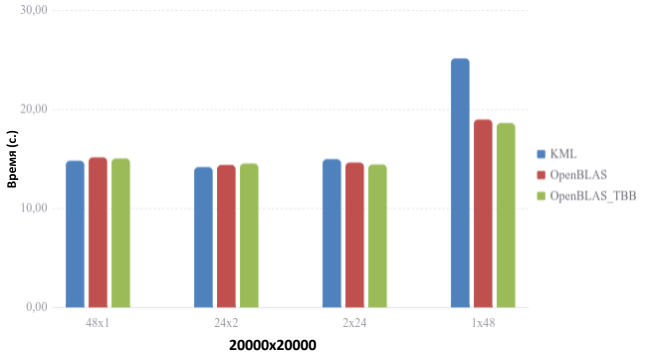
✓ Вложенный параллелизм обеспечивает прирост производительности около 3-5% по сравнению с использованием только распараллеливания с помощью MPI (внешний уровень). Оптимальная конфигурация: 24 процесса MPI и 2 потока уровня ОС в процессе.

По эффективности выполнения TBB-версия теста HPL, полученная с использованием разработанного в рамках данного исследования компилятора, практически не уступает KML-версии.

Одним из возможных объяснений плохого ускорения теста HPL при использовании потоков является то, что не все вычисления, выполняемые процессом, идут через OpenBLAS, часть работы выполняется MPI-процессом последовательно.





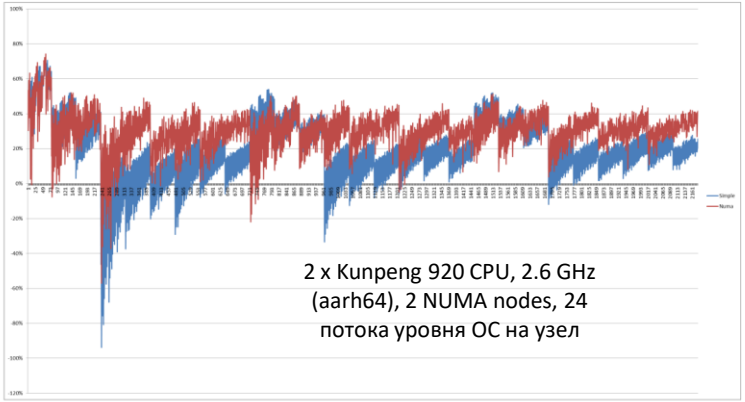


Эксперименты с тестами NAS NPB

! Одного приложения недостаточно, чтобы загрузить систему полностью, но можно выполнять несколько различных приложений одновременно.

✓ Возможно ограничить число создаваемых легковесных потоков для каждого цикла. Динамическое планирование легковесных потоков снижает влияние числа потоков и размера отдельной работы на общую производительность приложения.

Влияние привязки к NUMA узлам при использовании TBB в сравнении с исходными OpenMP версиями

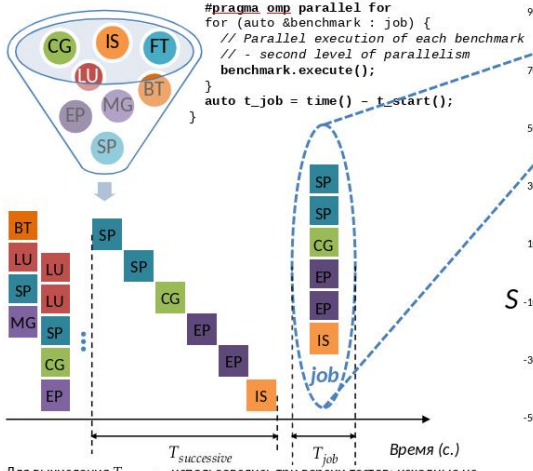


2 x Kunpeng 920 CPU, 2.6 GHz (aarh64), 2 NUMA nodes, 24 потока уровня ОС на узел

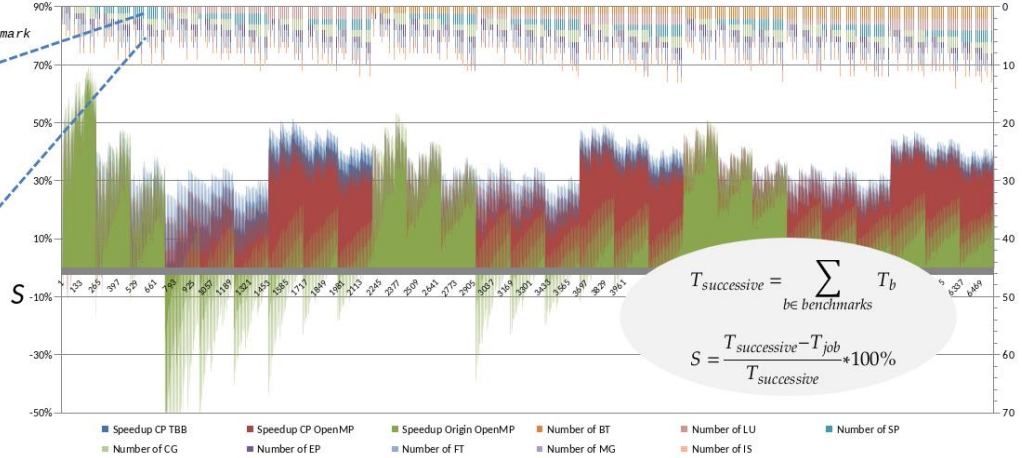
```
// Successive execution of all jobs
for (auto &job : list_of_jobs) {
  // Parallel execution of all benchmarks in a job
  // - first level of parallelism.
  auto t_start = time);
  #pragma omp parallel for
  for (auto &benchmark : job) {
    // Parallel execution of each benchmark
    // - second level of parallelism
    benchmark.execute();
  }
  auto t_job = time() - t_start();
}
```

```
for (auto &benchmark : job) {
  auto t_start = time);
  benchmark.execute();
  auto t_b = time() - t_start();
}
```

.vs



Для вычисления $T_{successive}$ использовались три версии тестов: исходные на стандартном OpenMP , измененные на стандартном OpenMP, измененные на разработанной библиотеке времени выполнения.



$$T_{successive} = \sum_{b \in benchmarks} T_b$$

$$S = \frac{T_{successive} - T_{job}}{T_{successive}} * 100\%$$

Одновременное выполнение нескольких параллельных тестов из NPB, класс A на Kunpeng (48 потоков уровня ОС). Используется библиотека времени выполнения на основе TBB, количество легковесных потоков определяется в динамике.

Дальнейшая оптимизация:

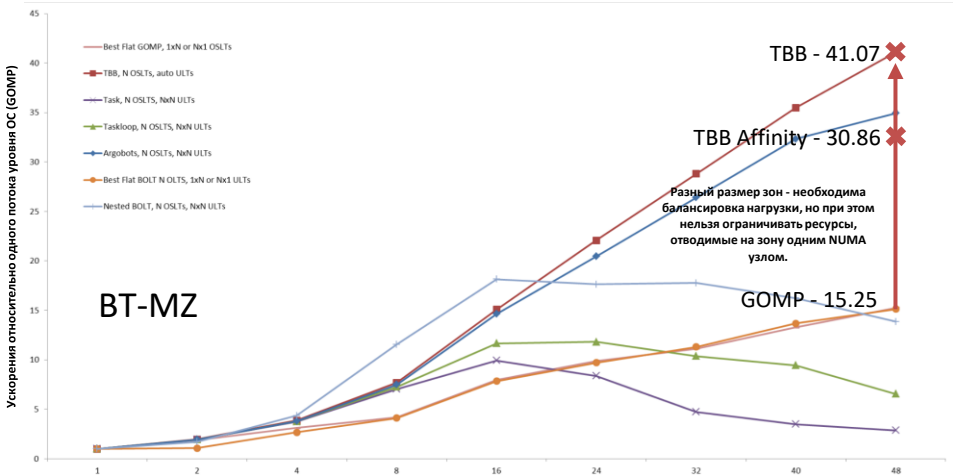
- *исследование особенностей каждого приложения,*
- *соотнесение необходимых и доступных ресурсов.*

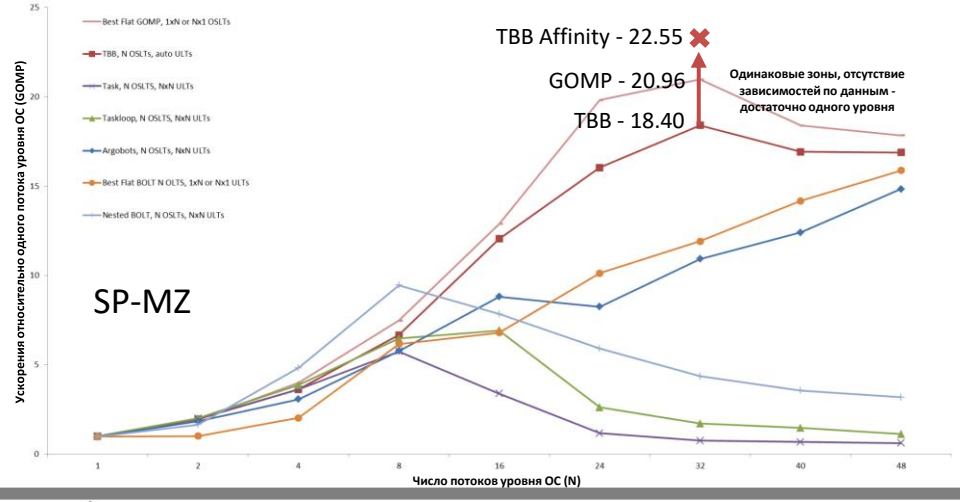
Эксперименты с NAS NPB MZ

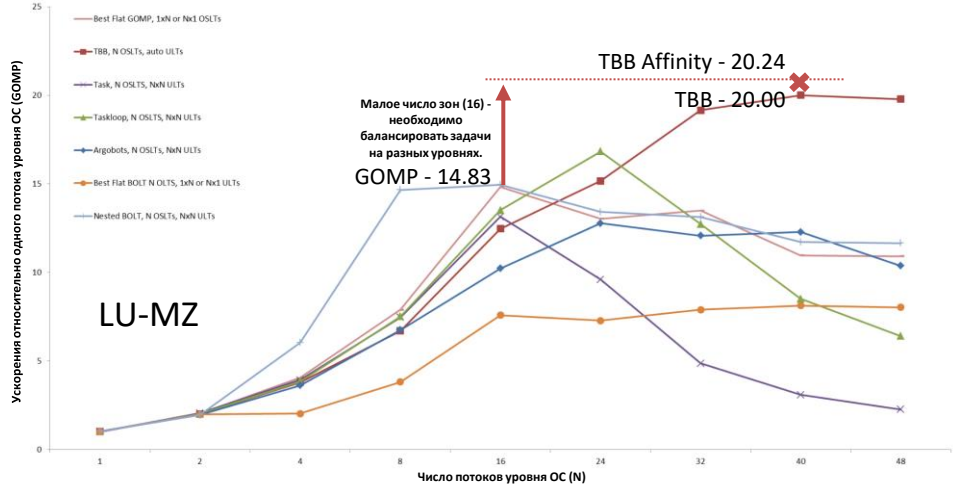
! Разный или небольшой объем работы для разных потоков уровня ОС не позволяет эффективно загрузить систему без дополнительной балансировки нагрузки.

✓ В данных тестах (MZ - несколько вычислительных зон) вложенный параллелизм присутствует явно.

Для пользовательских потоков возможно эффективно организовать динамическую балансировку, в том числе ограничив число потоков уровня ОС на зону и связав вычисления с отдельными NUMA узлами.







Заключение

! В нашем исследовании мы сконцентрировались на поиске способов преодоления барьера масштабируемости и считаем, что наше исследование может повлиять на принципы организации будущих многопоточных библиотек.

✓ Модели параллельного программирования, напрямую построенные на потоках уровня ОС, неадекватны в условиях перегрузки, но их возможно прозрачно заменить на легковесные подходы с минимальными изменениями в программных кодах, не затрагивающими пользовательский API, используемый современными моделями программирования, такими как OpenMP.

Нам удалось увеличить пропускную способность в среднем на 40%, используя существующие аппаратные и программные средства (с минимальными изменениями).

Стандартные подходы к реализации вложенного параллелизм (GNU OpenMP, LLVM OpenMP) требуют:

- ручной настройки количества потоков на каждом уровне,
- ручной балансировки нагрузки между внутренними уровнями различных порций работ.

Предложенный подход (composable parallelism) позволяет автоматически настраивать программу во время выполнения. Тем не менее, некоторые подсказки пользователя или ручная балансировка нагрузки могут влиять на производительность. ⇒ Анализ программы на этапах компиляции и выполнения может дополнительно улучшить производительность.

Спасибо за внимание

URL

http://dvm-system.org

dvm@keldysh.ru

E-mail