Статический анализ Си программ в системе SAPFOR для их распараллеливания

Н.А. Катаев[1], Ю.А. Лапенко[2]

[1]Институт прикладной математики им. М.В. Келдыша РАН

[2] Московский государственный университет им. М.В. Ломоносова

dvm@keldysh.ru

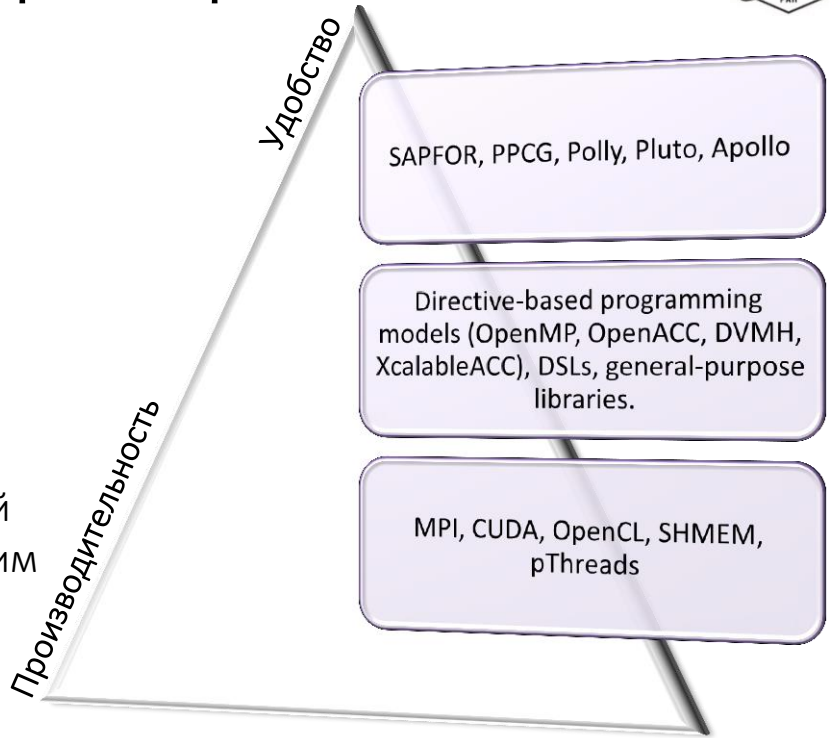
02 декабря, 2021 | Переславль-Залесский

Инструменты параллельного программирования

Автоматически распараллеливающие компиляторы создают параллельный код для входной программы (но не всегда оптимальный).

Директивные языки упрощают программирование и повышают удобство сопровождения ПО, обеспечивая при этом высокую производительность.

Низкоуровневые модели дают программистам точный контроль над выполнением программы и позволяют им добиться наилучшей производительности.



Наш подход к использованию параллелизма в последовательных и MPI-программах (C и Fortran):

- использование директивных моделей Fortran-DVMH и CDVMH;
- использование системы автоматизации SAPFOR;
- участие пользователя в процессе распараллеливания.

DVMH-модель

Язык параллельного программирования высокого уровня.
Модель программирования на основе директив.

DVMH

CUDA

OpenMP

MPI

Модель программирования на основе директив, которая направлена на создание параллельных программ для гетерогенных вычислительных кластеров (GPU NVidia, Intel Xeon Phi, многоядерные процессоры).

Модель включает в себя два языка программирования, которые являются расширениями стандартных языков C и Fortran спецификациями параллелизма: CDVMH и Fortran-DVMH

Параллельная программа разрабатывается в терминах последовательной.

SAPFOR (System FOR Automated Parallelization)

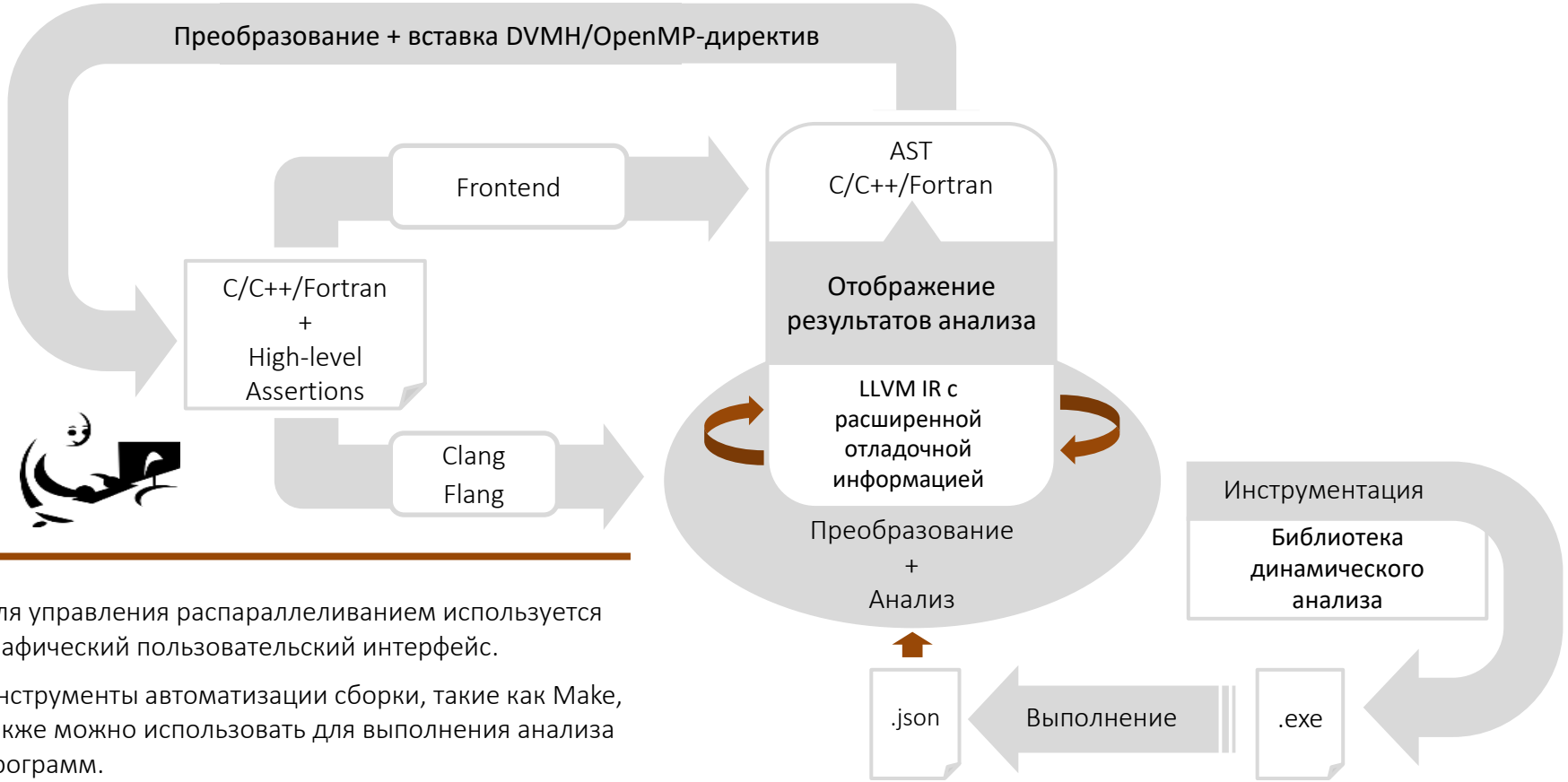
- Исследование последовательных программ (анализ и профилирование программ).
- Автоматическое распараллеливание (в соответствии с моделями DVMH или OpenMP) потенциально параллельной программы, для которой программист максимизирует параллелизм на уровне алгоритма и/или добавляет высокоуровневые аннотации для описания свойств программы.

  Следование определенным правилам при разработке программ на традиционных языках программирования, дополнительное описание свойств программ.

- Полуавтоматическое преобразование программы для получения потенциальной последовательной версии исходной программы.

  Устранение зависимостей в программе, оптимизация доступа к памяти, изменение структуры хранения данных и структуры вычислений (подстановка процедур и переменных, преобразование циклов и др.).

Внутреннее устройство SAPFOR



- Для управления распараллеливанием используется графический пользовательский интерфейс.
- Инструменты автоматизации сборки, такие как Make, также можно использовать для выполнения анализа программ.

Автоматическое распараллеливание с использованием DVMH модели

Для распараллеливания на вычислительные системы с распределенной (кластер) и/или общей памятью (многоядерный процессор или ускоритель в узле кластера) требуется, чтобы в исходный код были вставлены четыре вида директив:

- спецификации распределения данных (при использовании распределенной памяти в модели DVMH)
- спецификации для циклов, которые могут выполняться параллельно, а также спецификации приватных и редукционных переменных, а также шаблон доступа к массиву (при использовании только общей памяти),
- спецификация вычислительных областей, которые могут быть выполнены на ускорителях, каждая область может содержать один или несколько параллельных циклов,
- высокоуровневые спецификации передачи данных между памятью центрального процессора и памятью ускорителя (директивы актуализации данных).

Для каждого цикла рассматриваются следующие ограничения:

- безопасность потока управления (отсутствие операций ввода-вывода, побочных эффектов и т.д.),
- безопасность доступа к памяти (отсутствие зависимостей по данным в циклах и «захваченных» указателей),
- направление использования данных (входные, выходные и локальные данные),
- каноническая форма цикла в соответствии со стандартом OpenMP,
- возможность выразить свойство переменной с использованием спецификаций DVMH языков,
- возможность объединения итерационных пространств вложенных циклов в одно общее итерационное пространство.

Цели выполнения преобразований

Повышение качества проводимого анализа и для исследования информационной структуры распараллеливаемой программы:

- может выполняться над внутренним представлением программы в системе;
- нет необходимости преобразовывать исходную программу соответствующим образом;
- невидимо для пользователя.

До	После

`void copy(float *A, int N, int M) {`
`  int X;`
`  for (int I = 0; I < N; ++I) {`
`  X = I + M;`
`  A[I] = A[X];`
`  }`
`}`
	`void copy(float *A, int N, int M) {`
	`  for (int I = 0; I < N; ++I) {`
	`  A[I] = A[I + M];`
	`  }`
	`}`

Данное преобразование может быть скрыто от пользователя.

Повышение качества распараллеливания программы:

- приводит к изменению свойств программы;
- выполняется на уровне исходного языка (Си/Фортран);
- позволяет программисту оценить принимаемые системой решения и внести корректировки в программу.

Дерево псевдонимов: пример

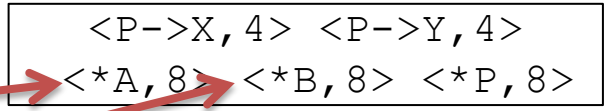
```
struct S {float X; float Y;};
void foo(struct S * restrict A,
struct S * restrict B) {
  struct S *P;
  P = A;
  P->X = 0;
  P->Y = 0;
  P = B;
  P->X = 0;
  P->Y = 0;
}
```

```
<P->X,4> <P->Y,4>
<*A,8> <*B,8> <*P,8>
```

- Участок памяти определяется адресом начала и размером.
- Два участка памяти попадают в одну вершину дерева псевдонимов, если они могут пересекаться.
- Объединение участков памяти из родительской вершины покрывает объединение участков памяти из дочерних вершин дерева.

Дерево псевдонимов: пример

```
struct S {float X; float Y;};
void foo(struct S * restrict A, struct S *
restrict B) {
  struct S *P;
  P = A;
  P->X = 0;
  P->Y = 0;
  P = B;
  P->X = 0;
  P->Y = 0;
}
```

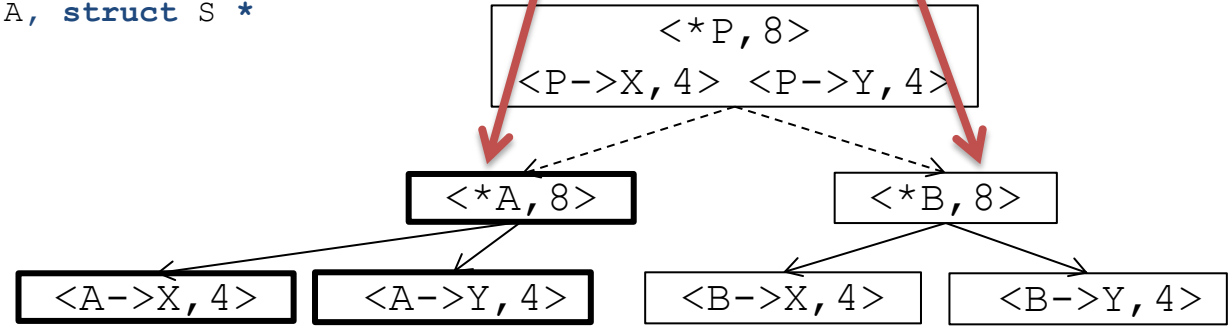


Преобразование внутреннего представления

```
struct S {float X; float Y;};
void foo(struct S * restrict A, struct S *
restrict B) {

  A->X = 0;
  A->Y = 0;

  B->X = 0;
  B->Y = 0;
}
```

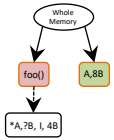


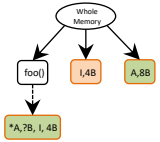
Анализ и преобразование программ

Построение AST + LLVM IR с помощью Clang

Построение дерева псевдонимов

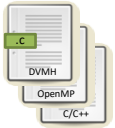
Сохранение дерева в виде метаданных LLVM IR





Анализ

Распараллеливание или Преобразование

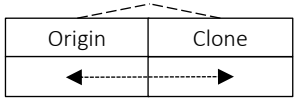


Преобразование исходного кода

Серверы анализа, запущенные в отдельных потоках

Клонирование LLVM IR всей программы

Описание участков памяти уточняется при выполнении затрагивающих их преобразований LLVM IRLLVM IR



Отображение между участками памяти в деревьях псевдонимов исходной и преобразованной программ

Перестроение дерева псевдонимов

Определение соответствия между участками памяти в деревьях псевдонимов исходной и преобразованной программ.

Преобразование и анализ LLVM IR

Объединение полученных результатов анализа и соотнесение их с исходным кодом программы, определение возможности распараллеливания/преобразования исходной программы

Исследование программ из NAS Parallel Benchmarks 3.3.1

Инлайн подстановка функций на уровне исходного кода:

- чтобы убедится, что функция не «захватывает» параметр-указатель (EP, BT),
- чтобы убедится в отсутствии зависимостей по данным (BT),
- чтобы убедится, что параметры-указатели не пересекаются между собой (CG),
- чтобы обеспечить возможность последующих преобразований (EP, LU).

Динамический анализ приватизируемых массивов (BT, EP, LU).

Использование аннотаций пользователя для указания редукционных переменных в MPI версиях программ (EP, CG).

Использование опций анализа:

- чтобы указать, что выражение индекса не выходит за пределы выделенной памяти (BT),
- чтобы игнорировать возможность побочного эффекта математических функций, сохраняемого в переменной *errno* (EP).

Выборочная подстановка функций

Автоматическая подстановка на уровне IR:

- не влияет на исходный код программы,
- сокращает время выполнения статического анализа за счет выбора подставляемых вызовов функций на основе предварительного анализа программы.

```
void bar(float *A) { *A = *A + 1; }

void foo(int N, float *A) {
 for (int I = 0; I < N; ++I)
   bar(A + I);
}
```

⇨

```
void bar(float *A) { *A = *A + 1; }

void foo(int N, float *A) {
  for (int I = 0; I < N; ++I)
    A[I] = A[I] + 1;
}
```

Автоматический выбор вызовов функций, которые должны быть подставлены в IR-коде:

- подстановка функций, вызовы которых порождают потенциальные зависимости по данным в циклах и которые не удалось проанализировать средствами межпроцедурного анализа,
- рассмотрение только тех циклов, которые могут быть кандидатами для распараллеливания (отсутствие операций ввода вывода, безопасные поток управления (нет выброса исключений и вероятности аварийного завершения программы)),
- эвристическая оценка влияния результата подстановки на время выполнения анализа на основе подсчета количества обращений к памяти в преобразованной функции.

Анализ редукционных переменных в MPI-программах

Анализ редукционных и индуктивных переменных в LLVM IR опирается на анализ представления программы в форме однократного присваивания (SSA, каждое использование переменной достигается одним ее определением).

Переменные, участвующие в операциях адресной арифметике, не находятся в SSA форме, так как возможно косвенное изменение описываемой ими памяти.

Возможное решение – построение SSA формы локально для анализируемого цикла: изменение IR-кода для разрыва явной связи между переменными и MPI функциями:

```
double sum = 0.0E0;

for (int j = 1; j <= lastcol - firstcol + 1; ++j)
  sum = sum + r[j - 1] * r[j - 1];

MPI_Send(&sum, 1, dp_type, reduce_exch_proc[i - 1],
         i, MPI_COMM_WORLD);
```

⇨

```
double sum = 0.0E0;
double sum_promoted = sum;
for (j = 1; j <= lastcol - firstcol + 1; ++j)
  sum_promoted = sum_promoted + r[j - 1] * r[j - 1];
sum = sum_promoted;
MPI_Send(&sum, 1, dp_type, reduce_exch_proc[i - 1],
         i, MPI_COMM_WORLD);
```

Автоматический выбор преобразуемых циклов и переменных в IR-коде на основе предварительного анализа LLVM IR:

- скалярные переменные, для которых не была построена SSA форма,
- переменные, для которых не удалось доказать отсутствие зависимости по данным,
- переменные используются в цикле как на чтение так и на запись и не имеют признака volatile,
- в цикле не происходит выброса исключений или других ситуаций приводящих к неожиданному завершению программы.

Статический анализ приватных массивов

Повышает потенциал применения динамического анализа программ, так как сокращает количество фрагментов программ, для которых требуется выполнение динамического анализа.

Возможен точный анализ в случае, если

- границы циклов и шаг – константы,
- известны размеры измерений массивов,
- индексные выражения в каждом измерении массива представлены в аффинном виде и зависят от одной индуктивной переменной цикла (a * i + b).

Возможен консервативный анализ, если:

- шаг цикла равен 1,
- индексные выражения в каждом измерении массива представлены в аффинном виде и зависят от одной индуктивной переменной цикла (a * i + b).

Вычисление пересечения участков памяти

- В случае точного анализа решается диофантово уравнение для каждого изменения массива:

$$D_1 = Start_1 + Step_1 \cdot X, \qquad X \in [0, TripCount_1)$$
$$D_2 = Start_2 + Step_2 \cdot Y, \qquad Y \in [0, TripCount_2)$$

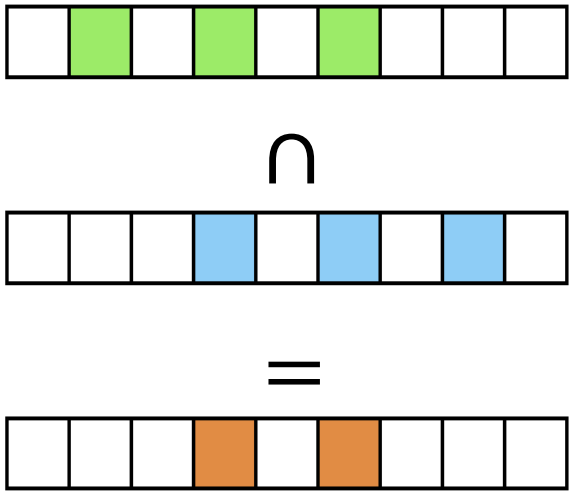
$$TripCount_i = \frac{(End_i - Start_i)}{Step_\mathrm{i}} + 1$$

$$D_1 = D_2$$

- В случае консервативного анализа сравниваются границы диапазонов обращений по каждому измерению массива, возможно с использованием подсказок пользователя:
  - опция анализа SAPFOR –finbounds-subscripts, чтобы гарантировать отсутствие выхода за границы измерения массива,
  - встроенная функция Clang __builtin_assume(bool), чтобы задать ограничения на границы циклов.

- На выходе получается новое измерение:

$$D_3 = [Start_3, \mathrm{End}_3], \qquad Step_3$$



Статический анализ приватных массивов: пример

```
for (;;) { // L1
  for (int I = 0; I < 10; ++I)
    A[I] = ...                    // L2
  for (int J = 0; J < 20; ++J)
    ... = A[J];                   // L3
}
```

$def[L_2] = [0, 10)$

$use[L_3] = [0, 20)$

$def[L_1] = [0, 10)$
$use[L_1] = [10, 20)$

Массив A **не приватный в L_1**
(множество $use[L_1]$ не пусто)

```
for (;;) {      // L1
  for (int I = 0; I < 10; ++I)
    A[I] = ...                    // L2
  for (int J = 0; J < 10; ++J)
    ... = A[J];                   // L3
}
```

$def[L_2] = [0, 10)$

$use[L_3] = [0, 10)$

$def[L_1] = [0, 10)$

Массив A **приватный в L_1**
(множество $use[L_1]$ пусто)

Заключение

В системе SAPFOR реализован автоматически распараллеливающий компилятор, который подходит для распараллеливания потенциально параллельных программ без участия пользователя.

Пользователь может добавлять свойства программы или определять последовательность необходимых преобразований.

Для повышения производительности параллельных программ система SAPFOR может полагаться на различные оптимизации, реализованные в компиляторе и в runtime системе DVMH.

В SAPFOR программа представлена на двух уровнях абстракции (высокий уровень – для описания свойств программы и ее преобразования, низкий уровень – для исследования свойств программы).

Проверки допустимости выполнения параллельного выполнения программы или ее преобразования, связанные с особенностями обращений к памяти (зависимости по данным, анализы потока данных), могут быть выполнены над низкоуровневым представлением программы. При этом дерево псевдонимов позволяет использовать невидимые для пользователя преобразования низкоуровневого представления, чтобы повысить точность описания используемой памяти.

Выполнение реализованных преобразований над внутренним представлением программы:

- упрощает сопровождение исходного кода программы пользователем, так как снижает количество необходимых модификаций исходного кода программы,
- уменьшает время статического анализа, так как позволяет выполнять преобразования после предварительной оптимизации представления программы и только в случае их необходимости,
- повышает потенциал применения динамического анализа программ, так как сокращает количество фрагментов программ, для которых требуется выполнение динамического анализа.

Исходные коды доступны на GitHub: https://github.com/dvm-system

Спасибо за внимание!

НСКФ
НАЦИОНАЛЬНЫЙ СУПЕРКОМПЬЮТЕРНЫЙ ФОРУМ

http://dvm-system.org

https://github.com/dvm-system

KIAM RAS DVM SYSTEM

ОРДЕНА ЛЕНИНА ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ им.М.В.КЕЛДЫША РАН