Технология фрагментированного программирования и система LuNA

Перепелкин В.А.
09.12.2015 г.

Проблематика

Параллельно программировать сложно

Параллельное программирование можно и нужно автоматизировать

Это позволит:

- Облегчить разработку параллельных программ
- Разрабатывать более сложные параллельные программы

Сложности параллельного программирования (базовые)

Требуются:

- Декомпозиция данных и вычислений
- Синхронизация потоков и процессов
- Организация коммуникаций
- Распределение данных и вычислений

Сложности параллельного программирования (продвинутые)

В сложных случаях требуются:

- Выполнение коммуникаций на фоне вычислений
- Асинхронность вычислений
- Динамическая балансировка вычислительной нагрузки
- Учёт неоднородности вычислителя
- Использование спецпроцессоров

Проблематика

Ясно, что использование традиционных средств параллельного программирования трудоёмко и ограничивает пользователя классом относительно простых параллельных программ

Со временем требования к параллельному программному обеспечению будут усиливаться

Система LuNA

- Разрабатывается в ИВМиМГ СО РАН
- Предназначена для поддержки параллельной реализации больших численных моделей для суперкомпьютеров
- LuNA-программа описывается на высоком уровне абстракции
- Система LuNA исполняет программу, автоматизированно настраивая её на доступные ресурсы и обеспечивая динамические свойства её выполнения

Вопросы

- Зачем нужно изучать другой язык программирования?
- Что представляют собой технология фрагментированного программирования и система LuNA?
- Почему они устроены так, как устроены?

Проблема

— На решение какой проблемы направлена технология фрагментированного программирования?

Параллельная реализация крупномасштабных численных моделей для суперкомпьютеров — сложная задача системного параллельного программирования

Цель технологии фрагментированного программирования (ТФП)

— В чем цель технологии фрагментированного программирования?

Автоматизация эффективной параллельной реализации численного алгоритма на суперкомпьютере

Должны обеспечиваться масштабируемость, эффективность, переносимость.

Представление алгоритма

Разные представления алгоритма обладают разными свойствами:

- явность параллелизма
- гранулярность алгоритма
- автономность частей алгоритма
- управление ресурсами

Для разных целей удобны разные представления алгоритма

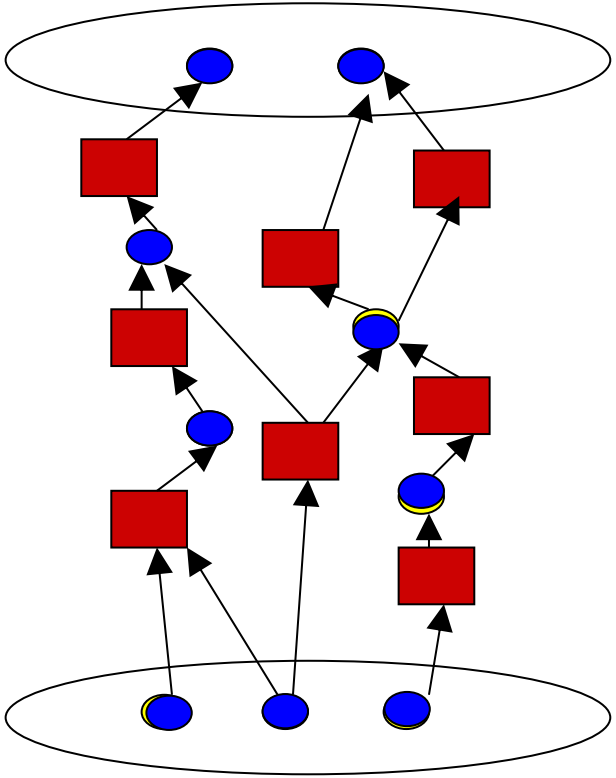
Фрагментированный алгоритм

— Каково представление алгоритма в ТФП и каковы его особенности?

Алгоритм представляется в явно-параллельной форме, ориентированной на автоматизацию обеспечения нефункциональных свойств

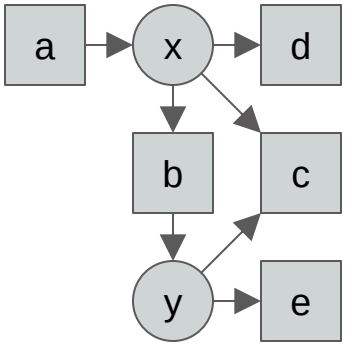
Фрагментированный алгоритм (ФА)

ФА — это набор фрагментов данных (ФД), фрагментов вычислений (ФВ), и отношения in/out
Выходные ФД вычисляются из входных пока все ФВ не окажутся исполненными



Определение фрагментированного алгоритма (ФА)

```
df x, y;
cf a: func1(out: x)
cf b: func2(in: x, out: y)
cf c: func3(in: x, y)
cf d: func4(in: x)
cf e: func4(in: y)
```



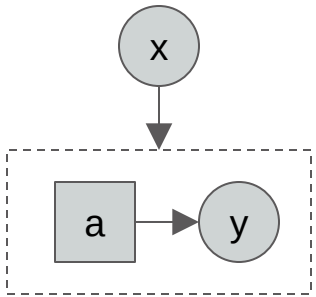
- ФА — это конечное множество фрагментов данных и вычислений (ФД и ФВ), связанных отношениями in и out
- Исполнительная семантика — data flow, то есть, по готовности данных. Выполнение ФВ означает, что значение выходных ФД будет вычислено
- ФД может быть вычислен или не вычислен
- ФД — единтственного присваивания
- Порядок выполнения может быть разным, в т.ч. возможно параллельное выполнение

Особенности ФА

- Порядок выполнения ФА влияет на расход памяти и доступный параллелизм в разные моменты времени
- Допустим любой порядок выполнения ФВ, не противоречащий информационным зависимостям
- ФД возможно передавать по сети, обеспечивая подачу входных аргументов для ФВ
- ФВ возможно передавать по сети, обеспечивая динамическую балансировку нагрузки на вычислительные узлы (*)
- Эффективная реализация ФА является труднорешаемой задачей

Определение ФА (продолжение)

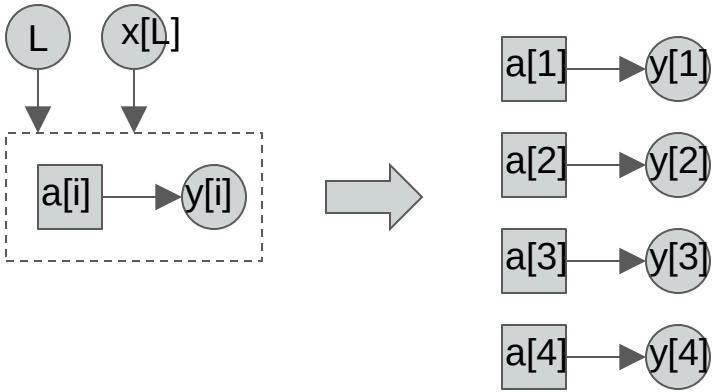
```
if (x>0) {
        df y;
        cf a: func1(out: y)
        ...
}
```



- if — это *структурированный ФВ*
- Входной параметр: выражение
- Входные ФД: необходимые для вычисления выражения
- Если выражение истинно, то ФВ перобразуется в можество фрагментов, описанных в теле оператора
- Множество ФД и ФВ фрагментированного алгоритма зависит от значений ФД

Определение ФА (продолжение)

```
for i=1..x[L] {
          cf a[i]: func1(in: i; out: y[i]);
}
```



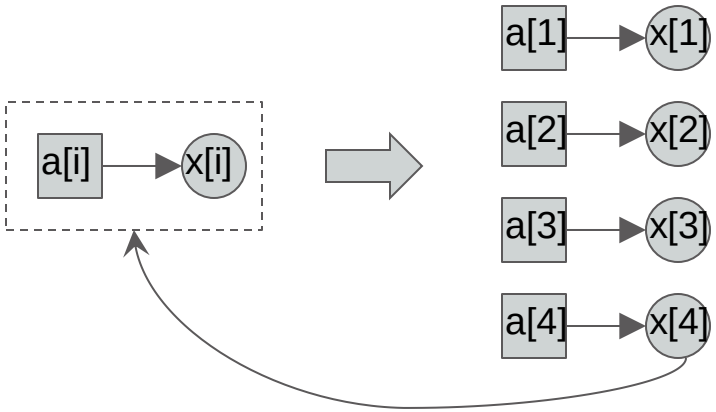
- for — это *структурированный ФВ*
- Входные параметры: выражения для первого и последнего значений параметра цикла — счётчика
- При исполнении ФВ раскрывается на несколько копий тел, в каждом из которых значение счётчика имеет разное значение
- Счётчик не является ФД, он — константное выражение

Особенности раскрутки оператора FOR

- Оператор FOR можно “раскручивать” постепенно в целях экономии памяти
- В этом случае система должна определить с какого конца следует раскручивать FOR
- В общем случае эффективная раскрутка FOR является сложной задачей
- На практике для численных алгоритмов существуют частные эффективные алгоритмы раскрутки FOR

Определение ФА (продолжение)

```
while x[i-1]>0, i=1..out N {
        cf a[i]: func1(in: i; out: x[i]);
}
```



while — это *структурированный ФВ*
Входные параметры:
1. выражение для первого значения параметра цикла — счётчика
2. значение выражения для текущего значения счётчика
При исполнении ФВ раскрывается на одну копию тела с заданным значением счётчика и аналогичный оператор WHILE, где начальное значение счётчика на единицу больше предыдущего

Определение ФА (продолжение)

```
sub A(in: x; out: y)
{
        df z;
        cf a: func1(in: x; out: z);
        cf b: func1(in: z; out: y);
}
...
cf a: A(in: x[0], out: x[1]);
```

- Фрагментированные подпрограммы — это структурированные ФВ.
- Входные и выходные значения определяются аналогично атомарным ФВ
- Весь ФА — это тоже подпрограмма (аналог main в C/C++)

Фрагменты данных

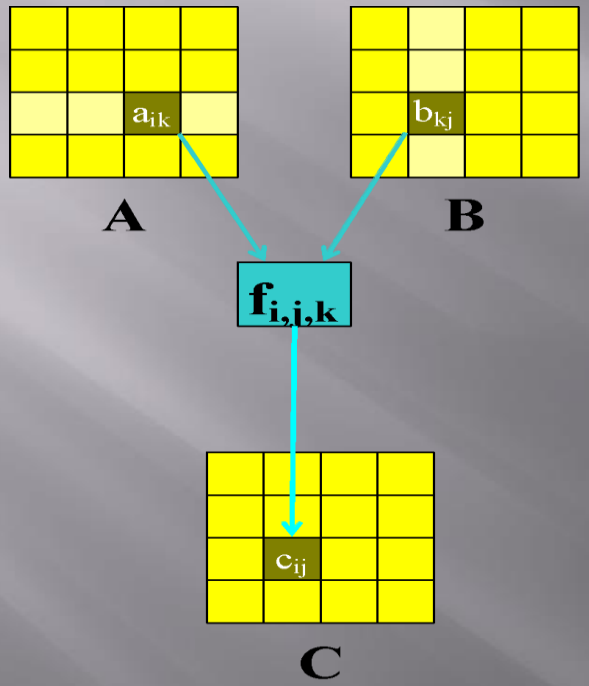
Описание фрагментов данных, фактически, является избыточным и определяется их употреблением
Декларирование ФД служит целям проверки на ошибки
Есть реализационные особенности, связанные с определением ФД, но они не принципиальные и выходят за рамки материала лекции

Особенности ФА

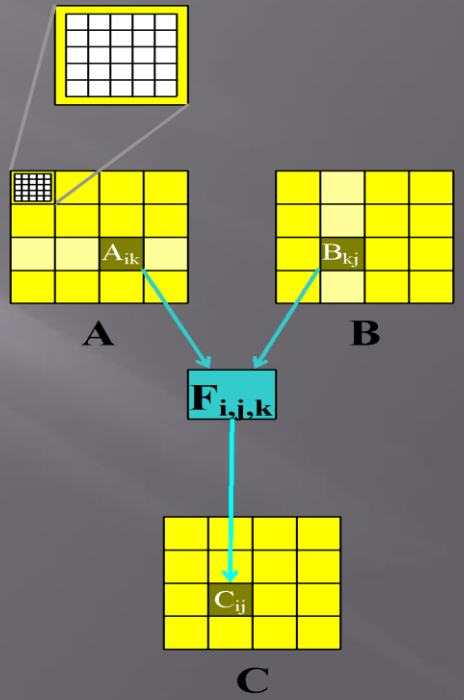
- Сериализуемые ограниченные по размеру ФД
- Ограниченные по времени фрагменты вычислений без побочных эффектов
- Единственность присваивания ФД
- Крупная зернистость ФА

Пример ФА: умножение матриц

Исходный алгоритм



Фрагментированный алгоритм



Полезные свойства ФА

- Высокая переносимость
- Ориентация на автоматизацию исполнения
  - миграция фрагментов
  - сохранение контрольных точек
  - различное управление и распределение ресурсов
  - контролируемая гранулярность
- Масштабируемость
- Явный параллелизм

Слабые стороные ФА

- Отсутствие привязки к конкретным ресурсам
- Отсутствие императивного управления
- Единственность присваивания ФД

Пример LuNA-программы 1

basic1.fa 99 Bytes

```
1  /*
2   Hello world example.
3  */
4
5  import c_helloworld() as hello_world;
6
7  sub main()
8  {
9          hello_world();
10 }
```

ucodes.cpp 81 Bytes

```
1  #include <cstdio>
2
3  extern "C"
4  void c_helloworld() {
5          printf("Hello world!\n");
6  }
```

Пример LuNA-программы 2

basic2.fa 243 Bytes

```
1  /*
2   Initialization of data fragments and data dependencies.
3  */
4
5  import c_init(int, name) as init;
6  import c_print(value) as print;
7  import c_iprint(int) as iprint;
8
9  sub main()
10 {
11         df x;
12
13         cf d: print(x);
14
15         cf b: init(7, x);
16
17         cf a: iprint(x);
18 }
```

Пример LuNA-программы 2

ucodes.cpp 362 Bytes

```cpp
#include <cstdio>
#include "ucenv/ucenv.h"

extern "C" {

void c_init(int val, OutputDF &df) {
        df.setValue(val);
        printf("c_init: %d --> %s, size: %d\n", val, df.getCName(), (int)df.getSize());
}

void c_iprint(int val) {
        printf("c_iprint %d\n", val);
}

void c_print(const InputDF &df) {
        printf("c_print %s is %d\n", df.getCName(), df.getValue<int>());
}

}
```

Исполнение ФА на мультикомпьютере

- ФВ и ФД — реализуются объектами, расположенными на различных узлах (способ реализации рассматривается позже)
- Исходно один ФВ (“main”) располагается на одном из узлов
- В дальнейшем происходит исполнение/раскрытие ФВ по готовности входных ФД
- Система обеспечивает миграцию ФД и ФВ по узлам для выполнения всех ФВ

Требования к реализации ФА

Необходимо: обеспечить доставку всех входных ФД ко всем ФВ. Эта задача не может быть невыполнена

Важно (но не необходимо): обеспечить нефункциональные свойства. Эта задача может быть выполнена лучше или хуже. В частности, система старается:

— Уменьшить время выполнения программы
— Увеличить равномерность нагрузки на вычислительные узлы
— Экономить сетевой траффик
— Своевременно поставлять ФД к ФВ, их потребляющим

(*)

Основные компоненты системы LuNA

- Компилятор LuNA:
  - Синтаксис языка LuNA
  - Транслятор LuNA —> внутреннее JSON-представление ФА
    - Тривиальное преобразование из одного синтаксиса в другой
  - Оптимизации и проверки на ошибки (JSON—>JSON)
  - API для определения пользовательских фрагментов кода (компилируется традиционным компилятором)
- Исполнительная система LuNA (далее)
- Дополнительные компоненты (профилировщик, визуализатор, ...)

Коммуникационный менеджер

Модуль, предназначенный для осуществления коммуникаций между вычислительными узлами

Использует MPI для фактической передачи сообщений (можно сказать, что он является “обёрткой” над MPI). Коммуникационная библиотека может быть заменена на другую без изменения остальной системы

Функционирует в отдельном потоке, фактически — непрерывно ждёт сообщений с помощью MPI_Probe

Отправка сообщений осуществляется асинхронно

Менеджер фрагментов данных (МФД)

- Фрагмент данных — это пара (указатель, размер)
- МФД в пределах узла — это коллекция таких пар с интерфейсом, позволяющим добавлять или удалять ФД
- Передача ФД по сети осуществляется как передача обычных данных. В отправляемый пакет упаковывается идентификатор ФД и его содержание (блок памяти)
- Для отправки ФД по сети используется коммуникационный менеджер

Доставка ФД

Одна из основных функций МФД — это доставка ФД по требованию

1. Некоторый узел посылает запрос на ФД
2. Запрос пересылается на нужный узел по маршруту (модуль PathFinder)
    a. Запрос — это сообщение, содержащее имя запрашиваемого ФД и номер запросившего узла
3. На нужном узле запрос ожидает ФД (если тот ещё не поступил)
4. Затем копия ФД отправляется на запросивший узел

Реализация ФА

Исполнительной системе в разных ситуациях нужна информация о ФА (например, для определения входных ФД для ФВ)

Существует сервисный класс, который считывает JSON-файл и затем может давать информацию о нём (какие есть подпрограммы, какие у них параметры, что находится в теле операторов и т.д.)

Менеджер ФВ (МФВ)

- ФВ реализуется как идентификатор соответствующего оператора в описании ФА
- По этому идентификатору можно определить какие входы и выходы есть у ФВ, какой именно это оператор (exec, if, for, ...)
- Наличие ФВ на узле означает, что этот ФВ необходимо исполнить (так отслеживается фронт выполнения ФВ)
- В начальный момент на одном узле имеется единственный ФВ — main.

МФВ: Первичная миграция

- При создании ФВ он мигрирует на узел, на котором он будет исполняться (т.к. порождение и исполнение ФВ, вообще говоря, могут происходить на различных узлах)
- Миграция ФВ, фактически, сводится к передачи по сети его идентификатора. Когда коммуникационный менеджер получает сообщение с пометкой “ФВ”, он передаёт это сообщение на обработку МФВ.
- Первичная миграция может проходить по нескольким узлам прежде, чем достигнет своего назначения (определяется через PathFinder)

МФВ: Ожидание входных ФД

- Когда ФВ достиг узла исполнения, он отправляет запросы на входные ФД. Иногда это проходит в несколько этапов (когда в индексных выражениях встречаются ФД)
- МФД осуществляет доставку запрошенных ФД на узел. Полученные ФД прикрепляются к ФВ.
- Процесс продолжается до тех пор, пока все входные ФД для ФВ не окажутся к нему прикрепленными

МФВ: Исполнение в пуле потоков

- Пул потоков (многопоточный портфель задач) содержит готовые к исполнению ФВ (с полным комплектом входных ФД).
- Пул потоков — это стандартный шаблон многопоточного программирования
- ФВ исполняется рабочими потоками по мере их освобождения. Информация о входах и выходах ФВ имеется в описании ФА, на основании которых формируется:
  — вызов к библиотеке пользовательских кодов (*.so файл) с нужными параметрами (для атомарных ФВ)
  — порождение новых ФВ, определённых в теле оператора (для структурированных ФВ)
- По завершению вызова выходные ФД передаются в МФД для хранения

МФВ: Поиск объектов (Path finder)

- Объекты (ФД и ФВ) должны быть распределены по вычислительным узлам. Это распределение может быть задано разными способами, в том числе корректироваться или определяться динамически. Так или иначе, должна быть обеспечена возможность нужный объект найти
- Модуль Path finder по заданному имени объекта выдаёт номер соседнего узла, где следует продолжить поиск, либо сообщает о том, что объект должеен находиться на текущем узле.
- Модуль Path finder должен гарантировать, что местоположение любого объекта будет найдено таким способом

МФВ: упрощённый пример Path finder

Пусть задано статическое распределение ФВ и ФД по узлам в виде функции, которая по имени объекта возвращает номер его узла

Тогда Path finder по имени объекта будет определять номер узла назначения и выдавать номер следующего узла на пути к искомому в соответствии с сетевой топологией

Динамическая балансировка нагрузки

При обнаружении дисбаланса вычислительной нагрузки ФВ при первичной миграции передаются соседним недогруженным вычислительным узлам
В более сложных случаях недогруженные узлы определяются иначе, но в остальном схема такая же

Другой вариант динамической балансировки нагрузки — это job stealing, часто применяющийся для распределённых портфелей задач (не реализован в текущей версии системы LuNA)

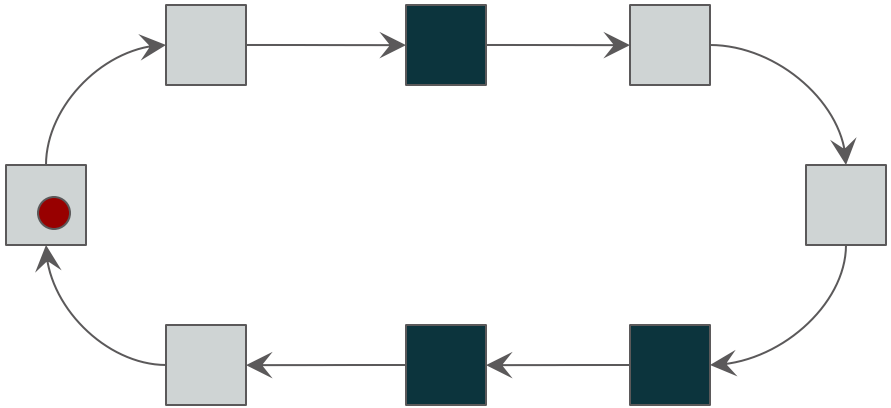
Сборка мусора

Сборка мусора осуществляется по завершению области видимости, в которой объявлен ФД. МФД отслеживает ФД, порождённые внутри области видимости (тела оператора — {...}). МФВ отслеживает ФВ, порождённые и исполненные в области видимости. Когда все ФВ исполнены, то удаляются все ФД в данной области видимости.

Такой алгоритм сборки мусора является малоэффективным, поэтому в системе LuNA существуют другие способы обеспечивать сборку мусора.

Остановка системы

Обнаружение ситуации, когда ни на одном узле уже нет работы, является сложной задачей.
В системе LuNA используется распределённый алгоритм с малыми накладными расходами и локальными взаимодействиями.



Исполнительная система LuNA — итоги

Исполнительная система представляет собой распределённую среду, в которой существуют ФД и ФВ. Система обеспечивает выполнение ФВ в соответствии с семантикой ФА. Рассмотрены все основные составляющие этих процессов:

— порождение, хранение, миграция и исполнение ФВ

— порождение хранение, миграция и удаление ФД

Спасибо за внимание!

Рекомендации

— Зачем нужны рекомендации при реализации ФА?

Рекомендации предназначены для оптимизации реализации ФА путем частичного решения труднорешаемых подзадач на высоком уровне

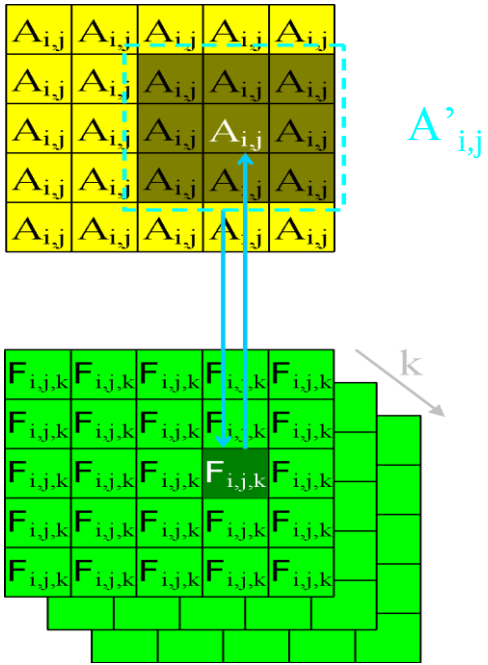
Рекомендации

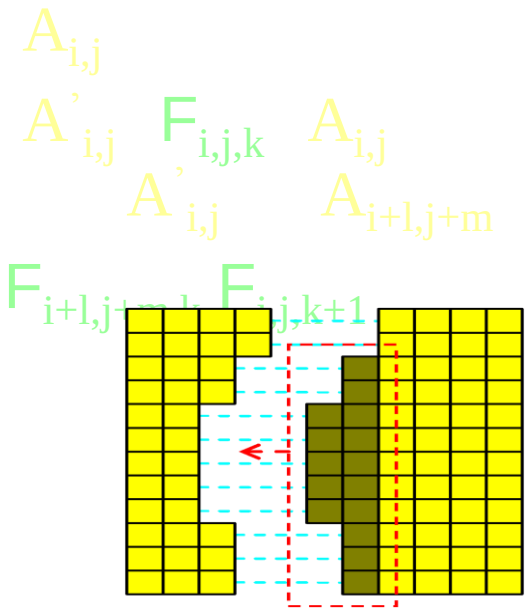
Частичные решения по способу реализации ФА — порядку выполнения ФВ, переиспользования памяти, отображения фрагментов на вычислительные узлы, и т.п.

Примеры рекомендаций

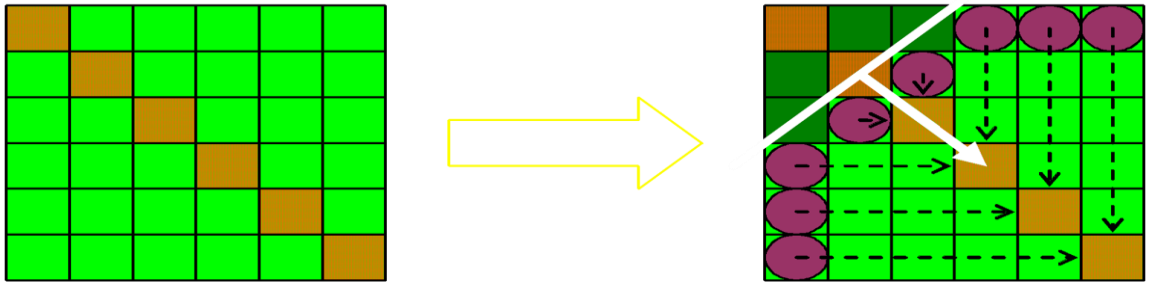
Приоритет выполнения ФВ
Соседство ФД и ФВ
Группы ФВ, образующие “гамаки вычислений”
Совмещение буферов ФД
...

Отношение соседства

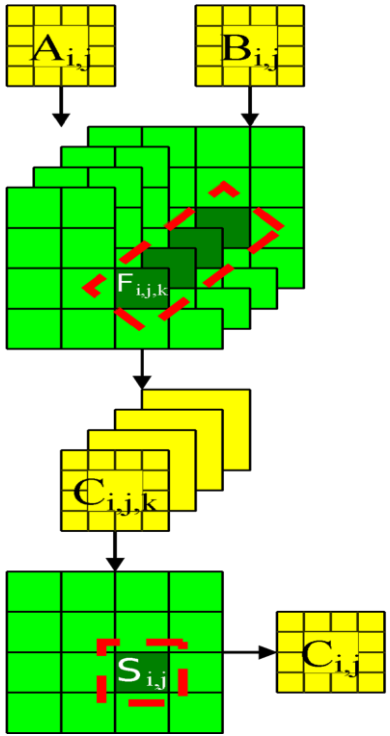




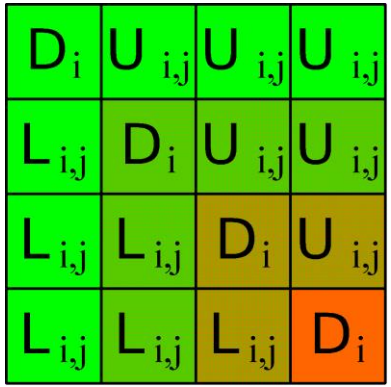
Приоритет ФВ



Группы ФВ



“Вес” ФВ



Рекомендации

ФА содержит все возможные способы его реализации (порядки выполнения ФВ и варианты отображения фрагментов на вычислительные узлы и т.п.)

Рекомендации ограничивают это множество реализаций до небольшого, содержащего преимущественно эффективные способы

Рекомендации — это высокоуровневое средство, их не требуется программировать, а лишь описать

Рекомендации

— Зачем нужны рекомендации при реализации ФА?

Рекомендации предназначены для оптимизации реализации ФА путем частичного решения труднорешаемых подзадач на высоком уровне

Мотивация

Цель доклада — познакомить с внутренней организацией и основными системными алгоритмами системы LuNA

- Понимание того, как устроена система программирования позволяет более эффективно её использовать
- Понимание принципиальной организации позволяет “доверять” системе и вкладываться в неё
- Знание внутренней организации системы необходимо для использования её продвинутых средств
- Материал лекции отражает особенности программирования на суперкомпьютерах вообще

Фрагментированный алгоритм

Понятие ФА определено, в модели больше нет ничего другого

Заключение

Рассмотрены с точки зрения реализации все ключевые компоненты системы программирования LuNA