Алгоритм генерации управляющих программ для оптимизации исполнения фрагментированных программ в системе LuNA

Докладчик: Ткачёва А.А.

12.04.2017

Введение

Система фрагментированного программирования LuNA, предназначена для автоматизации процесса реализации больших задач численного моделирования на суперкомпьютере.

Проблема

Динамическое управление вычислениями в системе LuNA является причиной больших накладных расходов, в то же время нельзя полностью его исключить, так как от параллельной программы требуется некоторая гибкость, например, поддержка миграции для обеспечения динамической балансировки нагрузки.

Цель

- Оптимизировать динамическое управление там, где это возможно, заменив его на императивное управление.
- С этой целью:
  - Разработать для языка LuNA средства задания прямого управления и модули, встраиваемые в runtime-систему для их поддержки.

Обзор систем и языков параллельного программирования

SMP Superscalar: приоритеты выполнения задач.

PaRSEC: алгоритмы линейной алгебры на плотных матрицах из DPLASMA, приоритеты выполнения задач.

LISP, SISAL: специализированные циклы типа forall.

Charm++

Введен язык Charisma для аннотирования коммуникационной активности задач.

Haskell

- Eden – координационный язык для организации параллельных вычислений.
- Поддержка создания своих скелетонов.
- Системная библиотека поддержки работы с распределенными массивами.

Обзор: результаты

Можно выделить 2 подхода к решению:

- Сузить предметную область
- Ввести средства прямого управления

Фрагментированная программа

Фрагментированная программа (ФП) представляет собой двудольный ориентированный граф, вершинами которого являются множество фрагментов вычислений (ФВ) и фрагментов данных (ФД), а дуги – информационные зависимости между фрагментами.

ФВ реализуется вызовом функции без побочных эффектов для некоторого набора входных ФД.

ФП исполнена, когда все ФВ исполнены. Порядок исполнения ФВ основан только на информационных зависимостях.

Базовый алгоритм исполнения ФП runtime системы LuNA

- Из множества невыполненных ФВ выбирается множество готовых к исполнению ФВ. (Будем говорить, что ФВ готов к исполнению, если значения всех его входных ФД вычислены.)
- Из множества готовых к исполнению ФВ выбирается один, несколько, все или ни одного ФВ и назначается на исполнение.
- По мере исполнения ФВ их выходные ФД становятся вычисленными. При этом у некоторых ФВ все входные ФД оказываются вычисленными и такие ФВ переходят во множество готовых к исполнению ФВ.
- Исполнившиеся ФВ исключаются из множества невыполненных ФВ.
- Пункты 2-3 повторяются, пока множество готовых к исполнению ФВ не окажется пустым и ни один ФВ не исполняется.

Runtime-система LuNA

Runtime-система LuNA является полу-интерпретатором ФП. В частности при исполнении ФП она осуществляет следующие функции:

- выделение ресурсов вычислителя для ФВ и ФД;
- выбор из множества всех ФВ множество ФВ готовых к исполнению;
- планирование вычислений (какой ФВ из множества готовых к исполнению ФВ исполнить: один, все или не одного и т.д.);
- обеспечение динамических свойств ФП (например, динамическая балансировка нагрузки).

Виды накладных расходов

1. Накладные расходы на организацию вычислений внутри узла
2. Накладные расходы на организацию вычислений между узлами
3. Накладные расходы на коммуникации

Такие накладные расходы характерны для параллельных программ, написанных для задач численного моделирования.

Предыстория (1): фреймворк LuNAFW

Для оптимизации исполнения ФП в распределенной памяти был разработан фреймворк LuNAFW, в котором управление вычислениями задается с помощью управляющих программ.

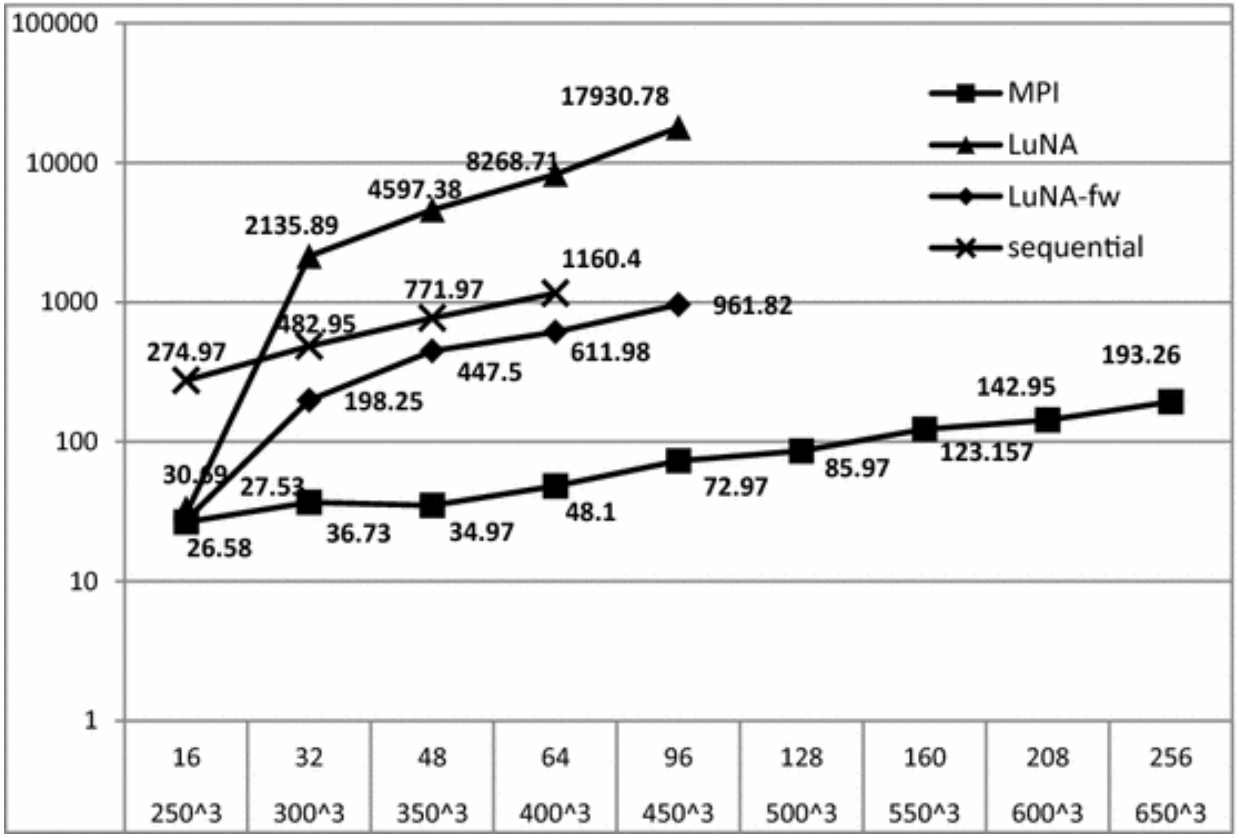
Управляющая программа пишется для конкретной ФП, и в ней заложены частичные решения о распределении ресурсов и порядке исполнения операций.

Предыстория (2)

На LuNAFW были реализованы следующие задачи:

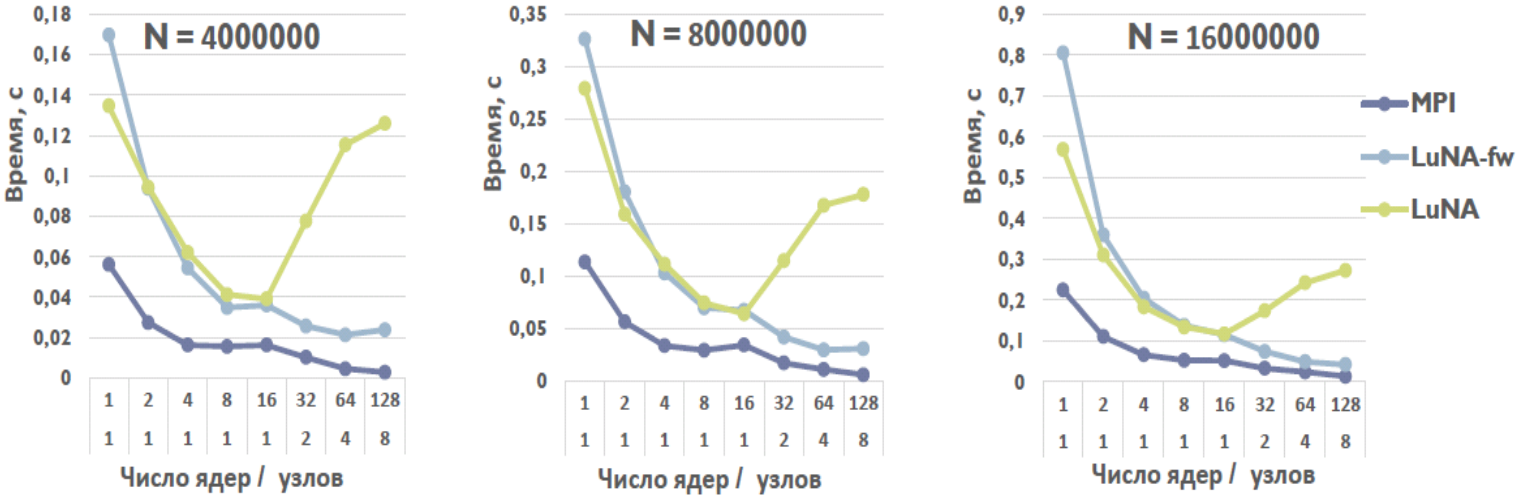
- Задача решение краевой задачи фильтрации трехфазной жидкости (нефть-вода-газ)
- Метод IADE-RB-CG

Задача решение краевой задачи фильтрации трехфазной жидкости (нефть-вода-газ), mvs10p



Тестирование:
Сравнение различных реализаций

IADE метод



- Выводы
  - В пределах узла варианты LuNA и LuNA-fw имеют сравнимую производительность
  - При использовании более одного узла производительность варианта LuNA падает
  - Вариант MPI обгоняет LuNA-fw примерно в 2 раза

Цель работы

Разработать алгоритм генерации управляющих программ для оптимизации исполнения ФП в распределенной памяти.

Для этого для подпрограмм из ФП частного вида генерировать управляющую программу, в которой заложены частные решения о распределении ресурсов и порядке исполнения операций.

Отступление

У каждого ФВ и ФД есть уникальный идентификатор.

Идентификатор может иметь:

- Атомарную форму (a, c, p)
- Индексную форму (a[1], c[i][j])

Вход алгоритма генерации (1)

Фрагментированная подпрограмма:

- Описание ФВ
- Множество ФВ, описанных через языковую конструкцию for
- Множество идентификаторов выходных ФД

Вход алгоритма генерации (2)

Фрагментированная подпрограмма:

- Описание ФВ
  - Идентификатор ФВ
  - Множество идентификаторов входных ФД
  - Множество идентификаторов выходных ФД
  - Имя функции без побочных эффектов
  - Множество идентификаторов ФД, которые следует уничтожить после завершения исполнения ФВ (*опциальный параметр)*
- Множество ФВ, описанных через языковую конструкцию for
- Множество идентификаторов выходных ФД

Вход алгоритма генерации (3)

Фрагментированная подпрограмма:

- Описание ФВ
  - …
- Множество ФВ, описанных через языковую конструкцию for
  - Переменная счетчика цикла
  - Нижнее граничное значение (должно быть целым числом)
  - Верхнее граничное значение (должно быть целым числом)
  - Множество ФВ или множества ФВ описанных через for
- Множество идентификаторов выходных ФД

Требования к входным данным

- Информационные зависимости между ФВ можно проанализировать на стадии компиляции.
- Индексная форма идентификатора может состоять из:
  - Целочисленных констант (a[0])
  - Переменных счетчика цикла (a[i])
  - Выражений вида +/- константа (a[i+1])

Этапы работы алгоритма генерации:

1. Конвертер – конвертация ФП из потоковой модели вычислений в модель типа event-driven
2. Генератор – генерация управляющий программы из выходных данных Конвертера, учитывая распределенное исполнение ФП.

Конвертер

Входные данные для конвектора такие же, как и для алгоритма генерации. Выходные данные Конвертера состоят из:

- *Init* - список ФВ идентификаторов, у которых нет входных ФД.
- *B* - список ФВ идентификаторов, у которых нет выходных ФД.
- *Out* - список выходных ФД ФП.
- Словарь *GarCol*:
  - *Ключ* - ФВ идентификатор.
  - Значение - список идентификаторов ФД, которые следует уничтожить после завершения исполнения ФВ с идентификатором *ключ*. Если идентификатор ФВ или ФД имеет индексную форму, то также записываются границы использования индексов.
- Словарь *DAG*:
  - *Ключ* - ФД идентификатор.
  - Значение - список идентификаторов ФВ, для которых *ключ* – входной ФД. Если идентификатор ФВ или ФД имеет индексную форму, то также записываются границы использования индексов.

Алгоритм работы Конвертер

В случае, когда ФД идентификатор имеет атомарную форму работа конвектора очевидна.

Если ФД идентификатор имеет индексную форму, возможна обработка 3 типов индексов:

- Если индекс – *целочисленная константа*, то она заменяется на новую переменную и значение переменной запоминается – это граница использования этого индекса.
- Если индекс – *переменная счетчика цикла*, то для однородности она заменяется на новую переменную и границы изменения индексов обновляются.
- Если индекс – *выражение вида переменная +/- константа*, то вводится новая переменная равная выражению. Т.о. старая переменная заменяется на *выражение новая переменная -/+ константа* в описании ФВ. Границы использования новой переменной равны границам старой переменной +/- константа.

Имя новой переменной выбирается исходя из порядка индекса в индексной форме.

Генератор

Выходные данные Конвертера – это входные данные для Генератора. Выходные данные Генератора – это управляющая программа, которая представляет собой C++ класс для исполнения в распределенной и общей памяти.

Следующие функции в управляющей программе должны быть определены:

- **onInit ()** – функция начального значения для начальной инициализации.
- **onComputed (df_id)** – функция, вызываемая после того как каждый ФД с идентификатором df_id вычислен.
- **onReceived (df_id)** – функция, вызываемая после того как каждый ФД с идентификатором df_id получен от другого узла.
- **onCfFinished (cf_id)** – функция, вызываемая после того как каждый ФВ с идентификатором cf_id завершил исполнения.

Действия, поддерживаемые LuNAFW

Внутри выше описанных функций могут быть вызваны следующие функции (действия), поддерживаемые фреймворком LuNAFW:

- **startCF** (CF disctiption) – запустить исполнение ФВ.
- **checkCF** (CF disctiption) – если все ФД доступны, то вызвать функцию **startCF** для CF.
- **destroyDF**(df_id) – удалить ФД с идентификатором df_id.
- **sendDF** (df_id, rank) - послать ФД с идентификатором df_id на процесс rank.
- **exit()** – завершить работу фреймворка.
- **getRank**(identifier) - функция, выдающая номер процесса, на который ФВ распределен.

Работа Генератора (1)

1. Чтобы сгенерировать функцию **onInit()** используется список *Init.* Для каждого ФВ из него проверяется распределен ли он на этот узел, и если да, то для ФВ вызывается функция **startCF.**
2. Чтобы сгенерировать функцию **onCfFinished** (*cf_id*) используются словарь *GarCol* и список *B*. Проверяются следующие условия:
   - Если в *GarCol* есть ключ *cf_id,* то для значения с этим ключом применяется функция **destroyDF.**
   - Если список *B* содержит *cf_id,* то это фиксируется.

Работа Генератора (2)

3. Чтобы сгенерировать функцию **onComputed** (df_id) и **onReceived** (df_id) используется словарь *DAG* и список *Out*. Если в *DAG* есть ключ df_id, то:
   - В обоих случаях: Для значения с ключом *df_id* применяется функция **checkCF,** если ФВ распределен на этот узел. Если список *Out* содержит *df_id*, то это фиксируется.
   - В случае функции **onComputed**: Если ФВ распределен на другой процесс, вызывается функция **sendDF** для *df_id* и номера процесса, куда ФВ распределен. Если ФД требуется нескольким ФВ, то поддержана оптимизация посылать ФД только один раз на каждый процесс.

Управляющая программа должна завершить исполнение, если все ФВ из списка *B* завершили исполнение, и все ФД из списка *Out* вычислены. Это также поддерживается Генератором и вызывается функция **exit**.

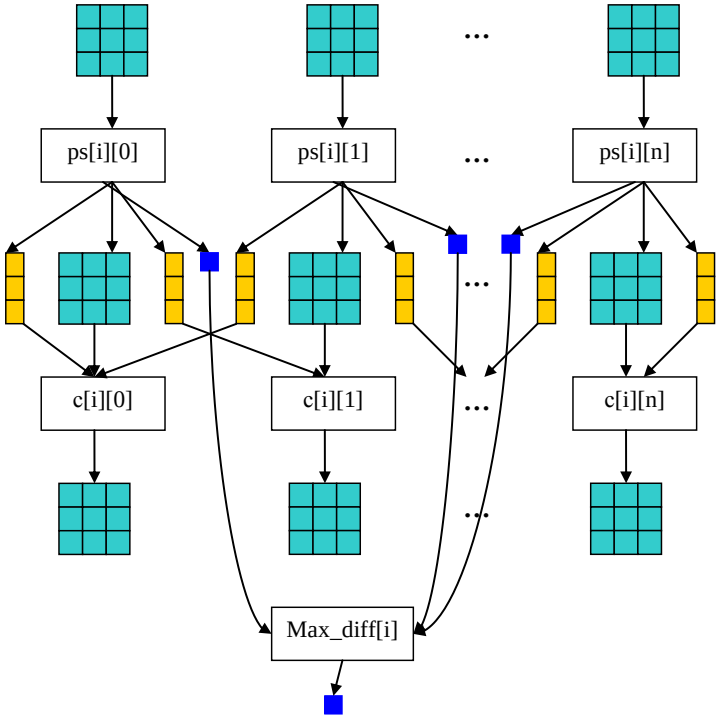
Порядок исполнения ФВ с помощью сгенерированной управляющей программы, описанным алгоритмом, не противоречит информационным зависимостям исходной ФП.

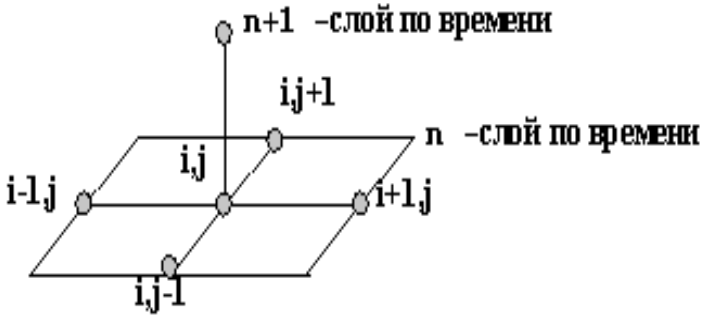
Предполагается, что представленный алгоритм можно будет использовать для итерационных задач численного моделирования, редукции, умножении матрицы на вектор и т.д.

Преимущества

1. В LuNAFW нет менеджера ФД (ФД посылается на нужный узел без необходимости дополнительных запросов)
2. Нет проблемы неконтролируемой раскрутки данных
3. Т.к. применяем для случая, когда динамика не нужна, то не необходимости поддерживать локальный алгоритмы распределения ресурсов, а можно посылать сразу на нужный узел.

Решение уравнения Пуассона явным методом (i-итерация 1D декомпозиция)





```
import c_change_border(int #id, value #dim, value #border1,value
    #border2, value #F, name, int)  as change_border;
import c_ps(int #id, value #sft, value #dim, value #F0, name #FF, name
    #down, name #up, name #mx, int #L) as ps;

sub puasson(int it, name sft, name dim, name F, name maxdif)
{
   df FF, down, up, mx;

   cf ps[0] : ps(0, sft, dim, F[it][0], FF[it][0], up[it][0], up[it][1], mx[it][0][0],
    it) --> (F[it][0]);
   cf ps[$SIZE-1] : ps($SIZE-1, sft, dim, F[it][$SIZE-1], FF[it][$SIZE-1],
    down[it][$SIZE-2], down[it][$SIZE-1], mx[it][0][$SIZE-1], it) -->
    (F[it][$SIZE-1]);

   for i = 1 .. $SIZE - 2
      cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i - 1], up[it][i + 1],
    mx[it][0][i], it) --> (F[it][i]);

   for i = 0 .. $SIZE - 1
      cf border[i] : change_border(i, dim, up[it][i], down[it][i], FF[it][i],
    F[it+1][i], it, 1) --> (FF[it][i]);

}
```

```
onAvailable F[it][i]{
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], up[it][i], up[it][i+1], mx[it][0][i], it)  i = 0;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i-1], down[it][i], mx[it][0][i], it)  i = $SIZE-1;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i - 1], up[it][i + 1], mx[it][0][i], it) i = 1 .. $SIZE-2;
}
onAvailable up[it][i]
   checkCF cf border[i] : change_border(i, dim, up[it][i], down[it][i], FF[it][i], F[it+1][i], it, 1) i = 0 .. $SIZE-1;
onAvailable down[it][i]
   checkCF cf border[i] : change_border(i, dim, up[it][i], down[it][i], FF[it][i], F[it+1][i], it, 1) i = 0 .. $SIZE-1;
onAvailable FF[it][i]
   checkCF cf border[i] : change_border(i, dim, up[it][i], down[it][i], FF[it][i], F[it+1][i], it, 1) i = 0 .. $SIZE-1;

onAvailable dim {
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], up[it][i], up[it][i+1], mx[it][0][i], it)  i = 0;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i-1], down[it][i], mx[it][0][i], it)  i = $SIZE-1;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i - 1], up[it][i + 1], mx[it][0][i], it) i = 1 .. $SIZE-2;
   checkCF cf border[i] : change_border(i, dim, up[it][i], down[it][i], FF[it][i], F[it+1][i], it, 1) i = 0 .. $SIZE-1;

}
onAvailable sft {
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], up[it][i], up[it][i+1], mx[it][0][i], it)  i = 0;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i-1], down[it][i], mx[it][0][i], it)  i = $SIZE-1;
   checkCF cf ps[i] : ps(i, sft, dim, F[it][i], FF[it][i], down[it][i - 1], up[it][i + 1], mx[it][0][i], it) i = 1 .. $SIZE - 2;
}
onFinished border[i]{
   destroyDF FF[it][i];
   destroyDF up[it][i];
   destroyDF down[it][i];
}
onFinished ps[i]{
   destroyDF F[it][i] if(i == 0);
   destroyDF F[it][i] if(i == $SIZE-1);
   destroyDF F[it][i] if (i >= 1 && i =< $SIZE-2);
}
```

Исследование производительности

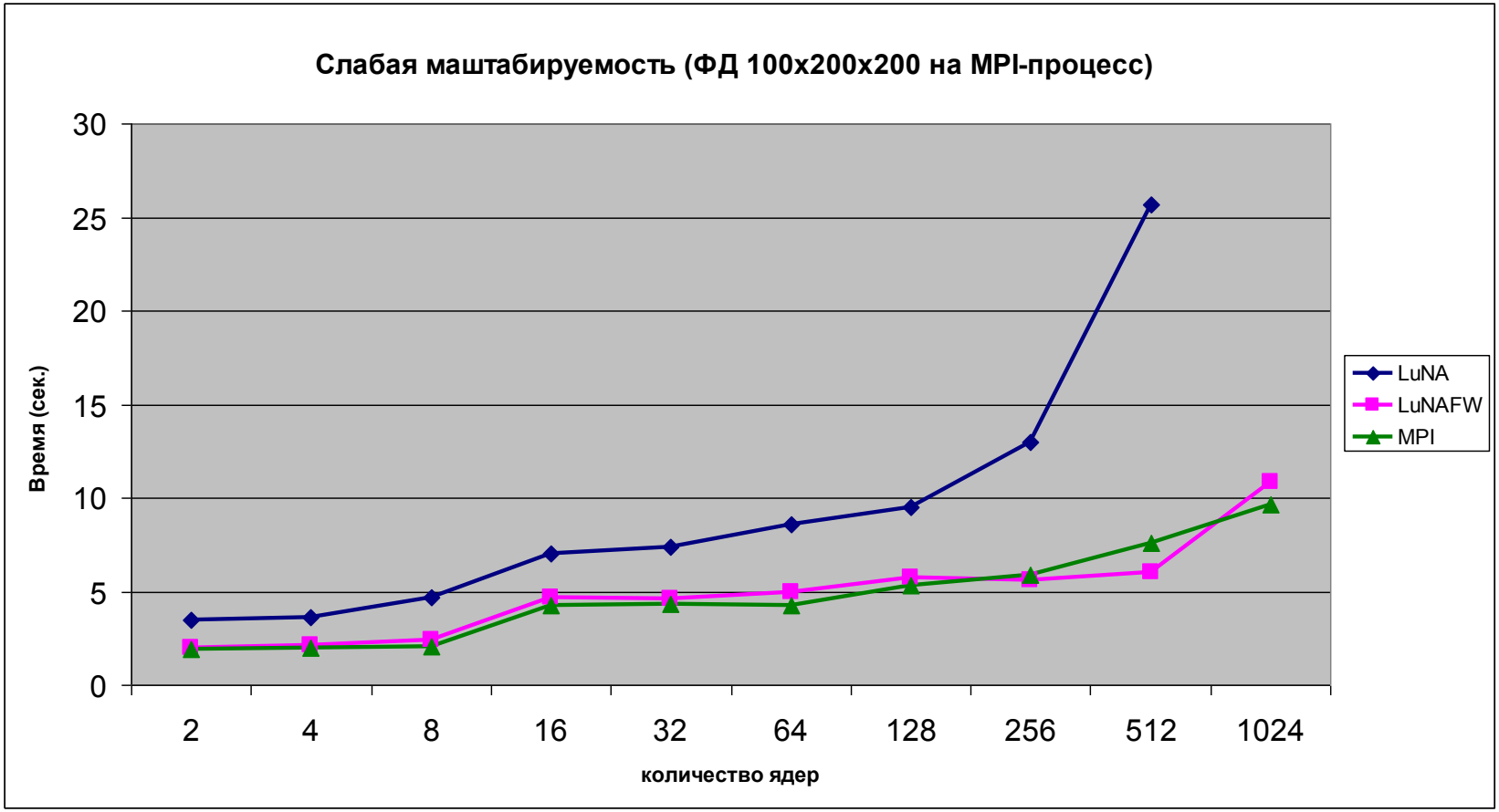
Вычислитель: mvs10p (16 ядер на узел)

- 1 поток в системе LuNA.
- 1 MPI процесс на ядро.
- 1 ФВ на 1 MPI процесс.

Параметры задачи:

Количество итераций 20.

Тесты производительности



Результаты

Автоматически сгенерированная управляющая программа позволила улучшить производительность исполнения ФП на 40% по сравнению с базовым алгоритмом исполнения ФП run-time системы LuNA.

Выводы

- Был разработан алгоритм генерации управляющих программ для оптимизации исполнения ФП частного вида.
- Проведено сравнительное тестирование производительность исполнения ФП предлагаемым подходом по сравнению с базовым алгоритмом исполнения runtime системы LuNA и аналогичной реализации на MPI на задаче решения уравнения Пуассона явным методом одномерная декомпозиция.
- Сгенерированная управляющая программа позволяет улучшить время исполнения на 40% и получить сравнимую с MPI производительность.

Дальнейшие планы

- Расширить применимость алгоритма генерации управляющих программ
- Провести исследование производительности для автоматически сгенерированных управляющих программах на других задачах.

Спасибо за внимание!