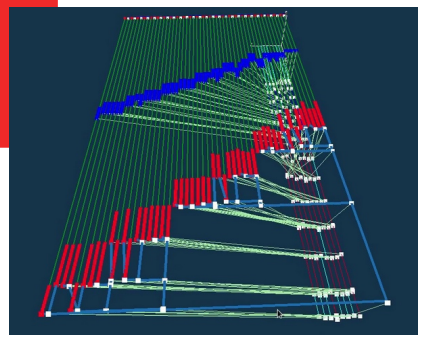
Национальный суперкомпьютерный форум
Россия, Переславль-Залесский,
ИПС имени А.К. Айламазяна РАН,
28 ноября – 01 декабря 2023 года

Гибридное планирование графов задач



Павел Васёв
Институт математики и механики УрО РАН
имени Н.Н. Красовского, г. Екатеринбург

Сектор компьютерной визуализации

www.cv.imm.uran.ru

План доклада

- Модель графов задач
- Проблема накладных расходов
- Предварительное и гибридное планирование
- Новые возможности по локальной постановке задач
- Высокоуровневые подходы

Параллельное программирование на основе задач

Задача – неблокирующееся конечное вычисление, привязанное к конкретным входам и имеющая адресуемый выход.

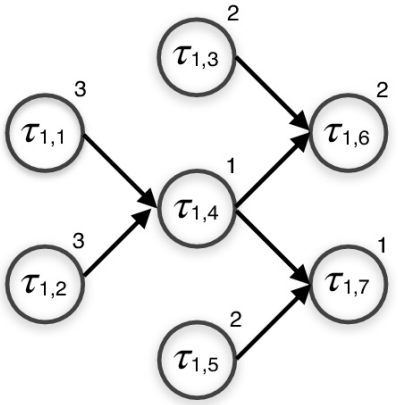
- Задача находится в одном из состояний: сформулирована, запланирована, выполнена, etc.
- Задача выполняется 1 раз.

Идея: представлять параллельную программу как множество задач, связанных по входам и выходам, возможно развивающееся во времени.

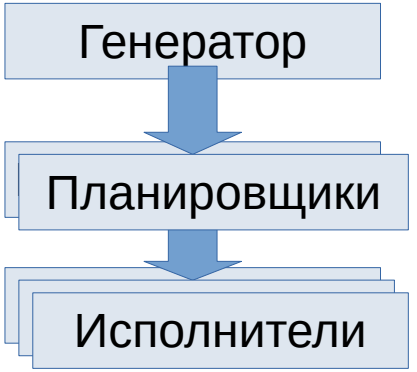
Преимущества:

- Это удобно мыслить.
- Это удобно исполнять.

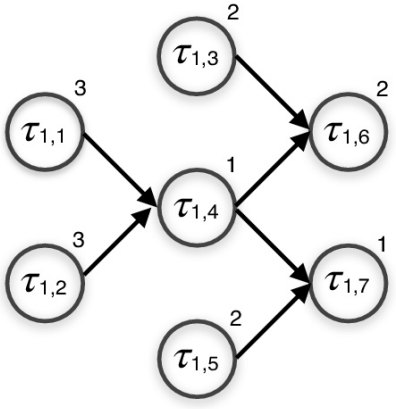
Такое множество называют граф задач.

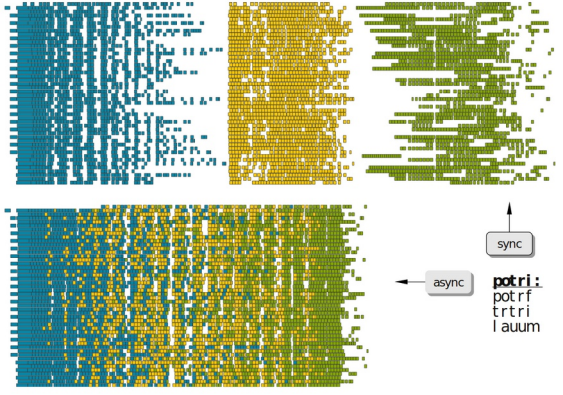


Пример



- Пользователь пишет **генератор**, т.е. алгоритм, который формулирует задачи. Его писать удобно!
- Задачи назначаются на выполнение процессам исполнителей (вручную или автоматически – **планировщиками** системы).
- **Исполнители** решают задачи по очереди, по мере возможности их решения и наличия ресурсов.
- По завершению решения очередной задачи становится возможным решать задачи, которые зависели от результата её вычисления.





Автор изображения Jack Dongarra

История и состояние вопроса

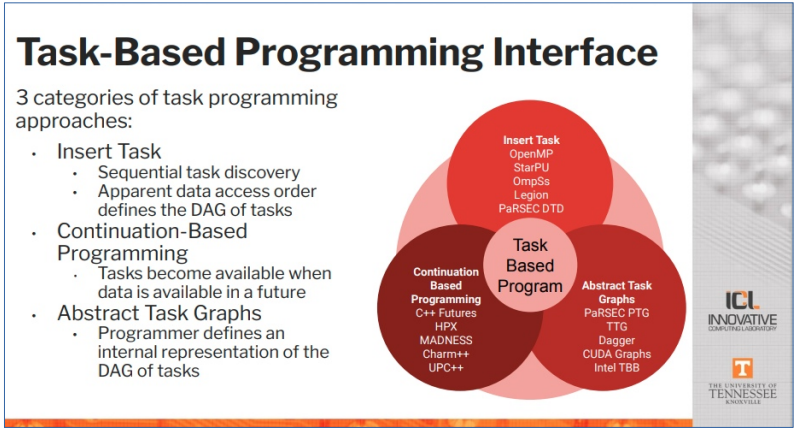
История вопроса см. например книгу Algorithms, Software and Hardware of Parallel Computers // Edited by J. Mildosko and V. E. Kotov. 1984.

- A-система Котов-Нариньяни, 1966 (**асинхронные вычисления**)
- Общая схема Karp-Miller, 1969

Современные англоязычной термины: **(asynchronous) Many-task systems. Task-based runtime systems. Sequental task flow. Parametrized task graph.**

- Dataflow – это другая абстракция!
- Примеры современных систем:

Open TS!, Template Task Graphs, PaRSEC, HPX, Charm++, Uintah, Kokkos, Legion, C++ Sender Library, LuNA, Dask, Flyte, TaskTorrent.

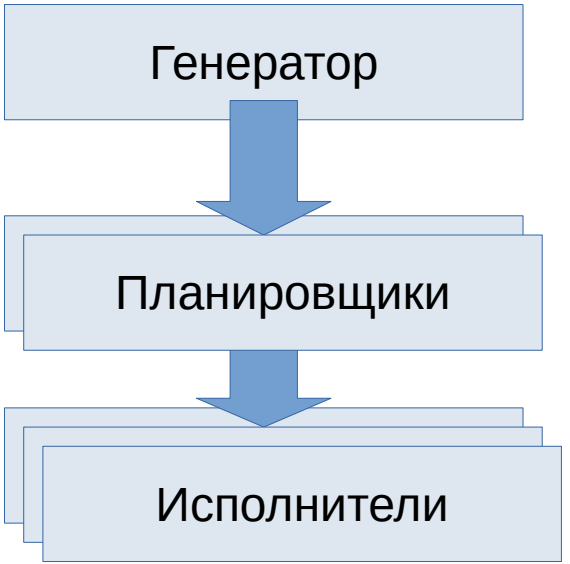


Вопрос эффективности

- Эффективность вычисления!
- Каждая задача **привносит накладные расходы**.
- Поэтому эффективность достигается на задачах относительно высокой гранулярности.
- А для средних и малых задач применяют AMT+X.
- Но у подхода все же находят перспективы!

Накладные расходы задач

- Алгоритм генератора задач.
- Алгоритм планирования задач (назначение исполнителям).
- Передача описаний задач между частями системы.

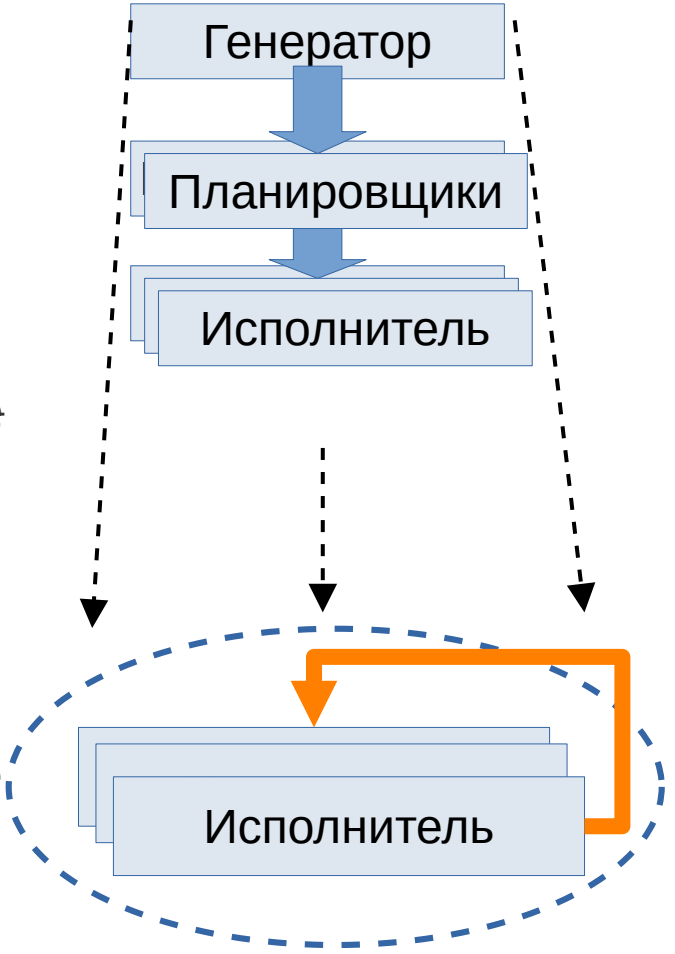


Обращение к внешним системам или к сети это основной источник задержек.

Пример рассмотрения накладных расходов: Decentralized in-order execution of a sequential task-based code for shared-memory architectures. // Charly Castes

Накладные расходы

- ~~Алгоритм генератора задач.~~
- **Переход к шаблонам.**
- ~~Алгоритм планирования задач (назначение исполнителям).~~
- **Ручное / гибридное планирование.**
- ~~Передача описания задачи между частями системы.~~
- **Локализация порождения задач ближе к исполнителям.**



Переход к шаблонам

- Примеры: C++ stdexec, Nvidia graphs

```
task1 = create_task_node( …, input: free-arg )
task2 = create_task_node( … , input: task1 )
for (let i in 1...10^6) p = spawn( task2, p )
```

- Шаблоны позволяют сжать передаваемый “командный” трафик, т.к. передаются системе единожды.
- Шаблоны применимы, когда структура вычисления имеет повторяющиеся участки.

Пользовательское планирование

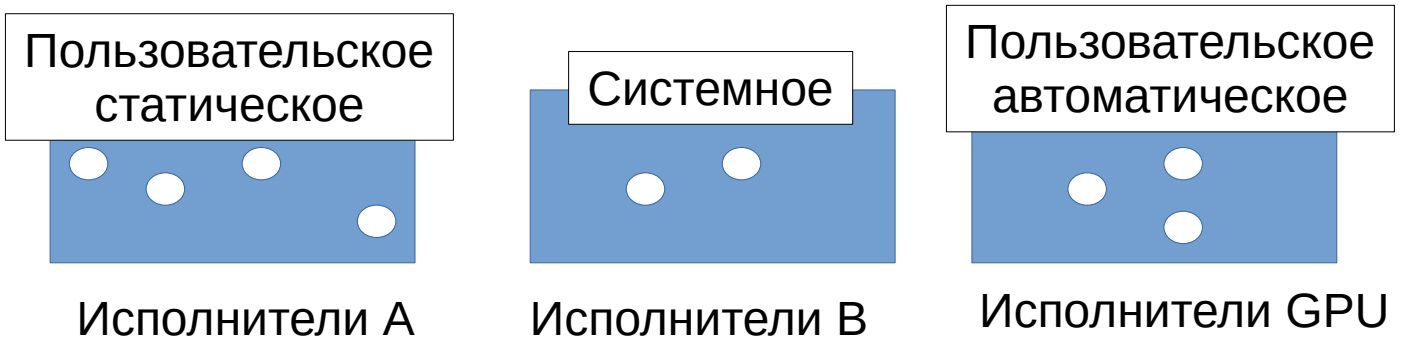
- **Системное планирование не обязательно**.
- Кроме того, оно может работать неоптимально.

- Планирование - ответственность и возможность для пользователя. Он может применить некоторый алгоритмом планирования, или распланировать задачи статически. **В этом есть свобода пользователя**.
- Пользователь знает структуру своей задачи и может её учесть при назначении задач.
- Особенно если есть план размещения данных!

Гибридное планирование

- **Часть** задач пользователь назначает самостоятельно.
  Там где пользователь (или его алгоритмы планирования) имеет представление об оптимальности.
- **Часть** задач назначаются алгоритмами системы. Там где ситуация неясна. Возможно использование каких-либо подсказок (например в шейдерах GPU – будут ли меняться данные в буфере).

- **Вариант**: разделение исполнителей на **подмножества**. (произвольно или по критерию, например по оборудованию).
- Независимое балансирование задач в этих подмножествах: где-то решением пользователя, где-то системой.



Гибридное планирование

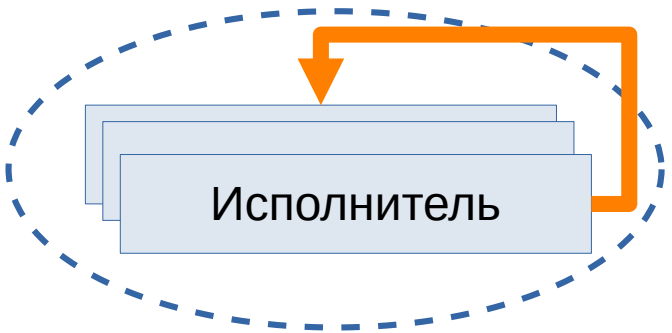
- Таким образом ответ на вопрос, где же следует выполнять задачу, определяется **решением пользователя**.

Это решение можно описать функцией (или процессом):

- Назначить по некоторому алгоритму, возможно статически.
- Отдать на планирование системе.
- Комбинировать оба варианта.
- Применять дополнительные методы управления вычислительным полем и множеством задач.

Локализация порождения задач к исполнителям

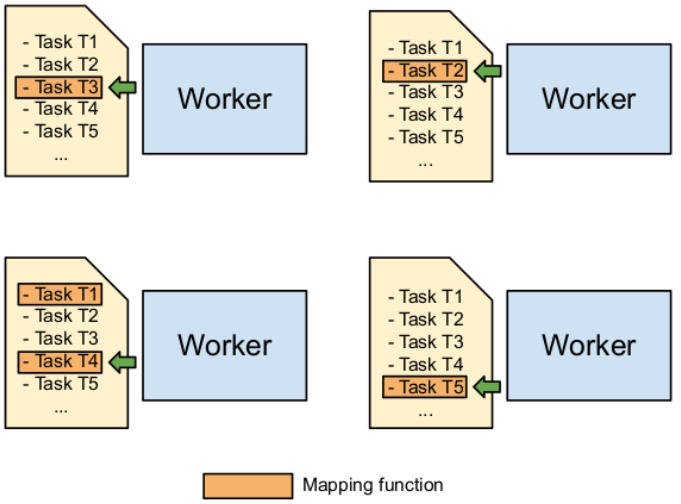
- Алгоритмы генерации и балансировки распределяются ближе к исполнителям.
- В идеале, на каждом исполнителе работает **локальный процесс порождения задач** для этого исполнителя.
- Отметим, что этот подход используется “испокон веков”, например в MPI. Но мы представляем его в более удобной форме для мышления и для эффективного вычисления.



Примеры работ

Глобальная **функция** порождения задач запускается на всех исполнителях, и отсекаются задачи не-этого исполнителя (подразумевается статическое планирование).

Parametrized Task Graph. Граф описывается функцией, которая вызывается системой по мере вычисления других узлов графа.



- Decentralized in-order execution of a sequential task-based code for shared-memory architectures. // Charly Castes, Emmanuel Agullo, Olivier Aumage, Emmanuelle Saillard. 2022.
- Dynamic task discovery in PaRSEC: a data-flow task-based runtime // Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017.
- TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++ // Léopold Cambier, Yizhou Qian, Eric Darve. 2020.

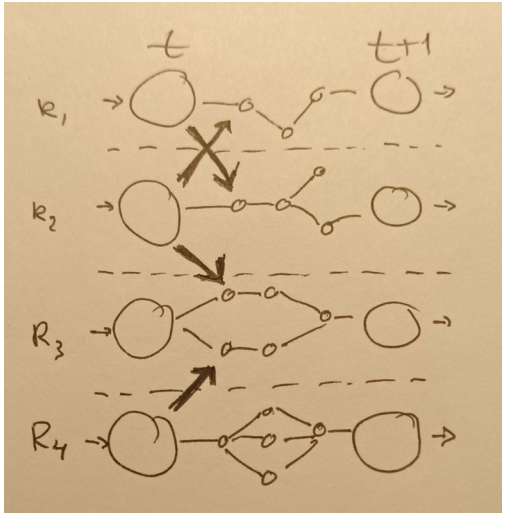
По строкам исполнители, а по столбцам задачи.

Графо-матричный метод

$runner_1$: $task_1$, $task_2$, ….
...
$runner_i$: $task_k$, $task_{k+1}$,...

- Считается, что вычисление итеративно.
- Следовательно, достаточно задать граф задач одной итерации. И повторять его.

- Вычисляется функция порождения графа задач. Подразумевается статическое планирование (т.е. каждой задаче в графе заранее сопоставлен исполнитель, возможно автоматически).
- Результат функции - словарь наборов задач:
  (tasks(runner1), tasks(runner2), ….)
  Для каждого исполнителя runner_i - свои задачи tasks(runner_i).
- Полученный граф рассылается по исполнителям.
- Исполнители запускают только свои задачи.



- Обмен данными между задачами - по каналам. Задача запускается по наличию её входных данных в связанных каналах. И свой результат пишет в канал.

Примитив синхронизации “Канал”

модель publish/subribe с темами сообщений...

- Вводится общее пространство имён
- Можно открыть канал на запись и привязать его к имени
- Можно открыть канал на чтение
- При записи значения рассылаются читающим.
- При подключении нового абонента канал **сообщает** ему последнее записанное значение. Это позволяет подключаться “чуть” позже.

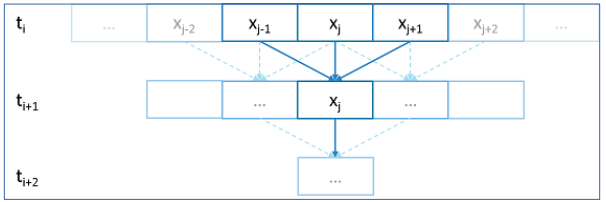
Связь с задачами:

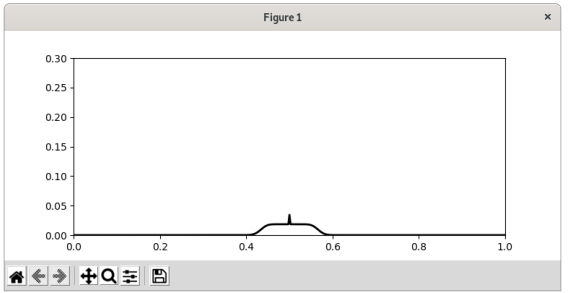
- Можно привязать какой-либо **вход задачи** к каналу.
- Можно указать чтобы **результат задачи** записывался в канал.

```
c = write_channel(“data_305”)
add_task( …, input: c )
add_task( …, output: c )
```

Тонкости: внутри процесса исполнителя данные передаются по указателю.

Пример

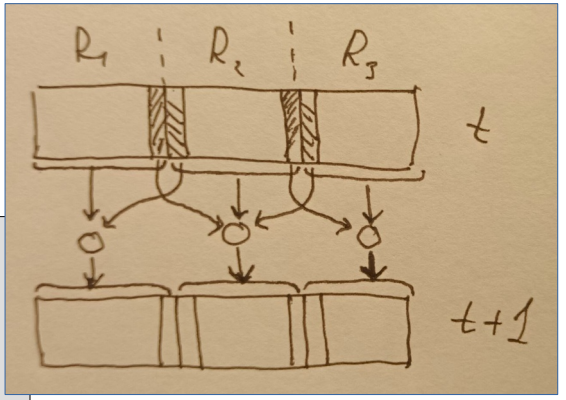




$$f(x_i,t+1) = g(\ f(x_{i-1},t),\ f(x_i,t),\ f(x_{i+1},t)\ )$$

Пусть итеративно считается скалярная функция на отрезке.

Вычислим её параллельно на P исполнителях.



- Создадим P каналов данных data.
- Функция генерации графа задач для итерации:

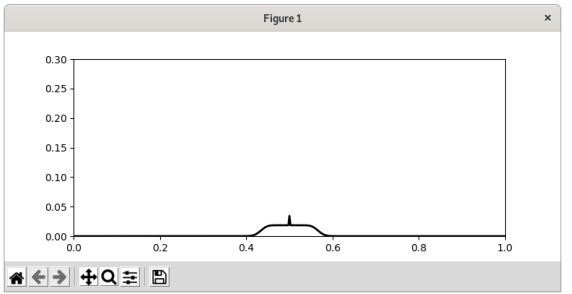
```
function mk_iter( data ) {
  return data.map x,iter =>
      create_task( f, me=reuse(data[i]), left=meta(data[i-1]),
            right=meta(data[i+1]), output = data[i])

}
```

- Каждому исполнителю из этого графа достаётся 1 задача на итерацию.
- На исполнителе i на вход задача берёт данные из каналов data[i-1], data[i], data[i+1]. И пишет на выход в канал data[i].
- Тонкости: reuse повторно использует память. Meta отбрасывает массивы.

Результаты

- Длина отрезка DN = 10^6
- Кол-во итераций: 1000
- Кол-во исполнителей: P = 4



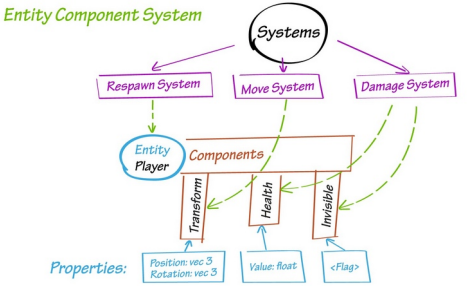
11 с. Последовательный счёт

5 с. Параллельный со статической балансировкой

3.6 с. Параллельный с ручным программированием исполнителей и обменом на каналах (TCP)

3.7 с. Параллельный представленным методом

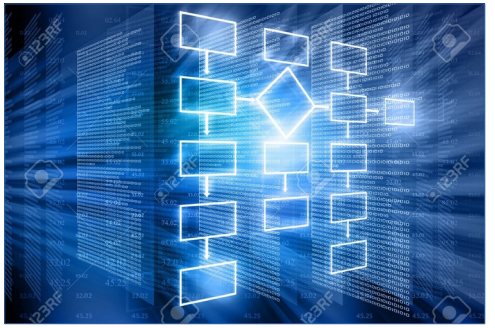
ECS-метод (предварительный)



- Модель программирования компьютерных игр Entity-Component-System (ECS). Игра – это похоже на моделирование.
- Есть сущности (entity). Они состоят из компонент (component). Есть список систем (system) – они запускаются поочередно. Система находит сущности (критерии поиска – наличие компонент, появление компонет, удаление компонент) и обновляет их компоненты.
- Игра строится как цикличный запуск систем.

- Назначим сущности на исполнители.
- Будем запускать системы на исполнителях.
- Каждая система инициирует поиск и обработку комбинаций сущностей. В каждой комбинации есть основная и справочные.
- Но система выполнения находит системам только те основные сущности, которые назначены исполнителю. А справочные – загружает с соседних исполнителей.
- Таким образом обмен данных можно автоматизировать. Надо рассылать те сущности (а точнее их грани – т.е. подмножества компонент) которые используются как справочные.

Высокоуровневые модели



Например:

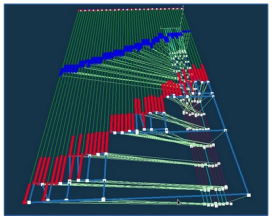
- **Блок-схемы**
- **Функциональное программирование**

iter_func := F(G(prev_iter), H(Q(prev_iter)))

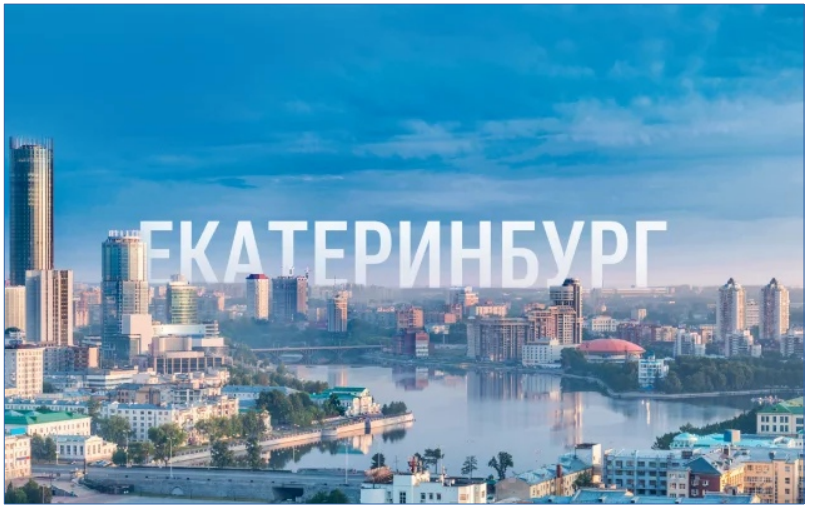
Из них можно построить локальные процессы порождения задач.
Например:

- ФП – функция берет на вход графы и возвращает граф задач (1 итерации), и полученные графы комбинируются.
- Блок-схемы – обходим и получаем граф задач для 1 итерации.

Выводы

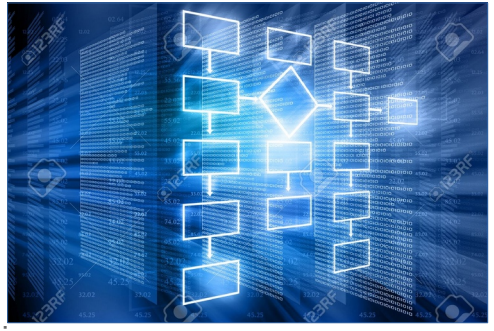


- Необходимо снижать накладные расходы задач.



<u>Направление решения</u>:

- Отказ от 100% динамического планирования.
- Локализация постановки задач на исполнителях.
- Переход к высокоуровневым моделям, на которых такая постановка автоматизируется.



Благодарю за внимание!

Павел Васёв

vasev@imm.uran.ru

Сектор компьютерной визуализации ИММ УрО РАН

https://cv.imm.uran.ru