Московский Государственный Университет им. М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики

Конвейерная модель представления параллельных программ

Докладчик:
Кривов М.А.
Участники:
Притула М.Н.

Владимир, 2009

Московский Государственный Университет им. М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики

Конвейерная модель представления параллельных программ

Докладчик:
Кривов М.А.
Участники:
Притула М.Н.

Владимир, 2009

Московский Государственный Университет им. М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики

Конвейерная библиотека создания параллельных программ *ttgLib*

Докладчик:
Кривов М.А.
Участники:
Притула М.Н.

Владимир, 2009

План выступления

Описание библиотеки

Примеры решения CPU-задач

Привязки к гибридным архитектурам

Дальнейшее развитие

Заключение

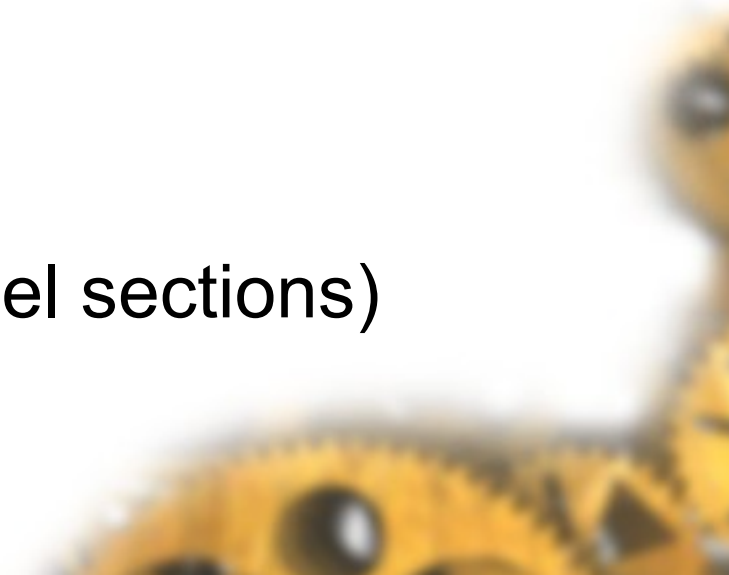
Подходы к параллельному программированию на SMP

Большинство современных подходов сводятся к использованию следующих примитивов:

- Parallel for/while
- Parallel reduce
- Tasks
- Pipeline
- Threads

Примеры:

- OpenMP (parallel for, tasks, parallel sections)
- Intel Threading Building Blocks
- Microsoft Parallel Extensions



Основная идея библиотеки ttgLib

Примитив «Конвейер» обладает следующими достоинствами:

- Возможность реализовать гибкую компонентную модель
- Однотипность обработки данных (позволяет использовать статистику для планирования)
- Использование более «понятной» для пользователя модели потоков данных

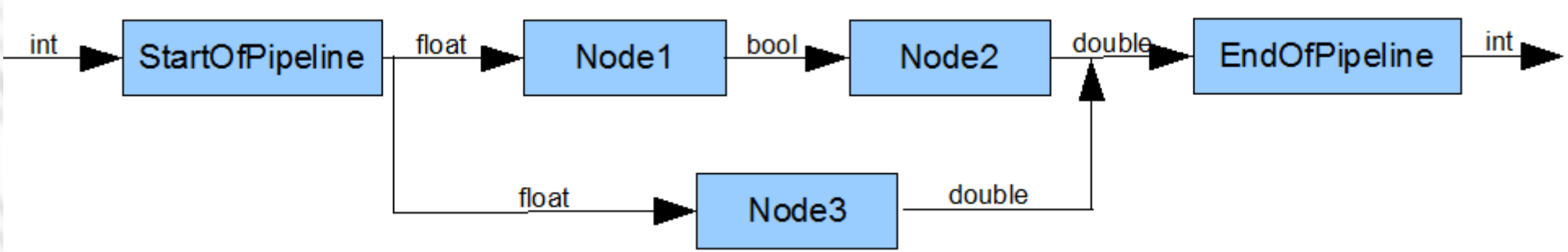
Идея работы: создать реализацию конвейера, которая:

- Реализует озвученные достоинства
- Позволяет *[эффективно]* реализовать любую задачу

Краткое описание библиотеки

Для использования ttgLib необходимо:

- Разбить программу на простые блоки, которые могут получить данные, обработать их и отправить дальше
- Соединить полученные блоки для получения требуемой функциональности



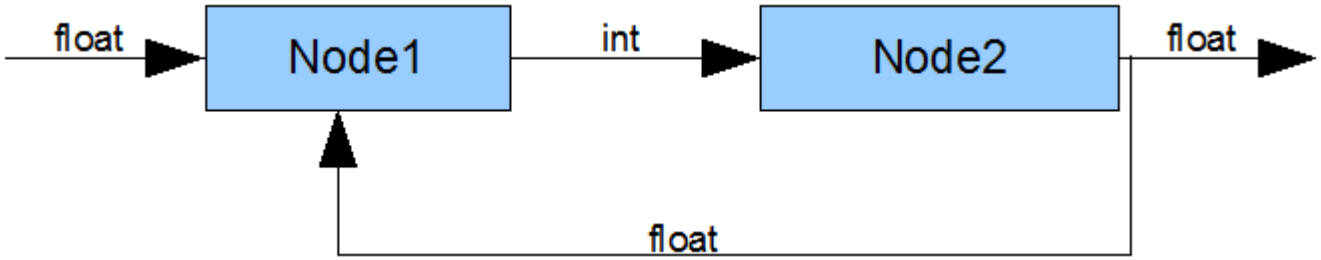
Примечание: Можно провести следующую аналогию: Конвейер — это ориентированный граф, вершинам которого соответствуют блоки-обработчики, а рёбрам — потоки данных

Возможности ttgLib

Поддержка событий конвейера:

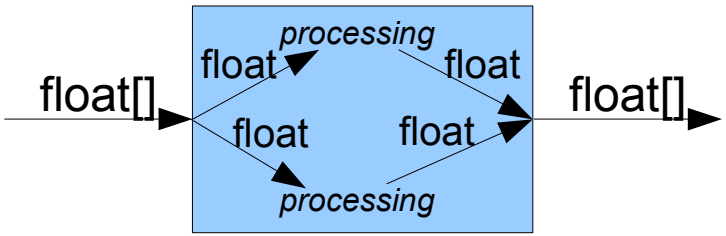
- onBegin — выполнить до начала обработки данных
- onEnd — выполнить после завершения обработки

Возможность использования «петель»:



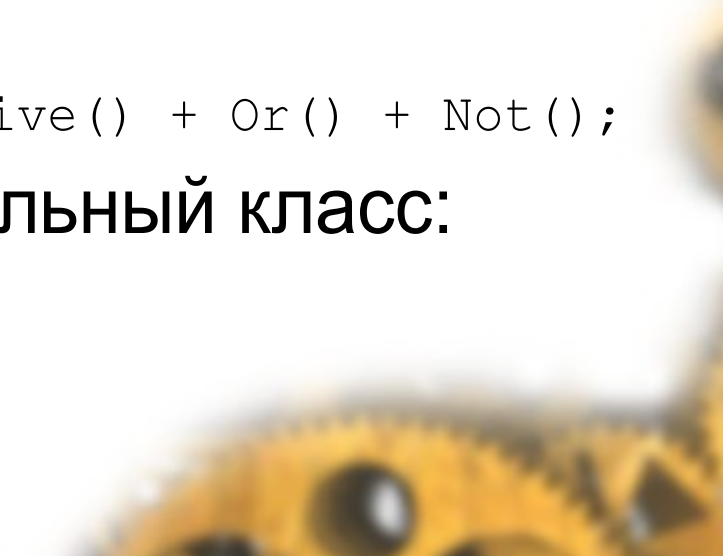
Возможность переупаковки данных (паттерн Scatterer / Gatherer):

Пришедшие данные до некоторых пор «накапливаются», после чего перепаковываются в новый формат и посылаются на обработку.



Пример (проверка на ноль)

- Алгоритм: !(val > 0.0f || val < 0.0f)
- Создание конвейера:
  - Каждое звено наследуется от специального предка:
    - `class IsPositive :public Node<float, bool> {`
    - `  virtual void process(float data)`
    - `  { sendNext(data > 0.0f); }`
    - `};`
  - Через перегруженные операторы + и * производится «склейка» конвейера:
    - `Wrapper n = isPositive() * IsNegative() + Or() + Not();`
- Запуск производится через специальный класс:
  - `Pipeline<float, bool> IsZero(n);`
  - `if (IsZero.start(0.0f))`
    - `printf("It works!");`



План выступления

Описание библиотеки

Примеры решения CPU-задач

Привязки к гибридным архитектурам

Дальнейшее развитие

Заключение

Рассмотренные задачи

Нахождение суммы элементов двоичного дерева

Дано двоичное дерево, каждый узел которого содержит *float* значение. Требуется найти сумму всех элементов

Вычисление Hash-функции от большого объёма данных.

Применить hash к данным, хранящимся в файле

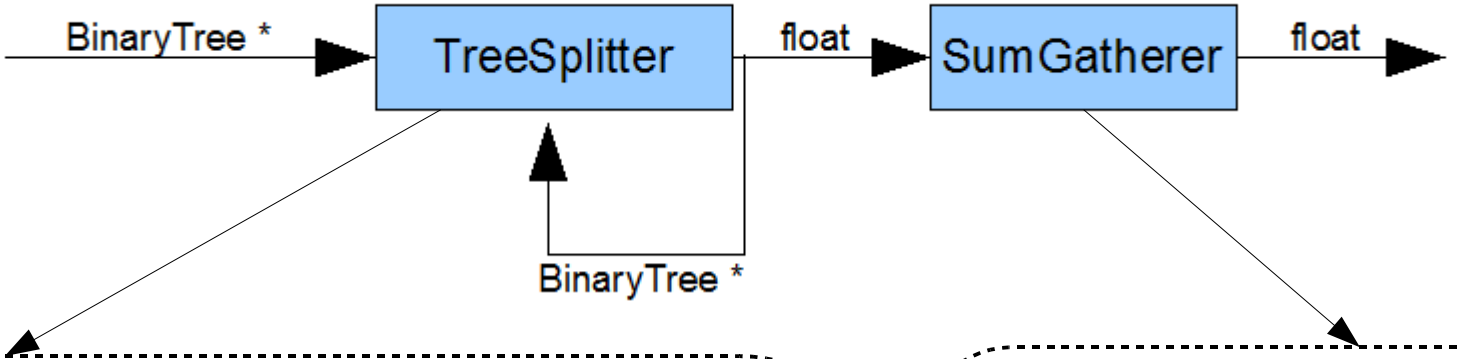
Вычисление чисел Фибоначчи

Вычислить *n*-ое число, используя формулу $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$

Нахождение всех простых чисел из отрезка

Найти все простые числа, принадлежащие заданному диапазону

Нахождение суммы элементов двоичного дерева




```
virtual void process(BinaryTree *data)
{
	if (data->getChildCount() <= 1024)
	 {
		sendNext(data->getSum());
	}
	else
	 {
		float res = 0.0f;
		sendThis(data->getLeft());
		sendThis(data->getRight());
		sendNext(data->getValue());
	}
}
```

```
float res;

virtual void onBegin()
		{ res = 0.0f; }

virtual void preEnd()
		{ sendNext(res); }

virtual void process(float data)
		{ res += data; }

TreeSumGatherer()
	:Node("TreeSumGatherer", true)
		{ /*nothing*/ }
```

Результаты тестирования

Метрика	Суммирование элементов дерева			Нахождение простых чисел			Обработка файла			Вычисление чисел Фибоначчи		
	Serial	ttgLib	TBB	Serial	ttgLib	TBB	Serial	ttgLib	TBB	Serial	ttgLib	TBB
Количество строк кода	24	61	65	28	73	99	28	75	93	19	57	42
Время работы (4 cores) [1]	0.12 с (100%)	0.041 с (298%)	0.039 с (312%)	2.97 с (100%)	0.63 с (471%)	0.62 с (479%)	1.47 с (100%)	0.37 с (393%)	0.47 с (309%)	0.38 с (100%)	0.094 с (401%)	0.095 с (401%)
Время работы (2 cores)[2]	0.21 с (100%)	0.11 с (192%)	0.1 с (196%)	4.83 с (100%)	2.48 с (194%)	2.43 с (198%)	3.09 с (100%)	1.57 с (196%)	1.56 с (198%)	0.48 с (100%)	0.29 с (164%)	0.30 с (159%)
Время работы (1 core with HT)[3]	0.3 с (100%)	0.21 с (142%)	0.22 с (136%)	6.82 с (100%)	4.93 с (138%)	5.16 с (132%)	4.6 с (100%)	2.98 с (154%)	2.94 с (156%)	1.37 с (100%)	1.15 с (119%)	1.17 с (117%)

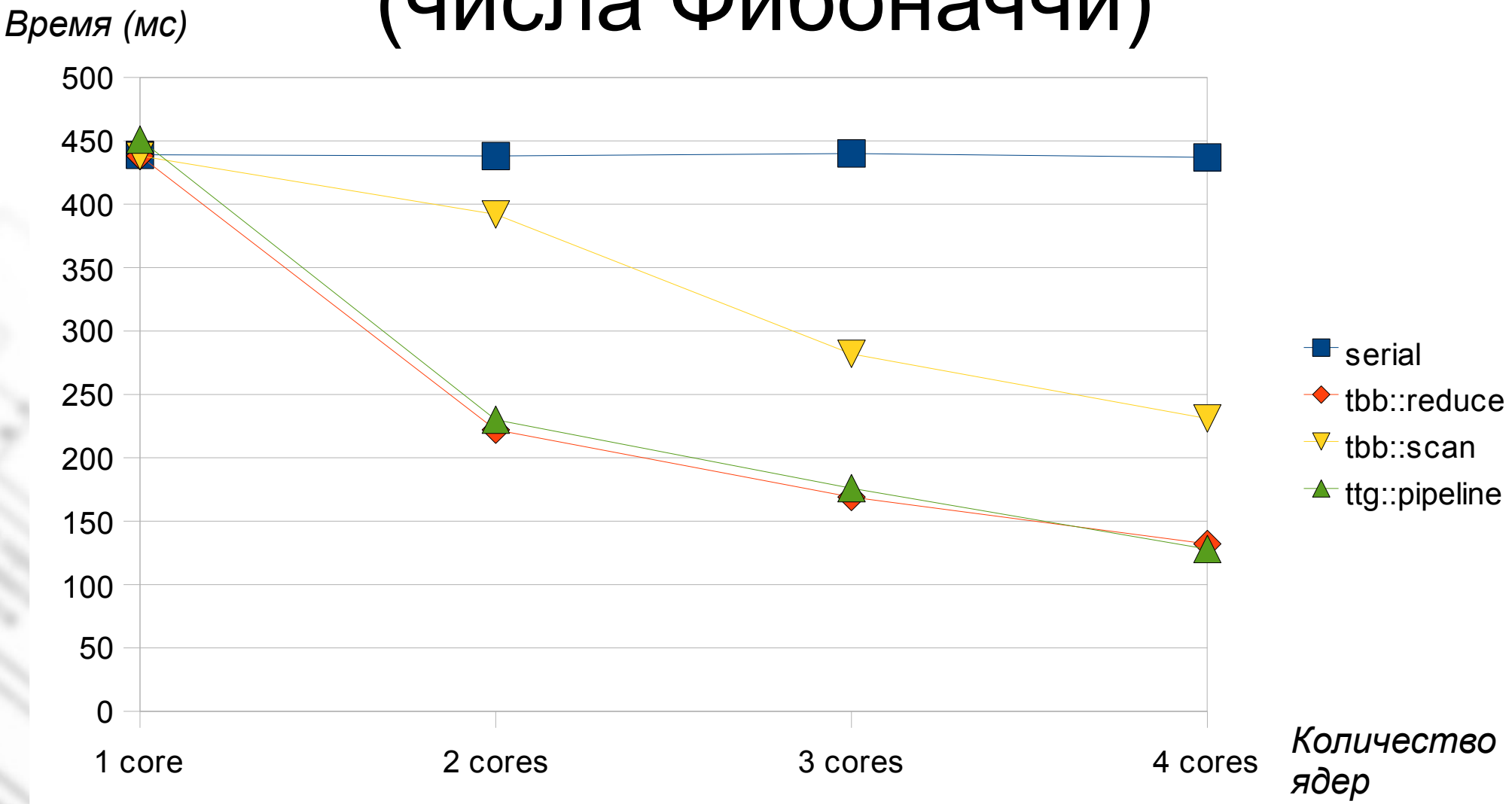
Конфигурации тестовых машин:

[1] Intel Core 2 Quad Q9400 @2.66 GHz, 2 GB RAM

[2] Intel Pentium D 945 @3.4 GHz, 1.5 GB RAM

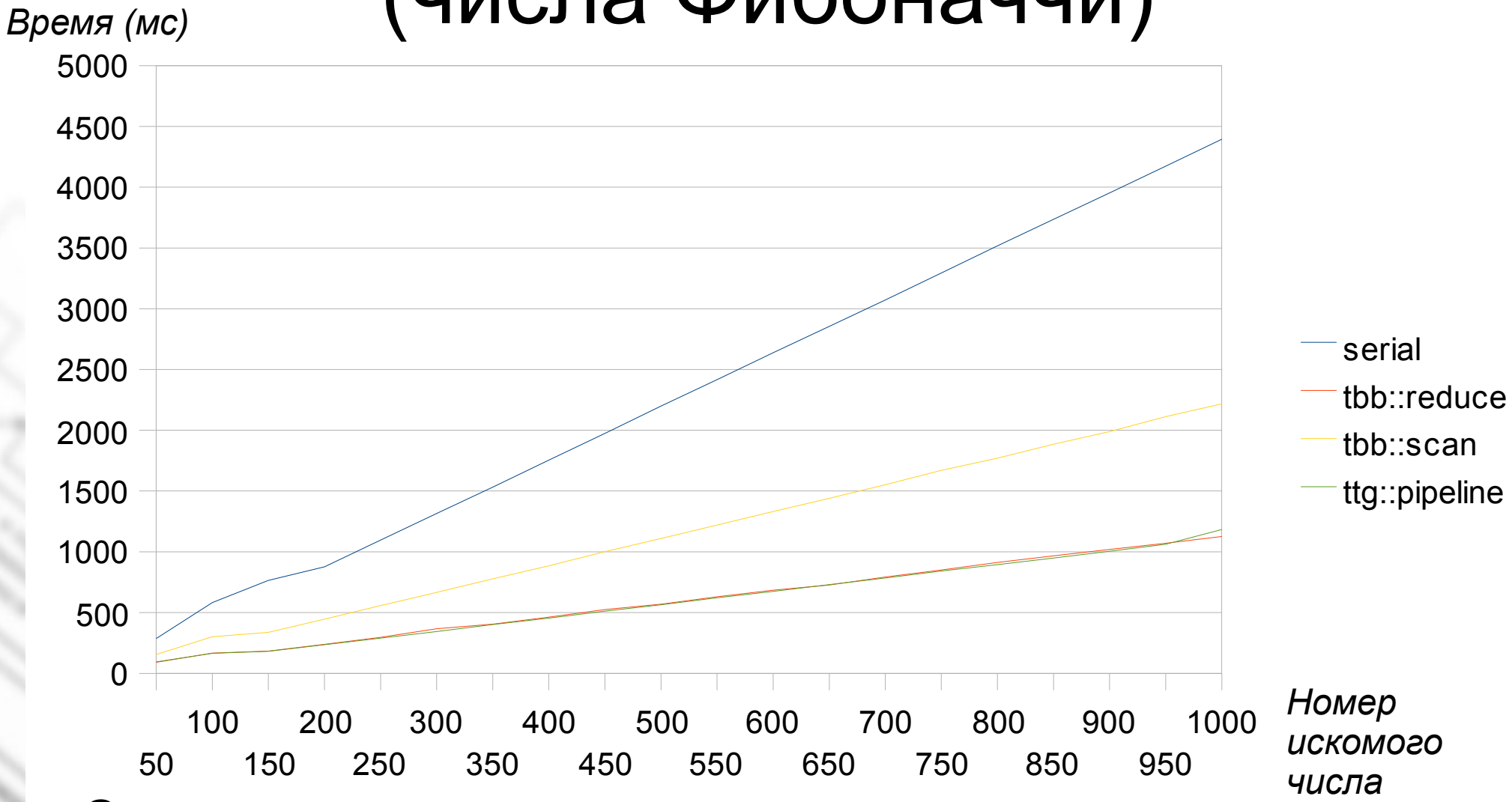
[3] Intel Atom N270 @1.6 GHz, 2 GB RAM

Результаты тестирования (числа Фибоначчи)



Зависимость времени работы от количества ядер

Результаты тестирования (числа Фибоначчи)



Зависимость времени выполнения от размера входных данных

План выступления

Описание библиотеки

Примеры решения CPU-задач

Привязки к гибридным архитектурам

Дальнейшее развитие

Заключение

Основная идея

Идея:

- В любом звене код обработки данных может дублироваться (для каждой архитектуры — свой вариант)
- В зависимости от конфигурации, звено будет обрабатывать данные на одном из доступных вычислителей.

Примеры (CPU + GPU):

- Моделирование систем частиц в компьютерных играх
- Ускорение научного моделирования с использованием явных разностных схем

Развитие идеи

Улучшения:

- Каждое звено может обрабатывать данные одновременно на нескольких вычислителях
- «Умный» менеджер распределяет звенья по вычислителям с учётом получаемого ускорения

Примеры:

- Перемножение матриц по Штрассену.
  - CPU-обработчик разбивает матрицу на блоки и перемножает их с размера N
  - GPU-обработчик начинает перемножать блоки с размера kN ($k > 1$)

Реализация

- Для подобных звеньев нужно добавить «привязку» к соответствующей архитектуре.
- «Привязка» является шаблонным классом, который:
  - Проверяет, может ли данный блок данных быть обработан на вычислителе
  - Оценивает сложность обработки
  - Производит обработку
- В настоящий момент поддерживаются «привязки» к архитектуре NVidia CUDA

План выступления

Описание библиотеки

Примеры решения CPU-задач

Привязки к гибридным архитектурам

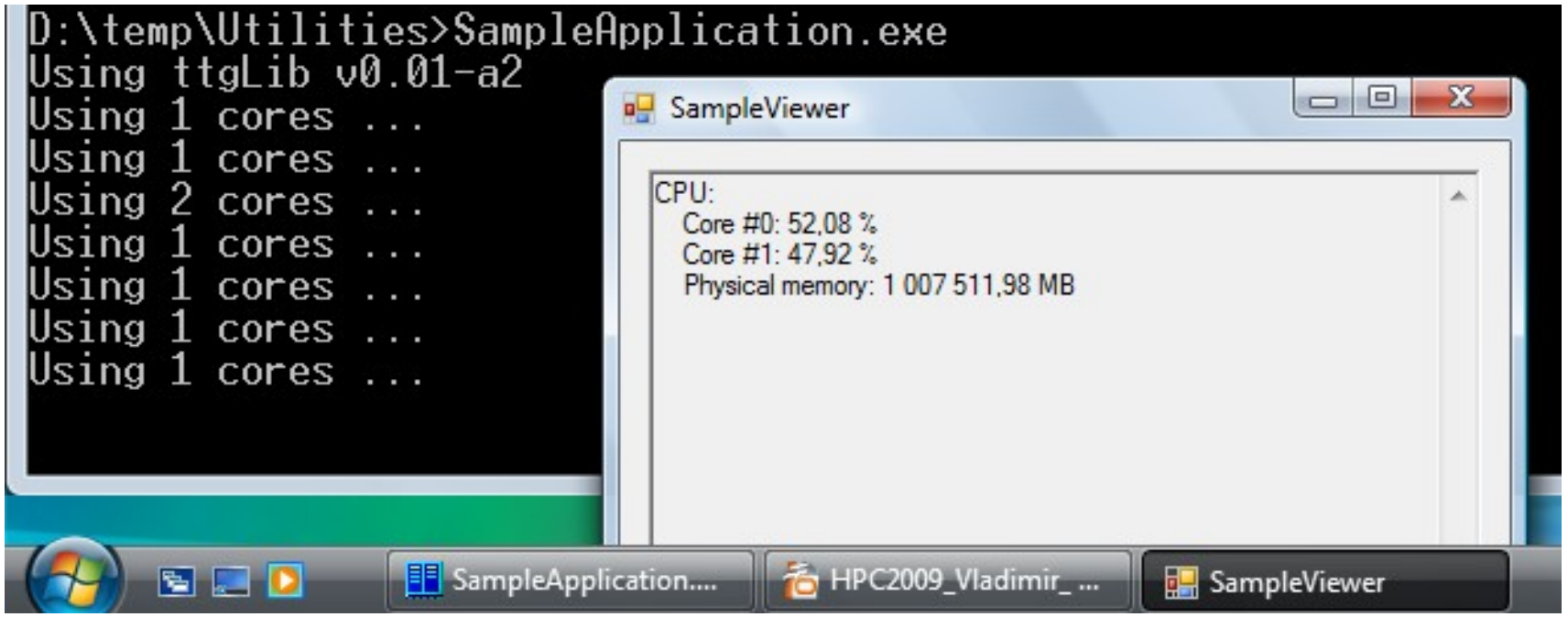
Дальнейшее развитие

Заключение

Ближайшее будущее

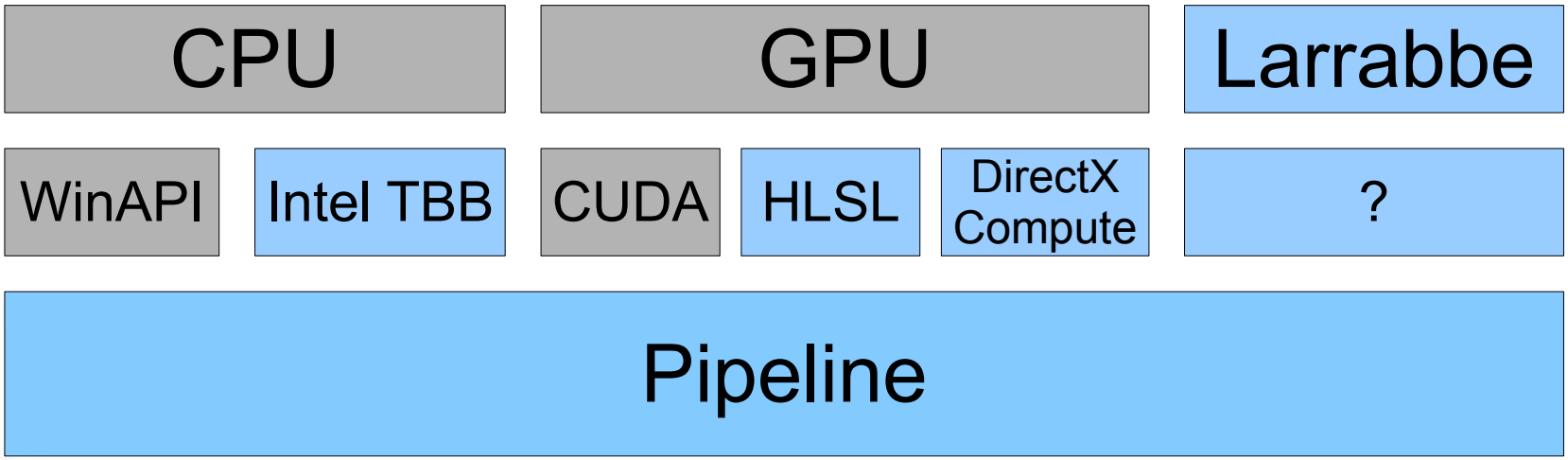
Расширение системы утилит:

- *ttgLib* собирает статистическую информацию и передаёт её внешней утилите
- Утилита производит её визуализацию и может влиять на работу конвейера (например, разрешить/запретить использование отдельных устройств)



Ближайшее будущее

Расширение поддерживаемых архитектур



Интеграция с библиотекой Intel TBB

- ttgLib SDK будет устанавливаться как дополнение к Intel TBB SDK
- ttg::pipeline будет «дружить» с tbb::task_scheduler
- Появится возможность использовать предложенный подход совместно с примитивами Intel TBB

План выступления

Описание библиотеки

Примеры решения CPU-задач

Привязки к гибридным архитектурам

Дальнейшее развитие

Заключение

Заключение

- Была разработана конвейерная модель представления программ
- Полученные наработки были оформлены в виде библиотеки ttgLib
- Было проведено сравнение предложенного подхода с библиотекой Intel TBB
- Была реализована базовая версия интеграции с NVidia CUDA

Вопросы?

Информация о проекте:

Сайт проекта: http://ttglib.org
Портал проекта: ttglib.codeplex.com
Проект поддержан программой «У.М.Н.И.К.»

Контактная информация:

Кривов Максим, m_krivov@cs.msu.su
Притула Михаил, pritmick@yandex.ru

Дополнительные слайды

Многомашинный конвейер

Идея:

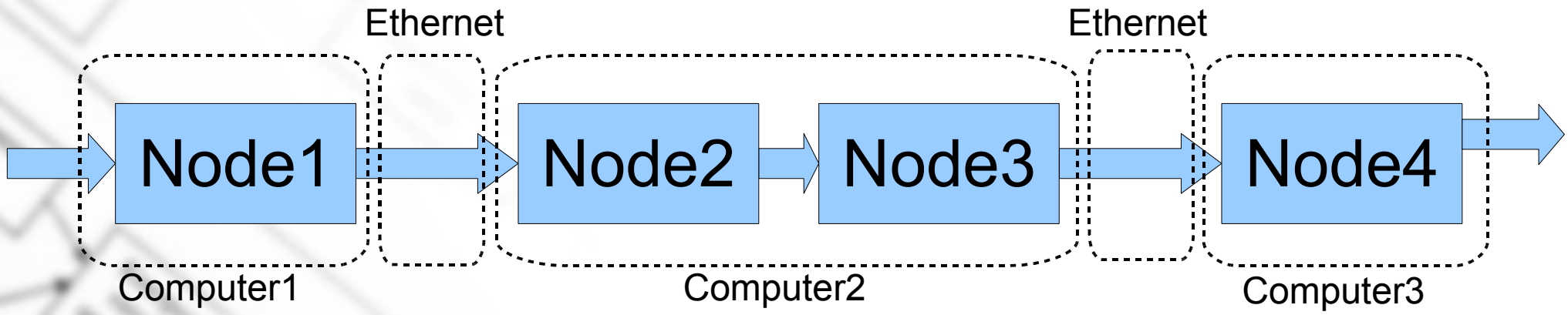
Весь конвейер бьётся на подсистемы, каждая из которых выполняется на своей машине

Все данные при переходе из одной подсистемы в другую сериализуются и передаются по сети

Реализация:

Каждое звено реализуется как отдельная dll

Каждую итерацию происходит динамическое перестроение конвейера с целью устранения узких мест



Логгирование и визуализация

Визуализация:

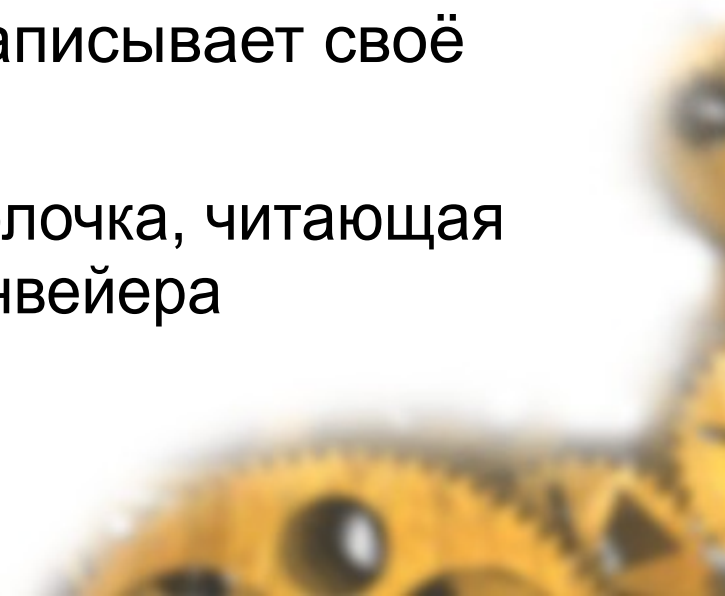
Каждую n-ую итерацию конвейер «замораживается», после чего его состояние запоминается

Полученные данные используются внешней программой для независимой визуализации работы

Логгирование:

В конце каждой итерации конвейер записывает своё состояние в log-файл.

Для воспроизведения используется оболочка, читающая log-файл и симулирующая работу конвейера



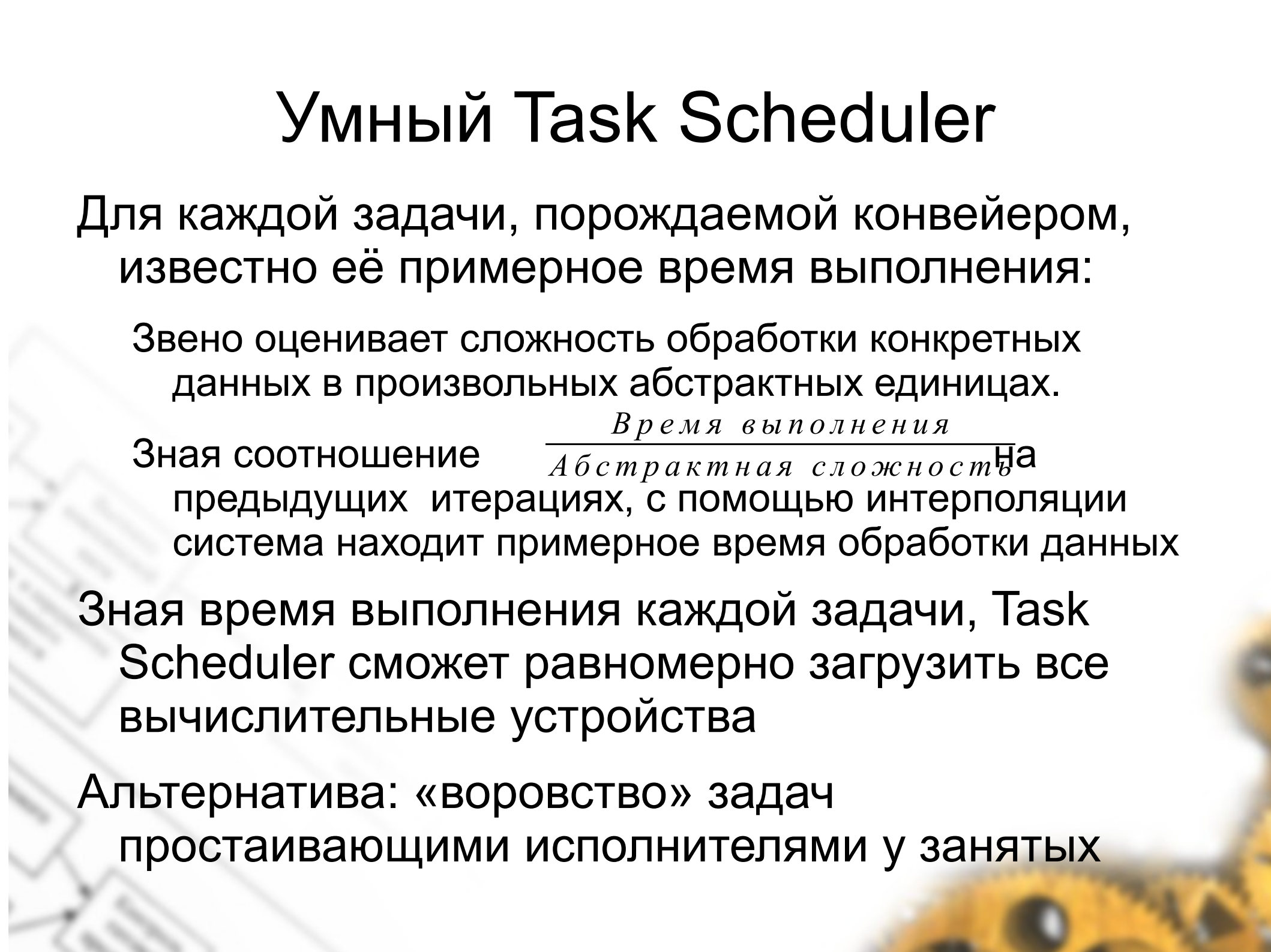
Умный Task Scheduler

Для каждой задачи, порождаемой конвейером, известно её примерное время выполнения:

- Звено оценивает сложность обработки конкретных данных в произвольных абстрактных единицах.
- Зная соотношение $\frac{Время\ выполнения}{Абстрактная\ сложность}$ на предыдущих итерациях, с помощью интерполяции система находит примерное время обработки данных

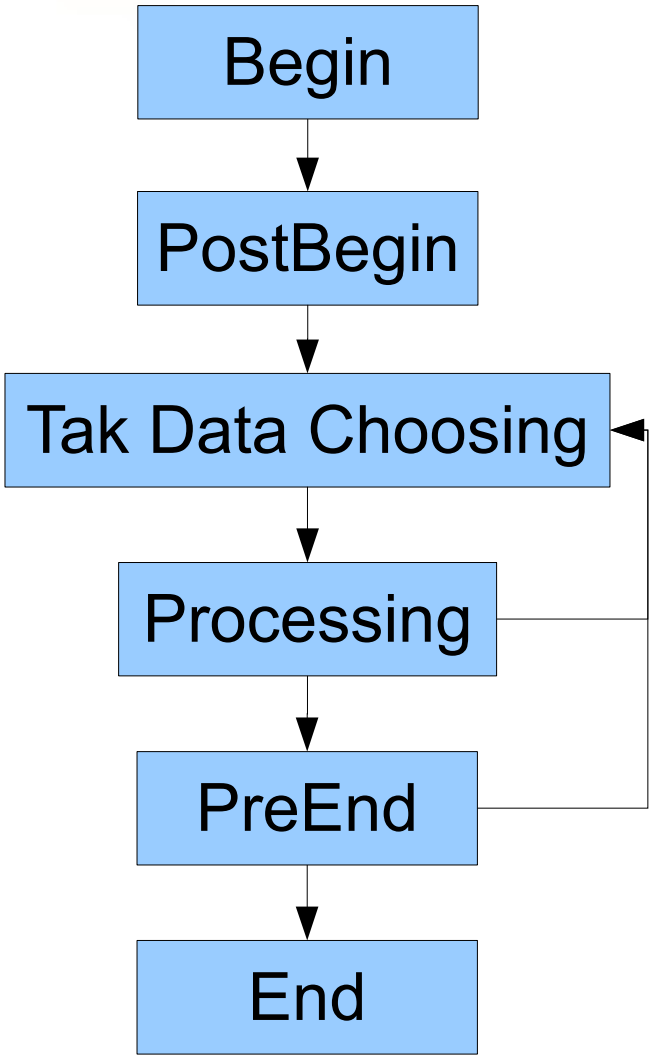
Зная время выполнения каждой задачи, Task Scheduler сможет равномерно загрузить все вычислительные устройства

Альтернатива: «воровство» задач простаивающими исполнителями у занятых



Модель конвейера

- Каждое звено может находиться в одной из 6 стадий
- Пользователь может задать поведение для любой стадии (перекрыв виртуальный метод)
- Для синхронизации могут быть использованы:
  - События onBegin()/onEnd()
  - Однопоточная обработка звена
  - Прямое обращение к звену по имени



Стадии звена