Динамическое распараллеливание программ на базе параллельной редукции графов
Назначение, функциональность и архитектура T-системы

В.А. Роганов <var@msu.ru>
А.Ф. Слепухин <pooh@msu.ru>

Центр телекоммуникаций и технологий Интернет МГУ им. М.В. Ломоносова

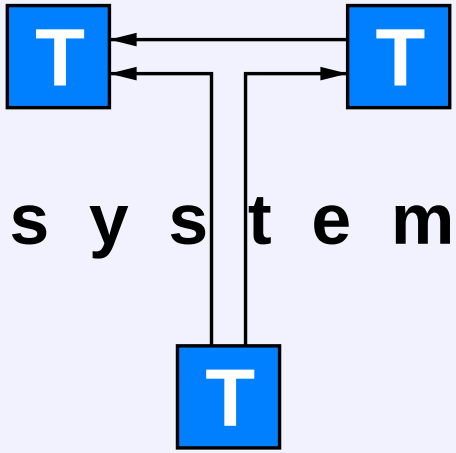
Назначение, функциональность и архитектура T-системы

- Назначение T-системы
- Функциональность T-системы
- Архитектура ПО T-системы

Назначение T-системы

- Какие задачи решает T-система
- Истоки и история создания
- Перспективы развития

Назначение T-системы



T-система — среда программирования с поддержкой автоматического динамического распараллеливания программ

Для реализации концепции автоматического динамического распараллеливания программ в T-системе предложена новая модель организации вычислительного процесса, основанная на следующих базовых принципах:

В качестве основной парадигмы рассматривается функциональное программирование. Программа представляет собой набор чистых (без побочных эффектов) функций. Каждая функция может иметь несколько аргументов и несколько результатов.

В то же время тела функций могут быть описаны в императивном стиле (на языках типа FORTRAN, C и т. п.) Важно только, чтобы:

- Всю информацию извне функция получала только через свои аргументы;
- Вся информация из функции передавалась только через явно описанные результаты.

Вызов функции G, производимый в процессе вычисления функции F выполняется нетрадиционным способом (т. н. сетевой вызов функции).

При этом порождается новый процесс с несколькими входами (в соответствии с числом аргументов функции G) и несколькими выходами (в соответствии с числом результатов функции G).

Выходы нового процесса связываются с соответствующими переменными процесса F отношением «поставщик — потребитель» , и тем самым, переменные-потребители принимают неготовые (не вычисленные) значения. Порожденный процесс G должен вычислить функцию G и заменить неготовые значения у всех своих потребителей на соответствующие результаты функции G.

Перспективы развития

Три базисных источника информации, существующих до начала разработки новой (промышленной) версии ядра T-системы:

- T-система (включая все ее компоненты на момент 15.05.2000)
- Находящееся в свободном доступе ПО для кластеров
- Совокупность всех алгоритмов и идей, накопленных участниками T-system group на момент 15.05.2000

Работы, которые должны быть выполнены на этапе разработки новой версии ядра T-системы:

Собственно T-ядро:

- Системные библиотеки, обеспечивающие «T-system safe»-взаимодействия прикладных программ с ядром ОС Linux.

- Система управления памятью (в т. ч. стеками), процессами и коммуникациями
- Транспортно- и архитектурно-независимая версия T-системы
- Использование MPI в качестве транспорта
- Использование MPI-программ в качестве вычислительных узлов
- Приоритеты задач и иерархическая структура выч. ресурсов (планировщик)
- Средства управления кластером для T-системы (инициация, рассылка, ...)
- Средства обеспечения отказоустойчивости вычислений

Библиотеки:

- Основные прикладные параллельные вычислительные библиотеки
- Параллельные прикладные библиотеки (ввод/вывод, объекты и т. д.)

Базовые средства разработчика:

- Графические интерфейсы (WWW/Java) для администрирования/мониторинга
- Базовые средства для разработки T-приложений (сборка, запуск, ...)
- Параллельный отладчик (Трассировка, визуализация трассы, повтор)
- Профилировщик приложений
- Система тестирования ядра T-системы (набор тестов и демонстр. примеров)

Функциональность T-системы

- «За» и «против» функционального подхода
- Система инвариантов T-системы
- Теоремы (Черча-Россера и др.)
- Параллельная редукция графов
- Расширения классической схемы
- Мемоизация

«За» и «против» функционального подхода

Счастливые билеты

```
tickets ds = (ds, foldl (\n c -> 10*n + digitToInt c) 0 ds) :
 [("("++ld++[op]++rd++")", f lv rv) |
    (op,f) <- [('+',(+)),('-',(-)),('*',(*)),('/',(div))],
          n<-[1..length ds-1],
      (ld,lv) <- tickets (take n ds),
      (rd,rv) <- tickets (drop n ds),
        op /= '/' || (rv /= 0 && lv `mod` rv == 0)]

happy = map fst . (filter ((==)100 . snd)) . tickets

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
ticket.hs
Type :? for help
Main> happy "314159"
["(3+(1+(4*(15+9))))","(31+((4*15)+9))","((3+1)+(4*(15+9)))",
 "(((3*14)-1)+59)","((31+(4*15))+9)","((3*14)-(1-59))",
 "((31*4)-(15+9))","(((31*4)-15)-9)","((3-1)*(4+(1+(5*9))))",
 "((3-1)*((4+1)+(5*9)))"]
Main> [Leaving Hugs]
```

Расстановка ферзей

```
module Queens where
import Gofer

queens number_of_queens  = qu number_of_queens where
    qu 0           = [[]]
    qu (m+1)       = [ p++[n] | p<-qu m, n<-[1..number_of_queens], safe p n ]

safe p n           = all not [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
                     where m = 1 + length p

check (i,j) (m,n) = j==n || (i+j==m+n) || (i-j==m-n)

q = putStr . layn . map show . queens
```

Система инвариантов T-системы

- Редукция графов — меняет граф, не меняет его значение
- Стратегия вычислений — решает, что вычислять в первую очередь
- Распределение работы — решает, какую задачу кому отдать

Теоремы (Черча-Россера и др.)

Теорема Черча-Россера

Если $\mathcal{R}$ — отношение, то пусть $\mathcal{R}*$ означает его конечно-транзитивное замыкание.

Пусть $X \to Y$ означает, что X редуцируется в Y.

Обозначим X cnv $Y \Leftrightarrow X \to Y \vee Y \to X$.

Тогда:

X cnv$* \; Y \Rightarrow \exists N : X \to * \; N \wedge Y \to * \; N$

Теорема о единственности нормальной формы (следствие из т. Ч-Р)

Любые две нормальные формы лямбда-выражения алфавитно-эквивалентны.

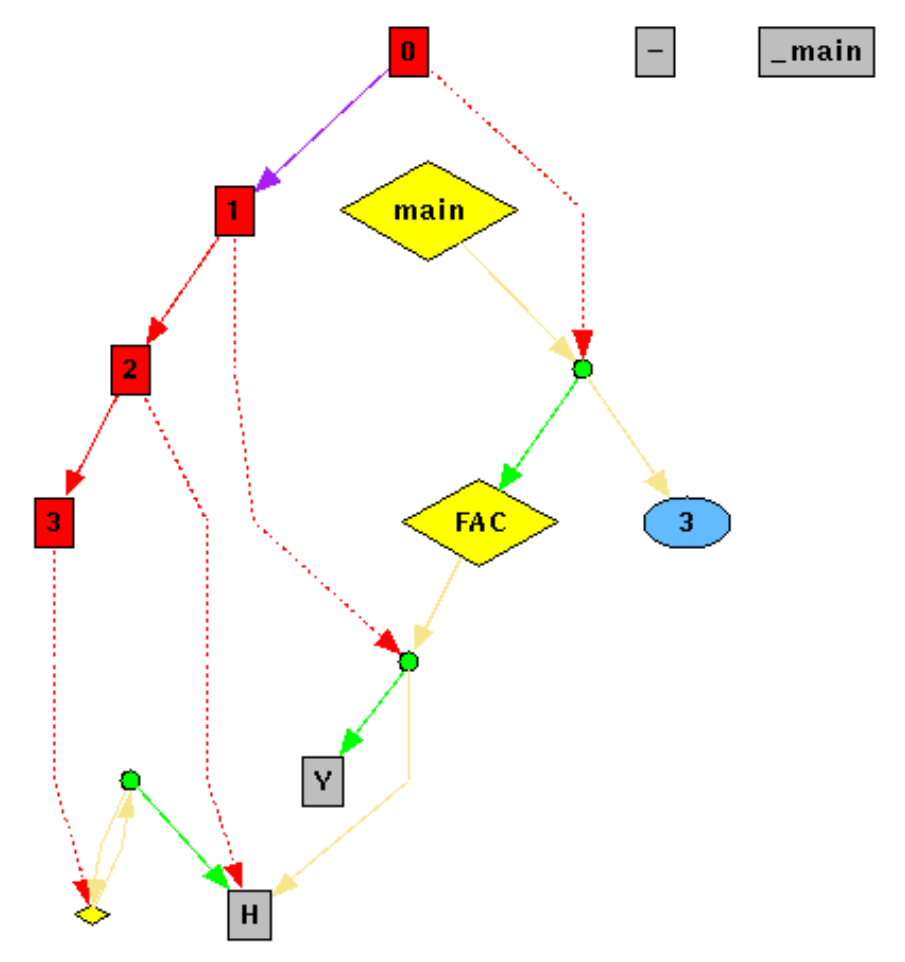
Ромбическое свойство

Если выражение E может быть редуцировано к двум выражениям $E1$ и $E2$, то существует N, в которое могут быть редуцированы $E1$ и $E2$.

Теорема стандартизации

Если выражение E имеет нормальную форму, то редукция самого левого из самых внешних редексов на каждом этапе редукции E гарантирует достижение этой нормальной формы с точностью до алфавитной эквивалентности.

Параллельная редукция графов



Краткая теоретическая база параллельной редукции графов

λ-исчисление — это исчисление анонимных (безымянных) функций. Оно дает, во-первых, метод представления функций и, во-вторых, множество правил вывода для синтаксического преобразования функций.

Редукция — это процесс упрощения λ-выражения.

- δ-редукция — это способ преобразования цепочек символов, являющихся функциональными константами.
- α-редукция заключается в переименовании свободных переменных
- β-редукция — это процесс копирования тела абстракции с заменой всех вхождений связанной переменной на выражение аргумента.

Редекс — это редуцируемое выражение.

λ-выражения могу быть представлены в виде графов. При этом множественные ссылки на одно и то же выражение аргумента представляются множеством дуг, идущих к единственной разделяемой копии соответствующего графа аргумента.

При редукции графа выражения вполне возможно присутствие в этом графе множества редексов в любой момент. Вследствии прозрачности ссылок мы знаем, что эти редексы будут всегда вычисляться с одинаковым результатом независимо от того, где и когда это вычисление имеет место. Поэтому вполне возможно вычислять их одновременно, разместив на отдельных процессорах. Каждый процессор может затем приступить к редукции соответствующих редексов, возможно генерируя новые процессы и, таким образом, создавая новые параллельные задачи. Ниже приведен алгоритм, реализующий параллельную редукцию графов.

Параллельная редукция графов

```
;; Graph node definition

(defstruct (n)
  (in)  (out)
  (fun) (val)
  (thread)
  (ct)  (dt))

;; Main functions

(defun getfor (n)
  (add-link *node* n)
  (que-put n *cal*))

(defun forget (n)
  (del-link *node* n)
  (que-put n *col*))

(defun cal (n)
  (or (null (n-out n))
      (n-thread n)
      (setf (n-thread n) (start-thread n))))

(defun col (n)
  (or (n-out n)
      (null (n-thread n))
      (setf (n-thread n) (kill-thread n))))

(defun start (n)
  (call-with-node n
    (compute n))
  (if (n-ct n) (funcall (n-ct n) n))
  (mapcar #'kill-thread (n-out n)))

(defun stop (n)
  (mapcar #'kill-thread (n-out n))
  (if (n-dt n) (funcall (n-dt n) n))
  (call-with-node n
    (mapcar #'forget (n-in n))))
```

Расширения классической схемы

- Стратегии выбора и распределения работы
- Вызов по состоянию
- Подъем функциональной схемы
- Монотонные объекты
- Инкрементальные вычисления

Стратегии выбора и распределения работы

При вызове T-функции имеется две возможности, что считать дальше:

- Сначала считать «вширь»
  Порождается больше пренатальных процессов,
  вычисления становятся более «ленивыми»
  и может потребоваться больше памяти для стеков
- Сначала считать «вглубь»
  Экономия на стеках и на кэше,
  вычисления становятся более «энергичными»
  и порождается меньше пренатальных процессов

Вызов по состоянию

Результатом работы функции может быть некоторая фиктивная величина, которая означает, что над глобальными данными произведено некоторое преобразование.

Мемоизация

Для чистых функций (не имеющих побочных эффектов) можно запоминать результат их вычислений и затем при повторных вызовах от одних и тех же аргументов просто брать значение из таблицы.

Архитектура ПО T-системы

- Программный и пользовательский интерфейсы T-системы
- Ядро параллельной редукции графов
- Кластерный уровень
- Компилятор (source-to-source converter)
- Работа в качестве вычислительного сервера
- Приложения

Программный и пользовательский интерфейсы T-системы

- Библиотека классов C++ и интерфейс с другими языками
- Гладкие T-расширения языков C и Fortran
- Среда для разработки, отладки и профилировки программ

Библиотека классов C++ и интерфейс с другими языками

В качестве средства реализации ядра T-системы выбран язык C++. Это обеспечивает хорошую эффективность и переносимость кода, а также упрощает достижение таких важных целей, как модульность и расширяемость ПО. В ряде случаев (например, для многих невычислительных задач), использование C++ также очень желательно и с точки зрения удобства разработки параллельных приложений.

Интерфейс T-системы для разработчика ПО, программирующего на языке C++, выглядит как совокупность классов, которые он может гармонично включить в дизайн своего приложения и эффективно использовать все возможности T-системы на всех уровнях распараллеливания, начиная от динамического распределения вычислительной нагрузки и заканчивая интерфейсами с клиентской стороной.

Гладкие T-расширения языков C и Fortran

Важным достоинством новой версии T-системы является то, что она поддерживает гладкое расширение языка C. Гладким оно названо потому, что все необходимые конструкции для указания мест распараллеливания программы просто и естественно погружены в синтаксис языка C.

Компилятор преобразует программу на языке TC в программу на языке C. При этом он производит необходимый анализ программы и вставляет в текст все необходимые обращения к ядру T-системы. Затем C-программа транслируется обычным оптимизирующим компилятором.

Ключевым свойством гладкого расширения языка является то, что программа на этом языке превращается в программу на языке C и без специального компилятора, если определить ключевые слова, добавленные в язык TC как макросы.

Предполагается, что это удобство будет оценено пользователями, которым будет проще начинать разработку и переделку своих программ под T-систему.

TFortran: первая версия будет на базе `f2c` — Fortran-to-C конвертора.

Среда для разработки, отладки и профилировки программ

Среда разработки для T-системы является надстройкой над средой разработки обычных последовательных программ.

Дополнения будут сделаны в следующих пунктах:

- Поддержка расширенного синтаксиса (конструкций T-расширений языков)
- Online-help по интерфейсным классам T-системы
- Запуск T-задачи
  - в нормальном режиме
  - с профилировкой
  - под отладчиком
  - с трассировкой и возможностью повторения трассы
- Вызов дополнительных утилит (визуализация статистики, состояния и т. п.)

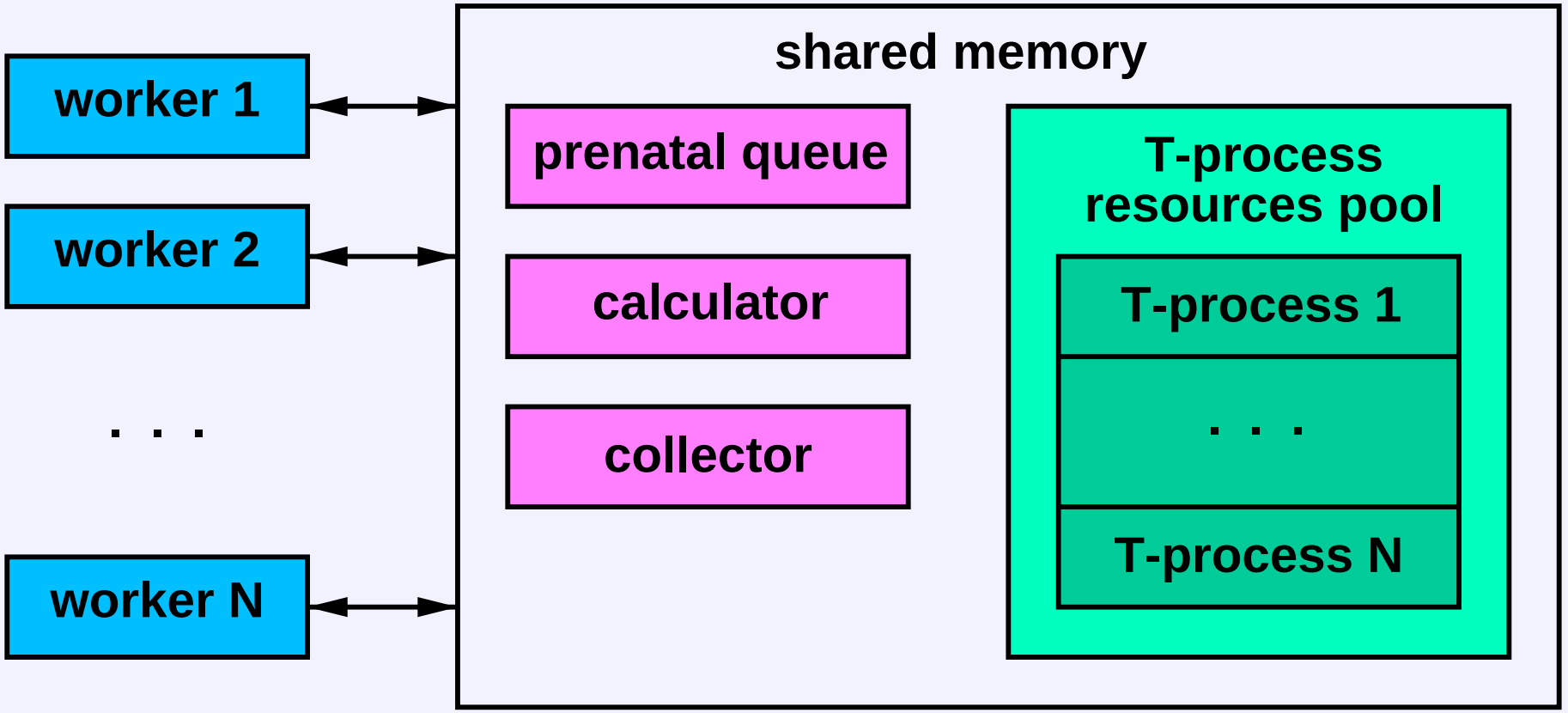
Ядро параллельной редукции графов

- Структура SMP-вычислителя
- Внешнее вычисление узлов графа
- Сопряжение с кластерным уровнем
- Отказоустойчивость

Структура SMP-вычислителя

- Основные классы SMP-вычислителя
- Диаграмма связей между объектами
- Система управления памятью

Структура SMP-вычислителя



Основные классы SMP-вычислителя

`Continuation` — T-процесс в пренатальном состоянии (функция с аргументами)

`Node` — Динамическая составляющая T-процесса

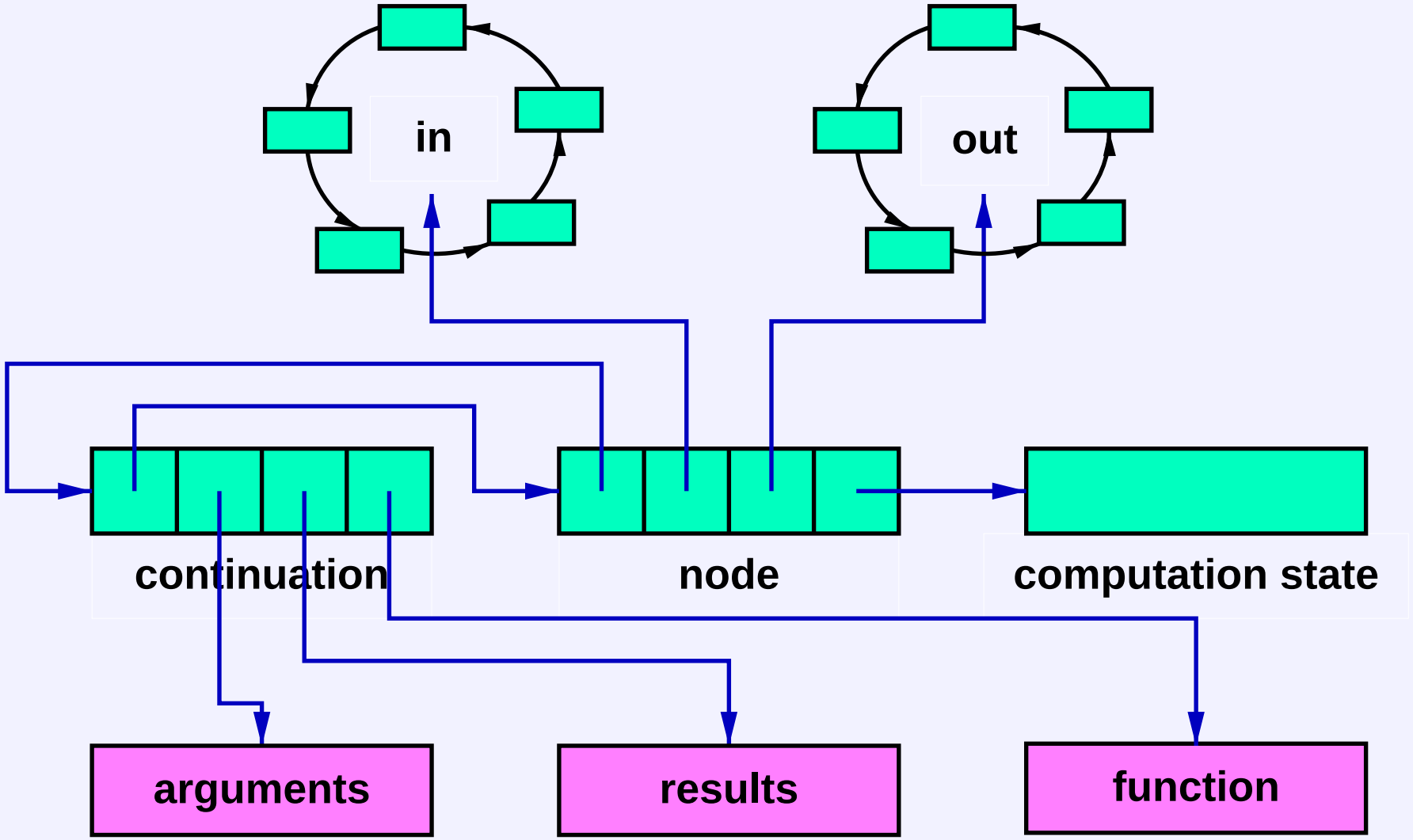
`ComputationState` — Динамические ресурсы T-процесса

`T_Var<val_t>` — Темплейтная обертка T-переменной

`T_Res<val_t>` — Темплейтная обертка результата T-функции

`Worker` — Рабочий процесс SMP-вычислителя

Диаграмма связей между объектами



Система управления памятью

- Динамическое распределение памяти на основе `mmap()`
- Быстрые чанковые аллокаторы для структур фиксированного размера
- Возможность реализации легковесных мигрируемых процессов (lightweight migratable processes)
- Автоматическое обеспечение возможности создания «контрольных точек» (checkpointing)

Внешнее вычисление узлов графа

Внешнее вычисление узла графа:

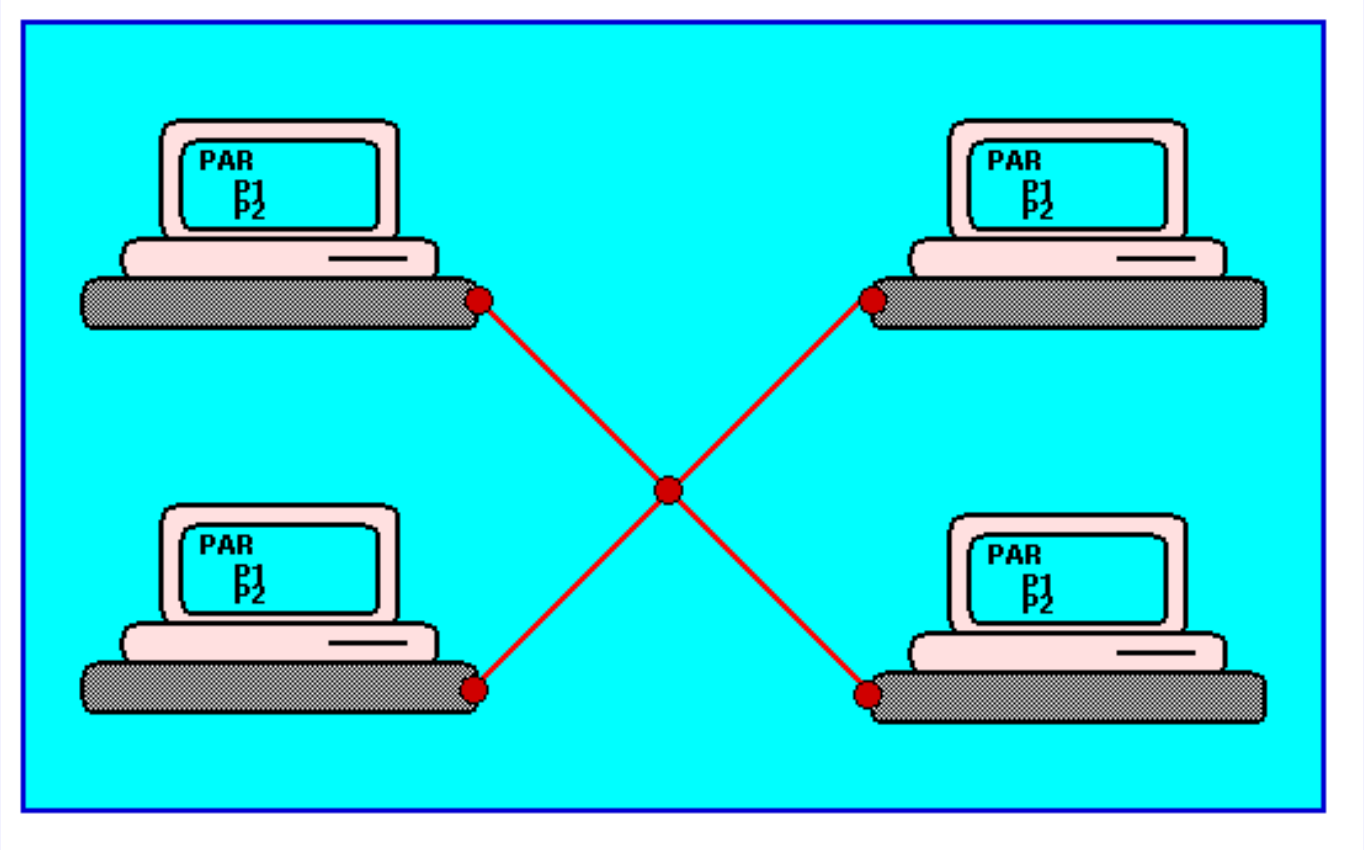
- Не требует стека
- Не требует процессора (но имеет статус «вычисляется»)
- Циклическое (для обеспечения отказоустойчивости)
- Использует информацию о свободных ресурсах в кластере

Примеры, где внешние вычисления возникают:

- Асинхронный ввод-вывод
- Вычисление на удаленном узле кластера

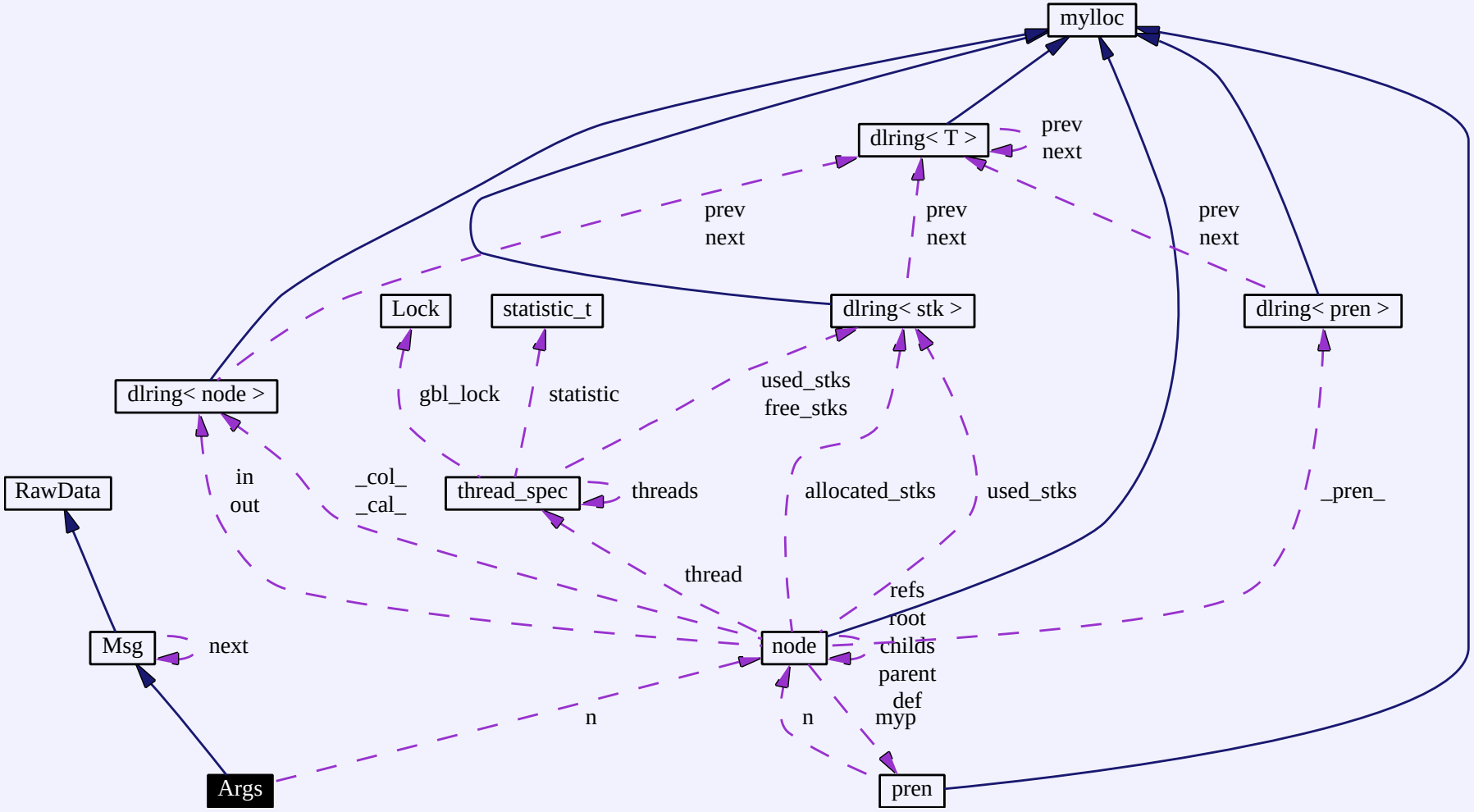
По окончанию (или сбросу) вычисления узел графа вновь попадает в очередь к калькулятору (или коллектору)

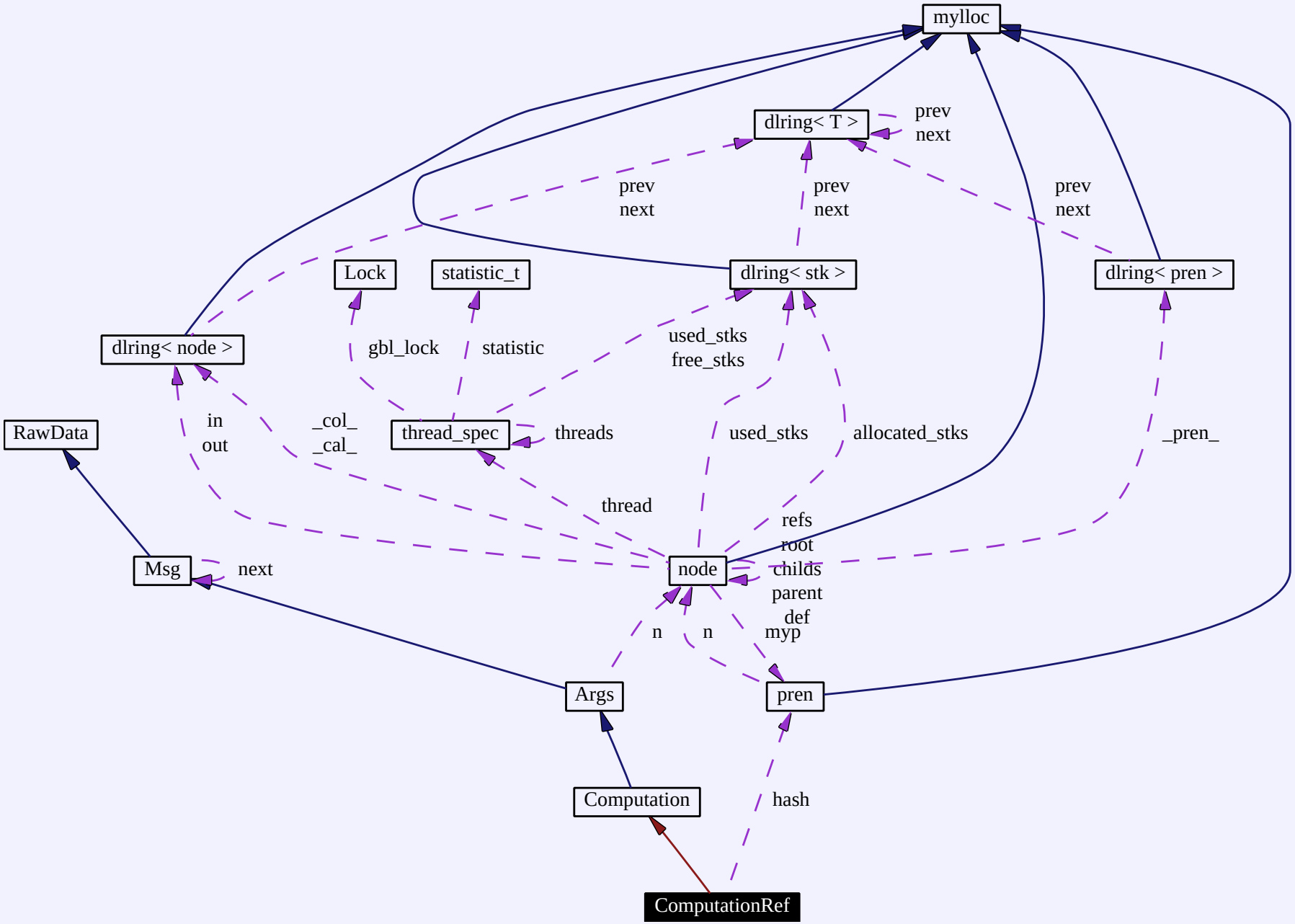
Сопряжение с кластерным уровнем



Для повышения эффективности преобразование в транспортную форму производится по необходимости.

Сопряжение с кластерным уровнем




mylloc
dlring< T >
prev
next
prev
next
prev
next
prev
next
dlring< stk >
dlring< pren >
Lock
statistic_t
dlring< node >
gbl_lock
statistic
used_stks
free_stks
RawData
in
out
col
cal
thread_spec
threads
used_stks
allocated_stks
pren
thread
refs
root
childs
parent
def
Msg
next
node
n
n
myp
Args
pren
Computation
hash
ComputationRef

Отказоустойчивость

В T-системе предусмотрена возможность динамического изменения конфигурации. Исключение и подключение компьютера может происходить без остановки процесса вычислений, при условии что сетевой транспорт аппаратно это поддерживает.

Обеспечение отказоустойчивости:

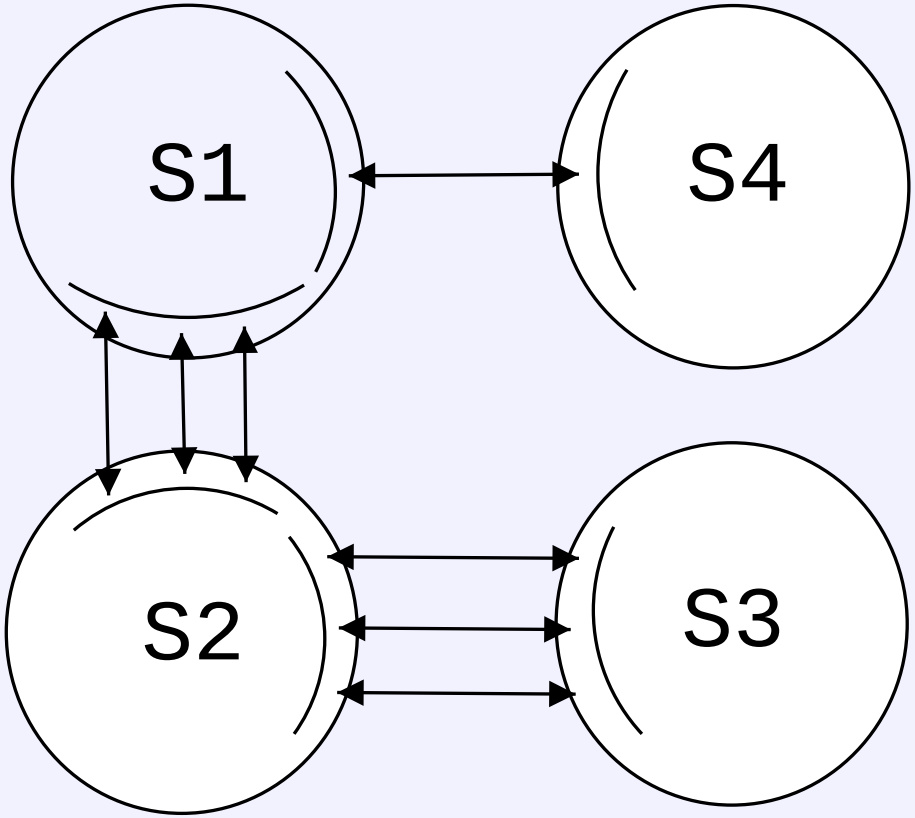
В случае отказа узла соответствующие порталы терминируют внешнее вычисление узлов графа, с которыми они были связаны.

После этого узлы повторяют цикл вычисления, используя новую информацию, где отказавший узел уже отсутствует.

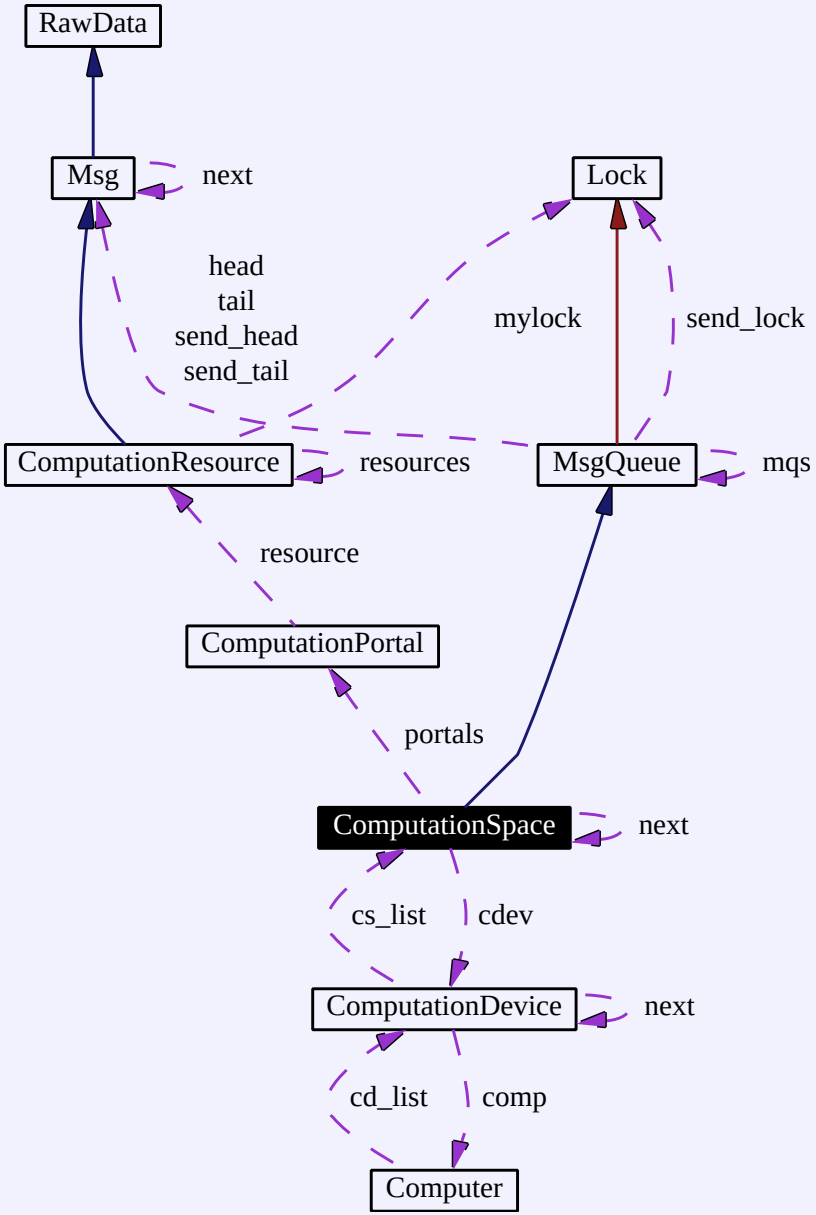
Кластерный уровень

- Вычислительные пространства и порталы
- Асинхронный ввод-вывод
- Активные сообщения
- Сопряжение с реализациями MPI/IMPI
- Распределенная мемо-таблица
- Внешнее планирование
- Дерево вычислительных ресурсов
- Начало и окончание работы

Вычислительные пространства и порталы



Два режима работы: реальный кластер и эмуляция кластера


RawData
Msg
next
Lock
head
tail
send_head
send_tail
mylock
send_lock
ComputationResource
resources
MsgQueue
mqs
resource
ComputationPortal
portals
ComputationSpace
next
cs_list
cdev
ComputationDevice
next
cd_list
comp
Computer

Асинхронный ввод-вывод

- Ввод-вывод как внешнее вычисление
- Распараллеливание для одного и нескольких процессов

Ввод-вывод как внешнее вычисление

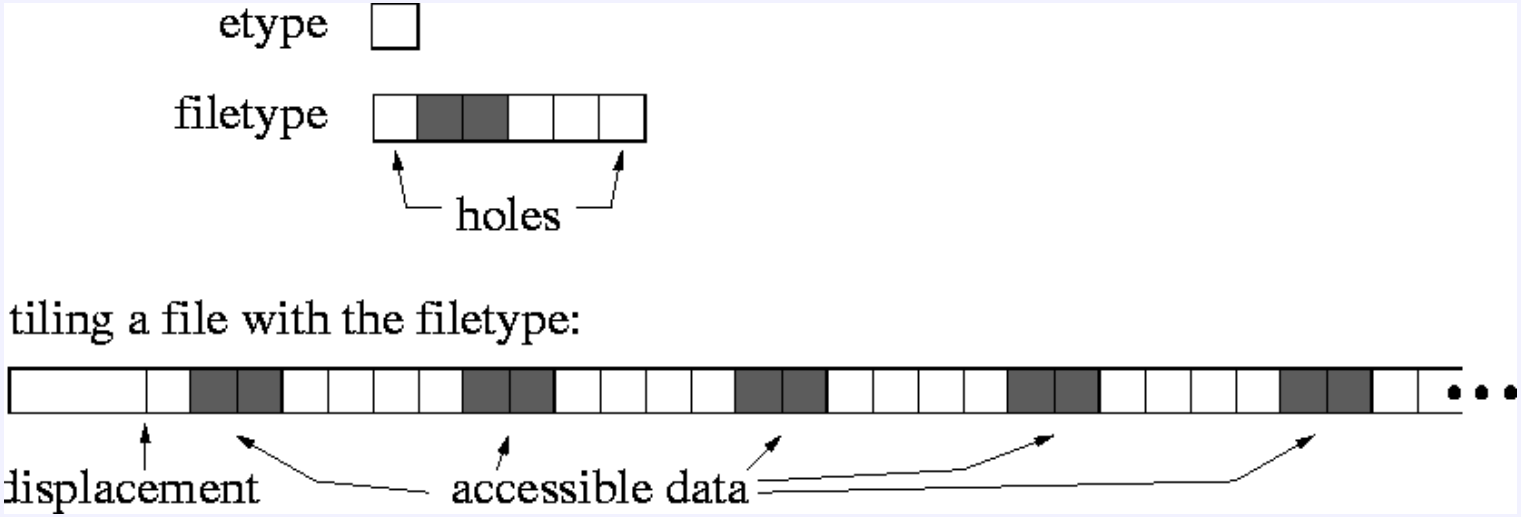
Асинхронный ввод-вывод рассматривается как внешнее вычисление

Распараллеливание для одного и нескольких процессов

PVFS обеспечивает распараллеливание ввода-вывода для одного процесса

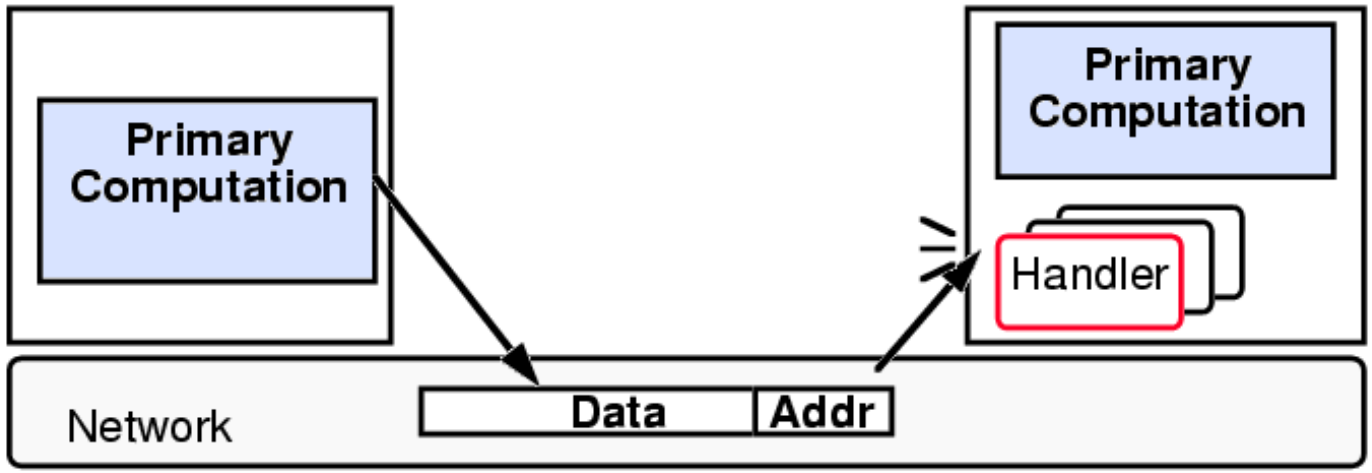
(socket I/O $\longrightarrow$ MPI)

ROMIO — подсистема MPICH, обеспечивающая асинхронный ввод-вывод для нескольких одновременно работающих процессов



Активные сообщения

Active Message Concept

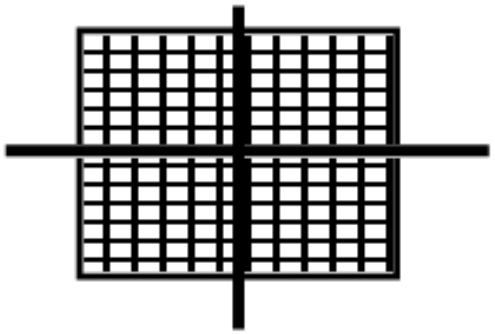


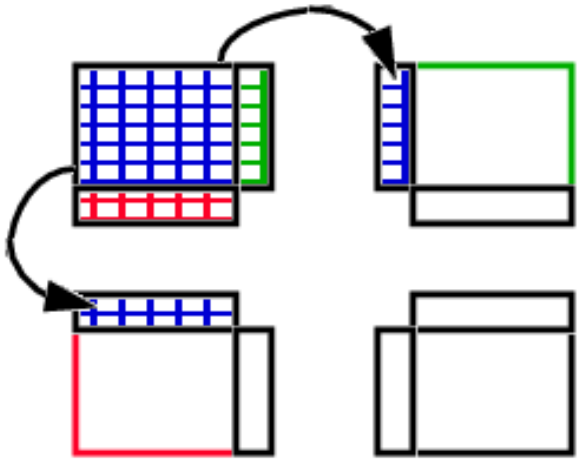
- Sender injects the message directly into the network
- Associates a small user-level handlerwith each message
- Handler executes immediately upon arrival
  - pulls the message out of the network and integrates it into the ongoing computation, or replies (as determined by higher level programming model)
- No buffering, parsing, or allocation (beyond transport).

SHPCC.8

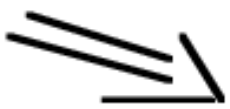
Global Address Space to Message Passing

HPF,
Fortran-D,





FORALL . . .
[= A[f(i),g(i)]



Send, Send, . . .
Compute local

Recv, Recv, . . .
Compute

?

Barrier

° Compiler already allocated the buffer space!

° Compiler knows which processor and what address to store into!

° Compiler knows if it is available!

=> Construct simple memory-to-memory transfer

(i.e., write, put, or store)

° For user decomposition, use read or get!

Сопряжение с реализациями MPI/IMPI


```
      impirun -server <count> -port <port_number>

Here, <count> is the number of client connections that the server expects
to see. When impirun is started with the ‘-server’ option, it creates a TCP/IP
socket for listening and then prints both the IP address of the local host
(in standard dot notation) and the port number of the socket (to stdout,
if on a UNIX machine). If the ‘-port’ option is specified, the server will
attempt to start on the given <port_number>. If the ‘-port’ option is not
given, the server is free to choose any port number.

      impirun -client <rank> <host:port> <cmd_line>

<rank> specifies where the processes belonging to this client should be
placed in MPI_COMM_WORLD relative to the other clients and must be a unique
number between 0 and count-1, inclusive.

     <host:port> is the host:port string provided by the server.
     <cmd_line> is implementation-specific.
```

Распределенная мемо-таблица

DPMT - Distributed Parallel Memo Table

Глобальное разделяемое между вычислительными узлами отображение

<функция,аргументы> -> вершина редуцируемого графа

Сборка мусора:

Каждую запись добавляем в список с индексом, равным целой части логарифма его времени вычисления. Каждый список просматриваем через соответствующее время.

Внешнее планирование

В процессе динамического распараллеливания может образовываться большое количество готовых к исполнению задач, значительно превышающее количество активных исполнителей (процессоров) в данном вычислительном узле.

Под внешним планированием понимается выбор узла кластера, на котором будет вычисляться данная функция.

Этот выбор осуществляется самим узлом графа на основе информации о производительности загруженности узлов кластера.

Дерево вычислительных ресурсов

Информация о производительности и загруженности узлов кластера хранится в дереве ресурсов.

При избытке готовых к исполнению вычислительных задач на одном вычислительном узле и недогруженности других узлов кластера происходит миграция задач на наиболее подходящий узел.

Узлы кластера обмениваются информацией (через широковещательные посылки) о своей загруженности и каждый узел лениво вычисляет статистику загрузки для каждого подкластера и всего кластера в целом (поддерживается иерархическая схема организации кластера). При избытке заданий узел, на который следует отправить задачу (например, пренатальный T-процесс), должен вычисляться за несколько сотен инструкций процессора. Полная загруженность кластера должна обнаруживаться за несколько инструкций.

Начало и окончание работы

Начало работы

- mpirun <user-t-executable>
- Запрос клиента

Окончание работы

- Вывод результата
- Сбор статистики и финализация

Компилятор (source-to-source converter)

- Синтаксис и семантика языка TC
- Подсчет аттрибутов
- Оптимизация обращений к T-ядру
- Порождение C-кода и C-структур

Синтаксис и семантика языка TC

- Язык TC является гладким расширением языка C
- Может быть откомпилирован как обычная C-программа
- Добавлены новые атрибуты переменных и функций
  - `tvar` — T-переменная
  - `tptr` — указатель на T-значение
  - `tarray` — T-массив
  - `tfun` — T-функция
  - `tlazy` — атрибут для ленивой семантики
  - `tmemo` — мемоизуемая функция
  - `tin` — параметр T-функции
  - `tout` — результат T-функции
- Добавлены новые функции(макросы)
  - `tecomp`
  - `tready`

Подсчет аттрибутов

Проход 1: подсчет атрибутов

Оптимизация обращений к T-ядру

Проход 2: оптимизация обращений к ядру T-системы

Порождение C-кода и C-структур

```
tfun fib (tin int n, tout int* res)
{
  if (n < 2) res = n;
  else {
    tvar int res1;
    tvar int res2;
    fib(n - 1, res1);
    fib(n - 2, res2);
    *res = res1 + res2;
  }
}
//---------------------------------

struct __fib__args
{
  T_Var<int> n ;
};

struct __fib__ress
{
  T_Res<int> res ;
};
```

```
void fib (__fib__args const* _args, __fib__ress* _ress)
  if (_args->n < 2) _ress->res = _args->n;
  else {
    T_Var<int> res1;
    T_Var<int> res2;
    Continuation<__fib__args, __fib__ress>* cnt1 =
      new Continuation<__fib__args, __fib__ress>(
        fib, new ComputationStateFactory<FastProcess>
      );
    cnt1->args->n = _args->n - 1;
    res1(cnt1, cnt1->ress->res);
    prenatal_queue->put_front((GenericContinuation*)cnt1);
    Continuation<__fib__args, __fib__ress>* cnt2 =
      new Continuation<__fib__args, __fib__ress>(
        fib, new ComputationStateFactory<FastProcess>
      );
    cnt2->args->n = _args->n - 2;
    res2(cnt2, cnt2->ress->res);
    prenatal_queue->put_front((GenericContinuation*)cnt2);
    _ress->res = res1 + res2;
  }
}
```

Тест на языке C++ с макросами (код, взятый из прототипа)

```
val_t fib (val_t k) {
   dprintf("%p: fib(%d)\n", this, k);

   int res;
   if (k < 2) return k;

   if (k < 20) /* C-call */
      return fib(k-1) + fib(k-2);
   else {      /* T-call */
      pren *k1 = CALL(fib,k-1);
      pren *k2 = CALL(fib,k-2);
      res = VAL(k1) + VAL(k2);
   }
   return res;
}
```

Тестовый пример (прототип компилятора)

```
tfun float x(int tvar u) {
       return u*2;
}

a() { return 2; }

int tfun main (int tvar e, int y) {
  tvar int a;
  if (e) {
    a *= (2);
    /*a++;*/
  } else {
    main(7,a);
a:
    main(7,a);
    main(7,a);
  }
  printf("Hello !");
}
//---------------------------------
/*tfun*/ float tfun_x(int /*tvar*/ u) {
  return TVAR_GET_CHECK(u)*2;
}

/*cfun*/ float x(int /*cvar*/ u) { return u*2; }

a() { return 2; }
```

```
int /*tfun*/ tfun_main (int /*tvar*/ e, int y) {
  /*tvar*/ int a;
  if (TVAR_GET_CHECK(e)) {
    TVAR_ASSIGN(a) = TVAR_GET_CHECK(a)*( (2));
  } else {
    tfun_main(7,TVAR_GET_CHECK(a));
a:
    tfun_main(7,TVAR_GET_CHECK(a));
    tfun_main(7,TVAR_GET_NOCHECK(a));
  }
    printf("Hello !");
}

int /*cfun*/ main (int /*cvar*/ e, int y) {
  /*cvar*/ int a;
  if (e) {
    a = a*( (2));
  } else {
    main(7,a);
a:
    main(7,a);
    main(7,a);
  }
  printf("Hello !");
}
```

Работа в качестве вычислительного сервера

- Сопряжение с прикладными программами
- Средства обеспечения безопасности

Работа в качестве вычислительного сервера

Доступ к вычислительному ядру кластера осуществляется через выделенный компьютер, выполняющий роль шлюза, и работающий под управлением доработанной версией ОС Linux, в которой устранены наиболее узкие звенья в системе безопасности. Доступ пользователей к шлюзу осуществляется через протокол и систему аутентификации SSH, по этому же протоколу происходит поступление небольших объемов входных и выходных, а также управляющих данных. Это обеспечивает хорошую защищенность передаваемых данных, но накладывает ограничения на скорость обмена ввиду больших накладных расходов на криптостойкое кодирование и декодирование информации. Для передачи больших массивов может использоваться другая схема, основанная на пересылке данных, закрытых высокоскоростным алгоритмом шифрования с автоматически генерируемым случайным ключом значительной длины, который передается по хорошо защищенному протоколу SSH.

Сопряжение с прикладными программами

- T-scripting languages
- Интерпретатор комбинаторной логики
- Параллельная реализация языка Рефал+

T-scripting languages

В некоторых случаях удобно иметь специальный язык для оперативного формирования вычислительных задач. На такое средство программирования можно смотреть как на аналог shell'а в системе Unix, из которого можно вызывать базовые системные и пользовательские функции, написанные на языках TC, TFortran и т. д.

По всему миру широко используются такие технологии, как TCL, и данный подход можно рассматривать как распространение этого хорошо зарекомендовавшего себя подхода на кластерные вычисления.

Основными отличиями shell для кластеров являются:

- язык функциональный, имеет ленивую модель вычислений и поддерживает огромное количество параллельно выполняющихся потоков (вызовов функций);
- динамически распараллеливает программы в кластере, используя специальные расширения T-системы (системные функции ядра – комбинаторы).

Интерпретатор комбинаторной логики

Комбинатором называется лямбда-выражение без свободных переменных.

Любую чистую функцию можно представить в виде аппликативного выражения, состоящего только из S и K комбинаторов и примитивных функций.

```
K x y = x

S f g x = (f x)(g x)

I x = x

I = S K K

I x = (S K K) x = (K x) (K x) = x

Y = S S K ( S ( K ( S S ( S ( S S K ) ) ) ) K )
```

Параллельная реализация языка Рефал+

- Реализован новый (непараллельный) рантайм для Рефала+
- Зафиксирован интерфейс для компиляции из Рефала+ в C++
- Смена рантайма не влечет за собой изменения компилятора и Рефал-библиотек
- При параллельной реализации Рефал-рантайма значительное количество кода будет общим с рантаймом T-системы

Средства обеспечения безопасности

- Организация и ограничение доступа
- Распределение ресурсов между пользователями

Организация и ограничение доступа

На управляющем компьютере находится также все основное и дополнительное ПО, которое может исполняться на вычислительных узлах кластера.

Часть функций по администрированию кластера выполняется только локально, причем для этого необходимы права суперпользователя на управляющем компьютере. К этим функциям относятся установка программных пакетов и библиотек, изменение распределения ресурсов для пользователей кластера, назначение администраторов.

В любой момент времени система может находится либо в штатном, либо в технологическом режиме.

В штатном режиме на вычислительных узлах кластера не предусмотрено никакой возможности получить привилегии суперпользователя. В системе нет никаких setuid-задач, и единственным управляющим элементом во время ее работы

является специализированное и системное ПО, которое позволяет пользователям дистанционно запускать свои задачи на счет, а администраторам выполнять ряд дополнительных функций по контролю и распределению ресурсов кластера.

Переход в технологический режим осуществляется в несколько стадий. Сначала производится изменение статуса вычислительных узлов, что отражается в таблицах вычислительных ресурсов. Все вычислительные процессы оповещаются об этом изменении и принимают решения о миграции в то или иное вычислительное пространство. Наименьший приоритет имеет вторичная память, поскольку у нее скорость исполнения программы равна нулю.

По умолчанию система после загрузки находится в штатном режиме. В технологическом режиме имеется возможность администрирования кластера из локальной сети.

Распределение ресурсов между пользователями

Всех пользователей можно условно разделить на две категории по уровню предоставляемого доступа:

- Пользователи, которые могут вызывать лишь некоторые (предопределенные) T-подпрограммы
- Пользователи, которые могут запускать свои собственные T-/MPI-программы

Если в первом случае ограничения на возможности накладываются путем аккуратного программирования удаленного T-интерфейса, то во втором случае необходима тщательная доработка системы квотирования ресурсов в ядре ОС Linux.

Приложения

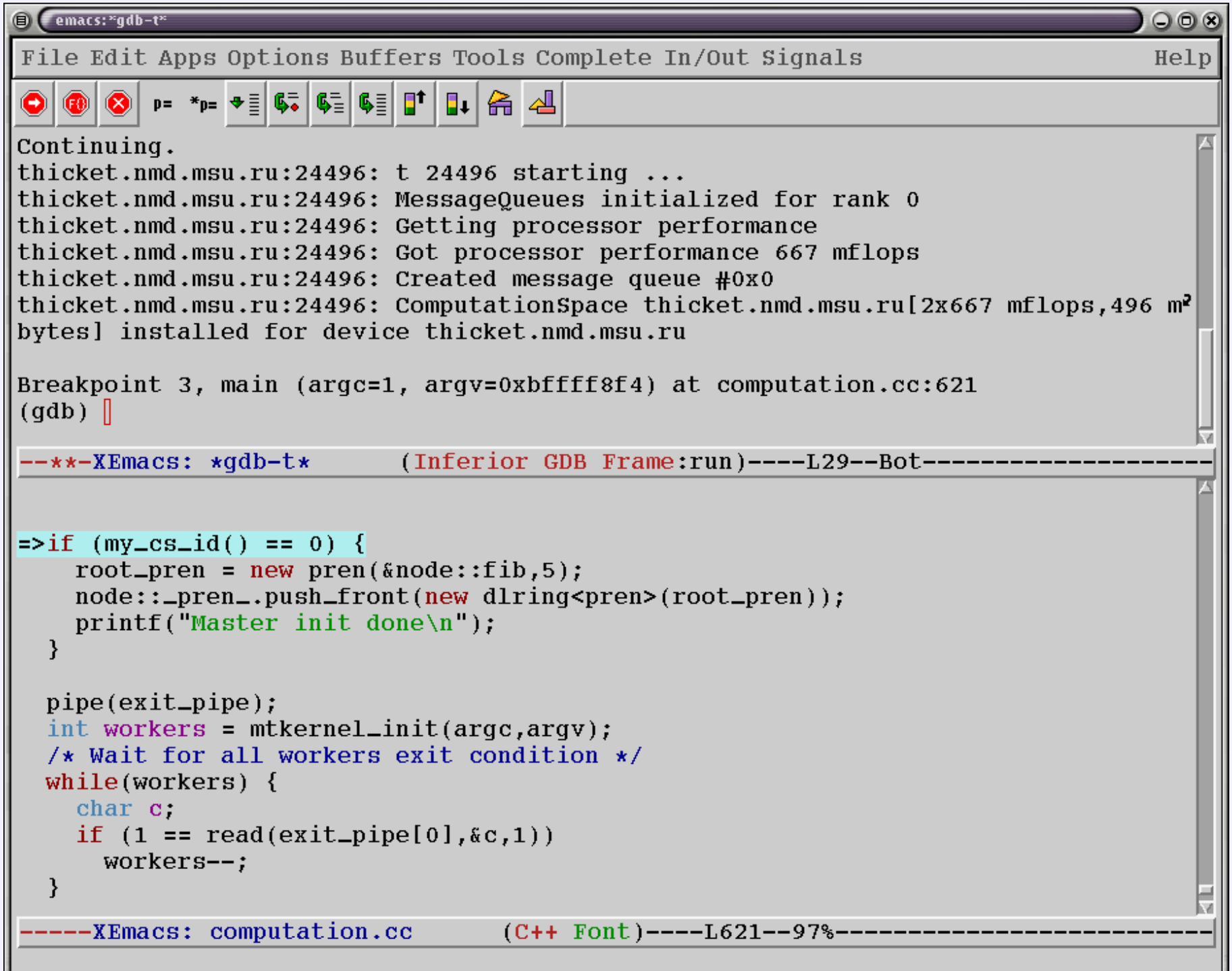
- Отладка и трассировка T-программ
- Примеры вычислительных задач
- Примеры невычислительных задач
- Диаграмма классов T-системы
- Аналоги T-системы

Отладка и трассировка T-программ

T-программа в режиме отладки прописывает в именованный pipe свой PID и ждет изменения некоторой своей переменной, а из среды разработки удаленно запускаются отладчики по SSH, которые присоединяются к процессам и взаимодействуют со средой (визуализируют код программы и воспринимают команды пользователя). Переключение между окнами отладчиков происходит через меню среды разработки XEmacs.

Поскольку XEmacs взаимодействует с отладчиком строго через поток ввода-вывода, то ему не важно, где и как запускается тот или иной отладчик. PID процесса.

Оттестирован также и способ, когда отладчик запускается только в случае программной ошибки.


emacs:*gdb-t*
File Edit Apps Options Buffers Tools Complete In/Out Signals Help
p= *p=
Continuing.
thicket.nmd.msu.ru:24496: t 24496 starting ...
thicket.nmd.msu.ru:24496: MessageQueues initialized for rank 0
thicket.nmd.msu.ru:24496: Getting processor performance
thicket.nmd.msu.ru:24496: Got processor performance 667 mflops
thicket.nmd.msu.ru:24496: Created message queue #0x0
thicket.nmd.msu.ru:24496: ComputationSpace thicket.nmd.msu.ru[2x667 mflops,496 m²
bytes] installed for device thicket.nmd.msu.ru
Breakpoint 3, main (argc=1, argv=0xbffff8f4) at computation.cc:621
(gdb)
--**-XEmacs: *gdb-t* (Inferior GDB Frame:run)----L29--Bot----------
=>if (my_cs_id() == 0) {
root_pren = new pren(&node::fib,5);
node::_pren_.push_front(new dlring<pren>(root_pren));
printf("Master init done\n");
}
pipe(exit_pipe);
int workers = mtkernel_init(argc,argv);
/* Wait for all workers exit condition */
while(workers) {
char c;
if (1 == read(exit_pipe[0],&c,1))
workers--;
}
-----XEmacs: computation.cc (C++ Font)----L621--97%----------

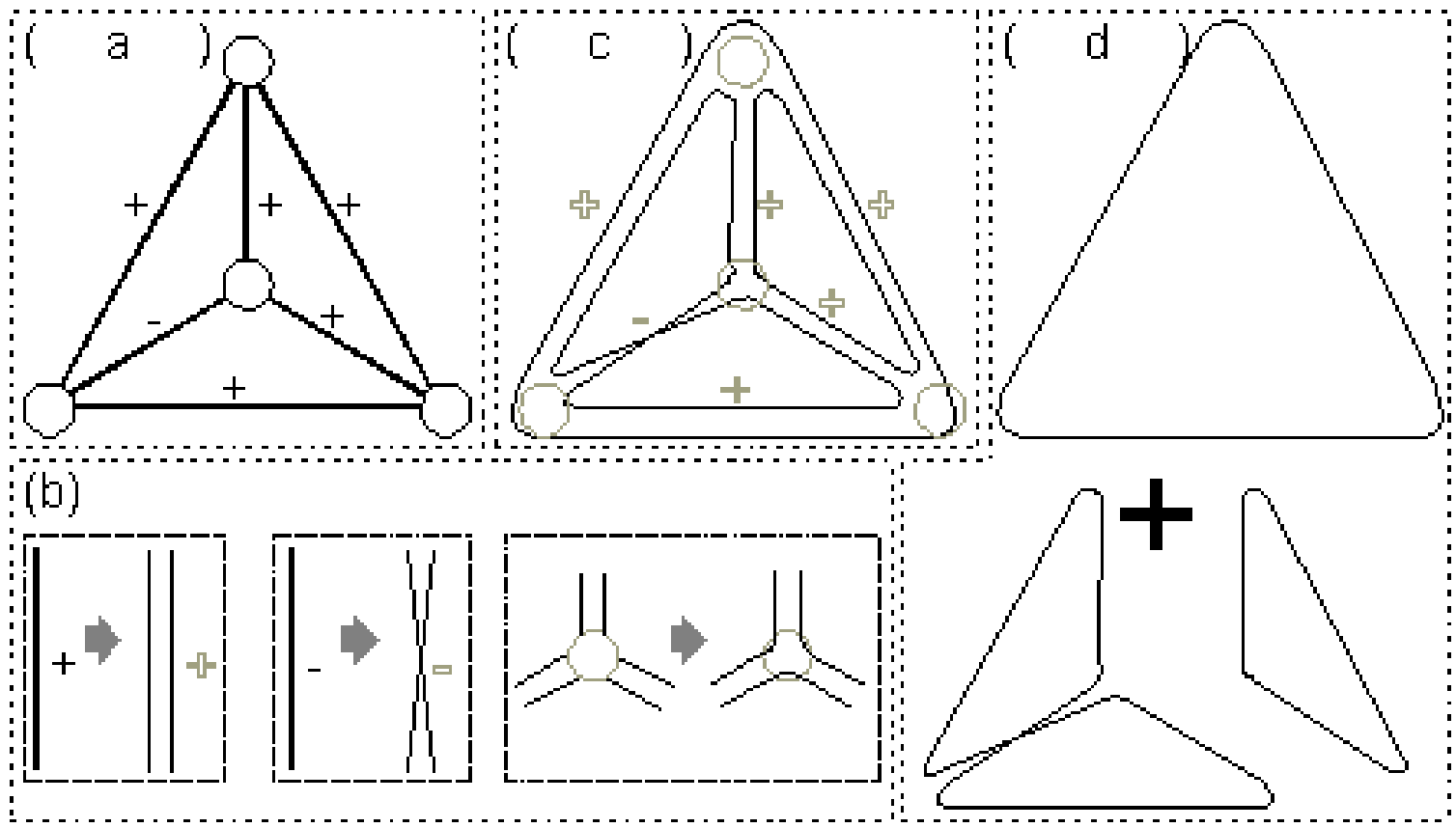
Отладка и трассировка T-программ

Трассировка предполагает запись всех событий (обмен, выборка из очереди, ...), которые влияют на ход работы программы.

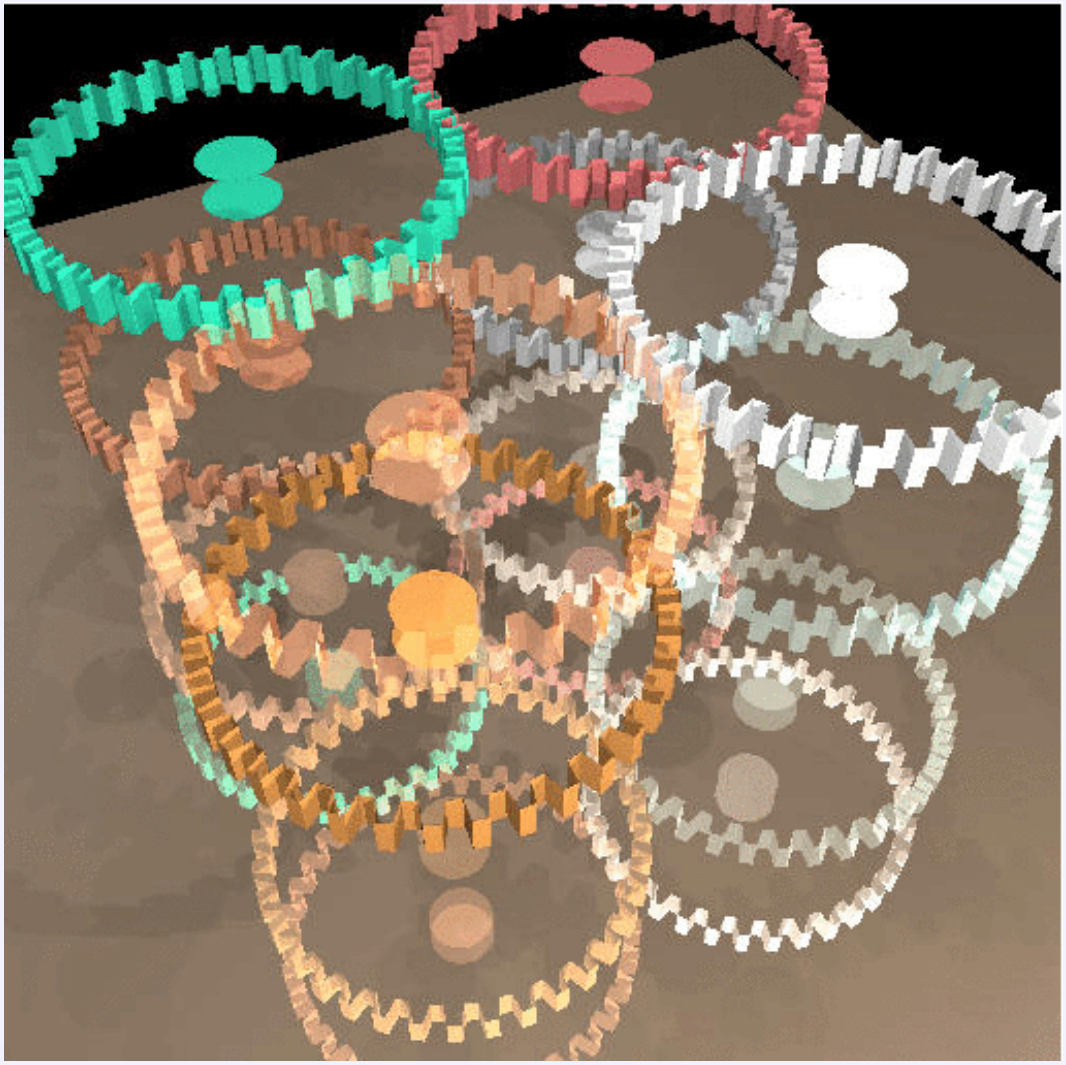
T-система использует систему трассировки MPI, добавляя к ней некоторую дополнительную информацию (возникающую на SMP-уровне)

В книге Хоара «Взаимодействующие последовательные процессы» приводится (несложная) теорема о том, что если на входы каждого последовательного процесса при двух прогонах системы приходят одни и те же события, то поведение системы в целом будет также одинаковым.

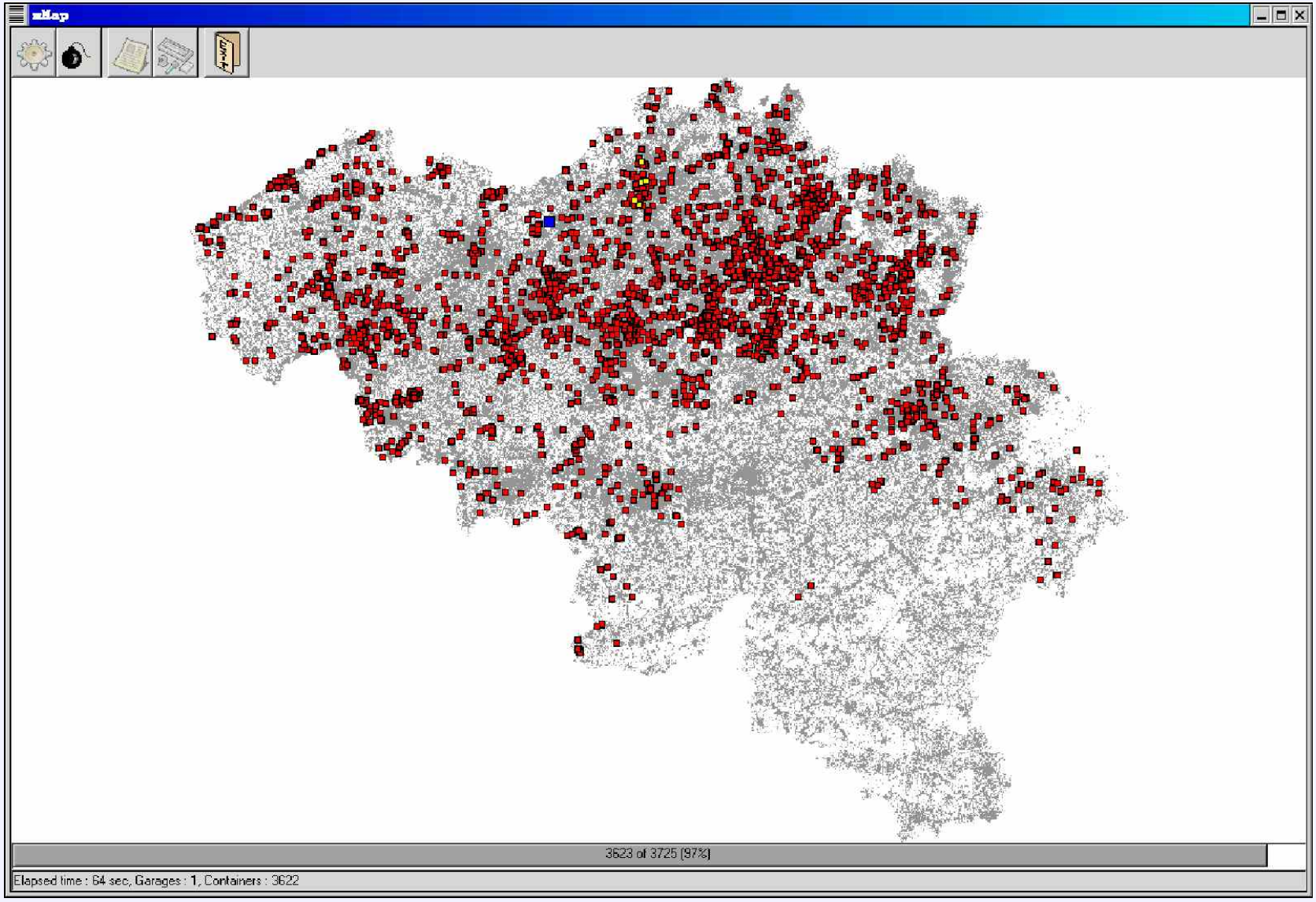
Примеры вычислительных задач



Примеры вычислительных задач



Примеры невычислительных задач



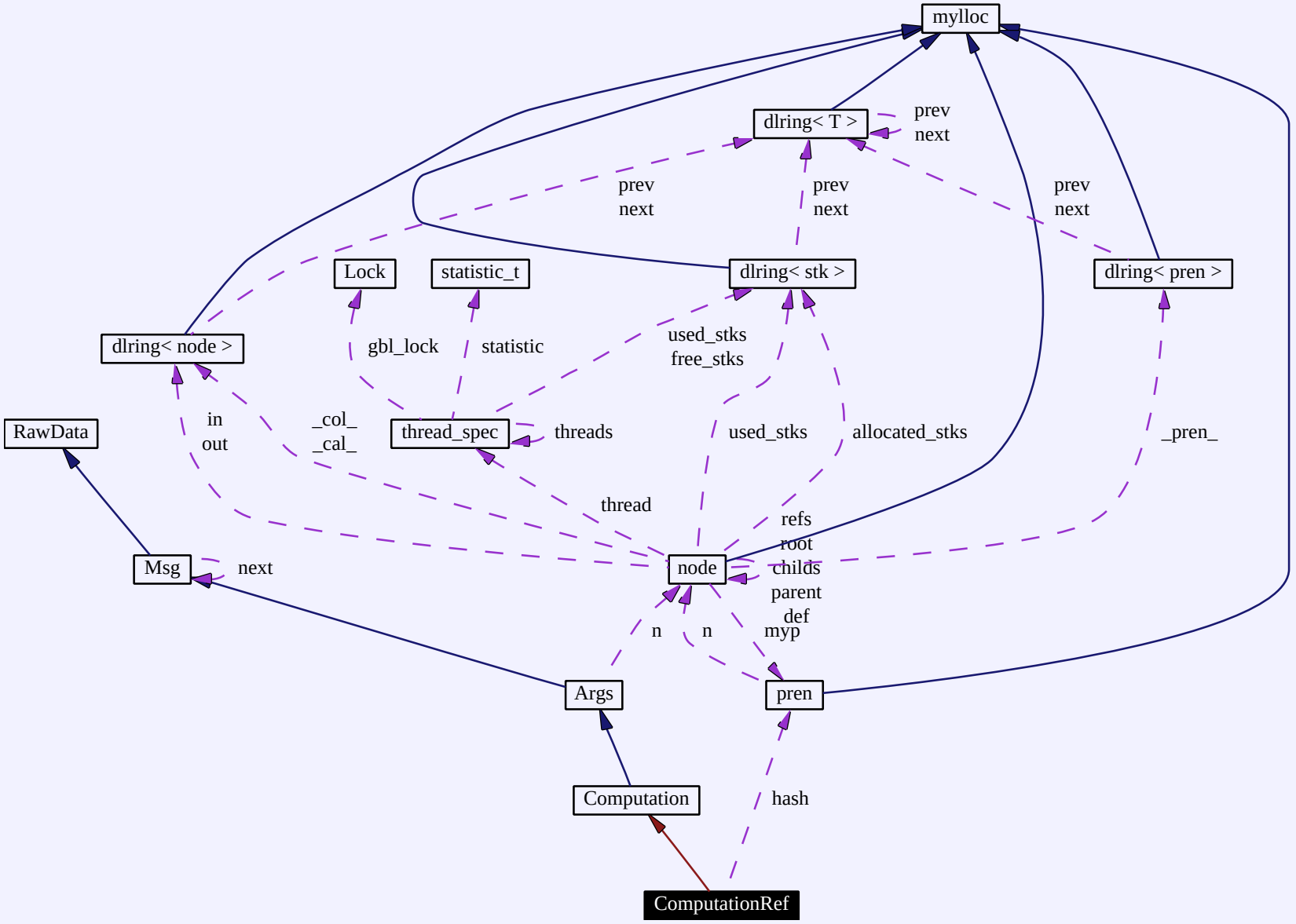
Примеры невычислительных задач

GQL – Graph Query Language

Различные операции над графами:

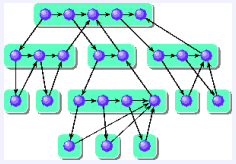
- Вычисление значения рекурсивной функции на деревьях
- Инкрементальное вычисление функций (используя мемоизацию)
- Поиск подграфа в графе

Диаграмма классов T-системы



Аналоги T-системы

Cilk



```
#include <cilk.h>
#include <cilk-lib.h>
#include <stdlib.h>
#include <stdio.h>

cilk int fib (int n)
{
  if (n < 2)
    return (n);
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
    return (x + y);
  }
}
```

```
cilk int main (int argc, char *argv[])
{
  int n, result;
  if (argc != 2) {
    fprintf(stderr, "Usage: fib [<cilk options>] <n>\n");
    Cilk_exit(1);
  }
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf("Result: %d\n", result);
  return 0;
}
```

Аналоги T-системы

SISAL

- Язык с однократным присваиванием
- Компиляция в две стадии:
  - Генерация промежуточного года на C/FORTRAN с оптимизацией
  - Генерация выполняемого файла (стандартный C/FORTRAN компилятор)
- Пример реализации быстрой сортировки на языке SISAL:

```
define  Main

type Info = array[ integer ]

function Main (Data : Info returns Info)

  function Split (
      Data : Info returns Info, Info, Info)
    for E in Data
    returns array of E when E < Data[ 1 ]
            array of E when E = Data[ 1 ]
            array of E when E > Data[ 1 ]
    end for
  end function

  % routine body
  %

  if array_size( Data ) > 2 then
    let
      L, Middle, R := Split( Data )
    in
      Main( L ) || Middle || Main( R )
    end let
  else
    Data
  end if

end function % quicksort
```

Аналоги T-системы

GPH

```
import Parallel

pfc :: Int -> Int -> Int -> Int
pfc x y c
   | y - x > c = f1 `par`
                 (f2 `seq` (f1+f2))
   | x == y    = x
   | otherwise = pf x m + pf (m+1) y
   where
      m  = (x+y) `div` 2
      f1 = pfc x m c
      f2 = pfc (m+1) y c

pf :: Int -> Int -> Int
pf x y
   | x < y     = pf x m + pf (m+1) y
   | otherwise = x
   where
      m  = (x+y) `div` 2

parfact x c = pfc 1 x c
```