ISBN 5-94052-066-0 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ. ПЕРЕСЛАВЛЬ-ЗАЛЕССКИЙ, 2004

УДК 519.68

С. М. Абрамов, В. А. Роганов

Динамическая специализация как средство оптимизации распределенных вычислений и как метод создания адаптивных сервисов для GRID-систем

Аннотация. Специализация является одним из важнейших и фундаментальных методов оптимизации сложных вычислительных алгоритмов. Частичные вычисления, смешанные вычисления, суперкомпиляция и другие подходы [1, 2, 4–6, 9] к специализации подчас позволяют изящно решать сложные практические задачи не только оптимизационного характера. Данная работа посвящена использованию динамической специализации для оптимизации распределенных вычислений и создания адаптивных сервисов в GRID-системах. В качестве средства распараллеливания вычислений рассматривается T-система [11, 12]. Предполагается, что читатель знаком с основными понятиями следующих областей теории и практики программирования: специализация программ [1, 2, 4, 9], GRID-технолодии [17, 18], T-система [11, 12].

Ключевые слова и фразы: Специализация, T-система, оптимизации распределенных вычислений, GRID.

1. Введение

Известно достаточно большое число подходов к реализации специализации: частичные вычисления, смешанные вычисления, суперкооомпиляция и другие [1, 2, 4–6, 9]. Они могут использоваться как технологии построения высокоэффективных программ по их функциональной спецификации. В настоящее время в мире ведутся интенсивные работы, направленные на практическое использование разработанных методов специализации и их внедрение в реальные интерпретаторы, компиляторы и среды исполнения программ. Простейшие формы специализации, которые являлись настольным инструментом

Работа выполнена при поддержке Суперкомпьютерной программы «СКИФ» Союзного государства, проекта РФФИ № 02-01-81024Бел2002_а и программы фундаментальных исследований Президиума РАН «Разработка фундаментальных основ создания научной распределенной информационно-вычислительной среды на основе технологий GRID» .

при программировании на таких языках, как Лисп, успешно перекочевали в мир объектно-ориентированного программирования [15, 16], изменив свое обличье и название. Идеи, развиваемые в этой статье, позволяют говорить о возможности использования инструмента специализации для создания и автоматической адаптации виртуальных сервисов под доступные конфигурации GRID-систем, изменяющиеся нужды и запросы пользователей.

Данная работа посвящена использованию динамической специализации для оптимизации распределенных вычислений и создания адаптивных сервисов в GRID-системах.

В качестве базового средства распараллеливания вычислений в данной работе рассматривается T-система [11, 12].

Предполагается, что читатель знаком с основными понятиями таких областей теории и практики программирования, как специализация программ [1, 2, 4, 9], GRID-технолодии [17, 18], T-система [11, 12].

В рамках развития T-системы понятие специализации возникало неоднократно [13]. Это вполне естественно, так как параллельная редукция графов, лежащая в основе подобных вычислительных систем, уже сама по себе обладает «врожденной» способностью к специализации и частичным вычисления. При реализации в T-системе мемоизации и поддержки контрольных точек было обнаружено еще одно возникающее при этом новое качество: сохраненный на диске образ T-программы, которая уже использовалась для обработки серии вычислительных запросов, может быстрее обрабатывать типичные запросы, поскольку мемо-таблица уже содержит в себе множество часто встречающихся записей вида

$$(\texttt{функция}, \texttt{аргументы}) \longrightarrow \texttt{значение}$$

Затем, когда мы обобщаем модель T-функций на *совокупность T-приложений*, мемоизация и сохраняемость вычислений естественно превращается в разного рода объектные кэши. Поясним это подробнее. В T-системе базовым средством для обмена данными является *неготовое значение*, и оно же является средством для синхронизации доступа между T-функциями: *поставщиком* и *потребителями* [11, 12]. Если теперь в этом высказывании сделать следующие замены:

- «неготовое значение» → «объект во внешней памяти»;
- «T-функция» → «T-приложение (или любое другое приложение)»;

- «T-система» → «T-GRID»;

то мы получим обобщение модели T-системы, способное вместить в себя как сохраняемость данных во внешней памяти, так и интероперабельность с любыми другими приложениями через объектный кэш. Наконец, при попытке построения на базе T-системы эффективной GRID-системы было обнаружено еще одно свойство, которое очень точно соответствует модели T-контекста[1] и может быть истолковано как динамическая адаптация (или, в нашей терминологии, специализация) к вычислительной конфигурации метакластера (то есть специализация по конфигурации вычислительной системы). Более того, весь процесс организации счета может трактоваться как многократная и, в некотором смысле многостадийная, специализация. При этом автоматически и очень красиво решаются вопросы организации непрерывного мониторинга ресурсов вычислительной среды, динамической загрузки системных расширений, эффективной доставки кода T-приложений, предварительный «разогрев» и быстрый «поджиг» вычислений (последнее свойство очень актуально в системах оперативной обработки запросов).

В качестве дополнительных преимуществ построенной схемы легко достигаются приоритетное планирование вычислительных процессоров и коммуникаций (как ре-специализация конкурирующих T-программ по новой системе приоритетов), а также отказоустойчивость вычислений в случае сбоев отдельных составляющих распределенной вычислительной системы (как ре-специализация T-системы и последней сохраненной контрольной точки по новой устойчивой конфигурации метакластера).

Все вышеперечисленные результаты достигаются вполне естественным путем; наиболее удивительным моментом является простота реализации. В настоящий момент эти схемы реализуются для T-системы с открытой архитектурой (OpenTS).

---

[1] T-контекст содержит информацию, которая отражает специфику распараллеливания, производимого T-системой над программой. T-контекст является дополнительную информацию к программе на базовом языке, в качестве которого могут выступать, например, C,C++, Fortran.

2. Специализация как метод организации счета в GRID-системах

2.1. Мониторинг как форма вычисления доступной конфигурации. В больших и территориально распределенных вычислительных системах отказы вычислительных и коммуникационных компонент могут возникать достаточно часто. Такие отказы можно рассматривать и учитывать по-разному; в некоторых случаях они даже не скажутся заметным образом на результатах счета (например, при работе алгоритма случайного поиска). Тем не менее, для обычных детерминированных вычислений это не так. Можно сказать, что фактор «ненадежности» неизбежно присутствует в процессе счета.

Если мы не выделим эту ненадежность, то она может серьезным образом повлиять на основной счет — вычисления либо аварийно прекратятся, либо пользователю самому придется заниматься обработкой исключительных ситуаций и аварий. *Остаточной программе* (термин из теории специализации), после такого выделения «ненадежности», соответствует надежная (в каждый момент времени) вычислительная конфигурация.

Практически нам нужно:

(1) В каждый квант времени знать, какое множество вычислительных узлов устойчиво работает и имеет надежную связь между собой. Это можно трактовать как вычисление доступной вычислительной и коммуникационной конфигурации.
(2) Быть готовыми к повтору вычислений, если произошел отказ одной из физических компонент. Обычно повтор осуществляется при помощи поддержки «контрольных точек».

Подсистема мониторинга в реализации Т-системы с открытой архитектурой (OpenTS) осуществляет запуск Т-исполнителей на доступных узлах и непрерывное наблюдение за этими узлами метакластера. Для реализации этого в Суперпамяти[2]выделено по одной специальной ячейке на каждый вычислительный узел, в которую заносится и становится доступной всем вычислительным узлам описание ресурсов (статус) каждого вычислительного узла.

При этом в каждый квант времени становится известным множество устойчиво работающих узлов и их загруженность, а также множество отказавших узлов. Вновь запущенные Т-программы, а также все Т-программы, которые использовали для счета ставшими недоступными узлы, подлежат запуску/восстановлению из контрольных точек на новой устойчивой конфигурации распределенной системы.

Запуск и восстановление производятся в «лениво-отделяющейся копии» Т-системы при помощи специального вызова `tfork`, механизм работы которого мы подробно опишем ниже.

2.2. Адаптация к архитектуре и конфигурации вычислительной системы. Текст высокопроизводительной программы обычно преобразуется в исполняемый бинарный код для исполнения на конкретной аппаратуре с помощью оптимизирующих компиляторов, так как несмотря на значительные успехи в области динамической (on-the-fly) компиляции для таких языков, как Java и C# пока еще не достигнут уровень скорости исполнения традиционных компилируемых языков.

Ключевыми моментами, облегчающим создание гетерогенных высокопроизводительных систем является относительно небольшое количество реально и широко используемых программно-аппаратных

[2]Суперпамять — это специальная реализация модели распределенной общей памяти, которая входит в базовый уровень реализации Т-системы с открытой архитектурой (OpenTS). Она используется практически для всех видов взаимодействий в распределенной среде (обмен данными и синхронизация доступа, общий пул заданий, доступность и степень загруженности узлов).

платформ на каждом витке технологии, а также относительную компактность вычислительных ядер[3] алгоритмов. Первый момент позволяет просто иметь в наличии набор компиляторов и кросс-компиляторов для поддерживаемых платформ, а второй — существенно ускорить доставку задачи, если доставлять только исполняемый код вычислительного ядра.

Мы можем рассматривать компиляцию как специализации текста программы для исполнения в определенном программно-аппаратном окружении. В отличие от классических схем [2, 3, 7, 10] реализации компиляции за счет специализации мы здесь используем то обстоятельство, что, зная вычислительную конфигурацию, можно выбрать наиболее оптимальные параметры, с которыми и скомпилировать вычислительное ядро; также можно предложить согласованную с этими параметрами эффективную стратегию для распределенного счета: на сколько частей дробить задачу, на какие вычислительные узлы в первую очередь посылать подзадачи и так далее. То есть, вполне содержательную часть работы Т-программы можно проделать заранее, *зная лишь* вычислительную конфигурацию системы.

2.3. Динамическая загрузка и инициализация кода и данных. Динамическую загрузку кода и инициализацию данных можно формально рассматривать как специализацию среды исполнения по вычислительной задаче. Такое использование идеи специализации хорошо известно (см., например, [14]), так что мы не будем на нем останавливаться.

Заметим лишь, что динамическая загрузка в метакластерах кажется значительно более перспективной, чем копирование уже скомпонованных программных модулей: объем доставляемого кода значительно сокращается вследствие того, что стандартные системные библиотеки и *различные библиотеки вычислительных алгоритмов* могут *уже* присутствовать на каждом вычислительном узле (стадия их инсталляции, которую лучше всего выполнять автоматически, также можно формально выделить в стадию адаптивной инициализации вычислительного окружения).

---

[3] Под вычислительным ядром алгоритма обычно подразумевают ту его часть, которая производит большую часть реальной вычислительной работы и в первую очередь подлежит распараллеливанию. Как правило, эта часть состоит из одного или нескольких циклов и обрабатывает данные, которые специально подготовлены в наиболее удобном для быстрой обработки виде на стадии предобработки.

Еще одно существенное замечание состоит в том, что глубокую инициализацию лучше по возможности отделить от динамической загрузки (также выделить в отдельную стадию), и выполнять при помощи *мемоизации* (рассматривается ниже) и прочих средств Т-системы. При ином подходе можно получить накопление побочного эффекта из-за вероятного наличия ошибок в сложном коде (инициализация и финализация C++-кода обычно реализуется при помощи конструкторов и деструкторов статических объектов; если при этом производятся сложные вычисления, то велика вероятность накопления таких эффектов, как, скажем, утечка памяти).

2.4. Мемоизация как форма динамической специализации. Явное конструирование объектов — не единственный способ сохранения результатов вычислений. Из теории функционального программирования широко известен принцип мемоизации (или табулирования) функций. И этот принцип может быть применен не только для вычислений. Кэширование распределенных данных, например, является мемоизацией функции доставки копий данных с места расположения оригиналов.

Мемоизация имеет то преимущество перед явным конструированием и запоминанием объектов, что ее можно реализовать «прозрачно». Например, в Т-системе если функция имеет в Т-контексте своего определения атрибут memo, то Т-система автоматически производит табулирование результатов этой функции во время работы, причем она это может делать на всех узлах кластера, чтобы избежать повторного вычисления данной функции с теми же аргументами на других вычислительных узлах.

Программа, которая уже произвела обработку серии вычислительных запросов, может быстрее обрабатывать типичные запросы, поскольку мемо-таблица уже содержит в себе множество часто встречающихся записей вида

$$(\texttt{функция}, \texttt{аргументы}) \longrightarrow \texttt{значение}$$

То есть при работе Т-программы, происходит автоматическая динамическая специализация мемоизируемых функций на типичные запросы.

Далее заметим, что в GRID-системах может одновременно выполняться несколько Т-приложений. Если сделать мемоизацию общей для всех них, то принцип динамической специализации сохраняет свою силу.

Действительно, для мемоизации нет никакой разницы, является ли она частной для каждого вычислительного процесса или общей для всех процессов; в последнем случае ее эффективность может быть даже выше по очевидным соображениям.

Внешняя мемо-таблица может быть успешно отождествлена с объектным кэшем или базой данных. Работа с этим кэшем может быть реализована таким образом, что будет учитываться старение (на основе различных критериев актуальности) информации, и его значение будет с разумной периодичностью обновляться. Другими словами, достигается возможность сочетания прозрачной эффективной работы при автоматической поддержке актуальности информации.

2.5. Примитив `tfork`: создание специализируемой копии среды вычислений. Примитив T-системы `tfork` является такой же надстройкой над стандартной функций ядра `fork`, как и сама T-система является надстройкой над стандартными вычислительными средствами.

Эта функция сводится к «отложенному копированию» среды исполнения, и после ее вызова у нас возникает логически новая копия T-программы (в смысле данных), но использующая те же самые коммуникационные каналы (как в случае `fork` разделяются файловые дескрипторы).

Несмотря на кажущуюся экзотику, именно так и запускаются приложения в ОС Unix: сначала выполняется системный вызов `fork`, а затем, после возможных настроек в отделившейся среде исполнения, деструктивный exec.

В рамках современной концепции T-Суперструктуры, T-GRID выглядит как операционная метасистема[4] над низлежащими операционными системами. При этом имеют место следующая аналогия: начальному процессу `init` соответствует T-процесс `tinit`, который в отличие от первого укоренен в нескольких ведущих (возможно, всего в одном) узлах кластера. Он обладает замечательным свойством работать на неустойчивой, постоянно изменяющейся доступной вычислительной конфигурации, так как *не производит никакой вычислительной работы* (также как и `init`), и занимается только порождением *временно устойчивых* вычислительных под?систем при помощи

[4] Термин «метасистема» используется в точном соответствии с его классической [8] трактовкой.

вызовов `tfork`. Эти временно устойчивые под?системы в некотором смысле могут рассматриваться как аналоги `shell` (и могут именоваться `tshell`), так как их задача сводится к менеджменту каналов и загрузке кода приложений. При аварии в том или ином сеансе `tshell`, `tinit` должен терминировать процессы сеанса на всех вычислительных узлах, а также заняться восстановлением из контрольной точки во вновь образованной (опять же при помощи вызова `tfork`) копии доступной в данный момент вычислительной среды.

Возникает вопрос: почему `tfork` является в этой схеме необходимым? Почему нельзя выполнять независимые вычислительные подзапросы в пределах одного T-приложения? Ответ: по соображениям надежности и безопасности. Поскольку код T-системы выполняется в том же пространстве, что и код приложения (что обусловлено вопросами эффективности; имеется лишь разнесение служебных и пользовательских компонент T-переменных по адресам), то это может привести к тому, что аварийное завершение одного из подзапросов повредит саму систему.

Второй вопрос — почему `tfork` является в этой схеме достаточным? Ведь при вызове `fork` наследуются дескрипторы, и *мы намеренно хотим*, чтобы наследовались дескрипторы коммуникационных каналов. То есть мы хотим, чтобы несколько «отфоркованных» T-программ разделяли одни и те же коммуникации. Несмотря на то, что при этом уже нет полной независимости, она, как это ни странно, является достаточной для большинства случаев. Действительно, если мы используем классическую схему мультиплексирования каналов, используя уникальные `tpid` для каждой T-программы, то у нас не должно произойти путаницы с доставкой сообщений. Если даже произойдет непредвиденное завершение одной из T-программ, то недоставленные сообщения будут просто проигнорированы. Кроме того, информацию о загрузке процессов (то есть часть того, что проходит через обработчик контрольных активных сообщений) вообще можно пропускать в единственном числе только через главный процесс.

При этом процессы `tinit` и `tshell` могут очень эффективно отслеживать1 процесс счета; это очень подходящие вычислительные процессы для подсчета и отображения разного рода статистики.

Интересно также отметить, что разделение коммуникационных каналов является хорошим средством для поддержки эффективного планирования порожденных процессов. Действительно, регулируя

потоки данных, зачастую можно реально регулировать вычислительную активность процессов в распределенной системе.

Вопросы обычной планировки процессов или даже схемы планировки в T-системе в рамках небольшого мультикомпьютера являются более-менее традиционными и могут решаться стандартными средствами, хотя и не всегда самым лучшим образом.

Что же касается метакластеров и одновременной работы нескольких T-приложений, то здесь уже классические схемы оперативного управления заданиями становятся неэффективными. Действительно, невозможно просто так взять и остановить тот или иной вычислительный процесс, если последний уже покинул породившее его вычислительное пространство[5] и, что вполне вероятно, неоднократно мигрировал с одного вычислительного узла на другой в процессе работы в поисках наименее загруженного процессора.

Тем не менее, задача управления приоритетами заданий стоит не менее актуально: нужно уметь быстро «снять» задачу, а также желательно иметь возможность динамически изменить приоритет (реализуя, тем самым, вытесняющую многозадачность).

Очень простой путь — реализовать планирование на уровне процессов `tinit` и `tshell`. Поскольку они контролируют коммуникации, то могут влиять на очередность доставки активных сообщений. Совершенно ясно, что этого влияния может быть достаточно для решения задач планирования.

2.6. Подзапросы. В некоторых случаях базовый запрос к информационной системе служит основной для серии последующих подзапросов. В этом случае, естественным образом используя `tfork`, мы получаем возможность проспециализировать вычислительный метапроцесс по базовому запросу и обработать серию последовательных подзапросов более эффективно.

2.7. Адаптивная специализация программы на основе пост-анализа. Наконец, зная результаты счета (полученные результаты и время, на них затраченное), мы можем оптимизировать тем или иным образом исходный код программы. Очень простой путь — использование условной компиляции. Более продвинуты схемы могут быть реализованы с использованием скриптов, которые будут в

[5] Вычислительным пространством для вычислительных процессов обычно называют место, в котором последние обладают способностью вычисляться. Обычно это аппаратура (вычислительный узел) и некоторая программная среда.

нужный момент времени генерировать проспециализированный (по итогам известной статистики) исходный код, который затем по той же схеме:

`[кросс]компиляция` ⟶ `доставка` ⟶ `динамическая загрузка`

станет доступным для вычислений в распределенной среде.

3. Заключение

Отрадно заметить, что работы в направлениях организации супервычислений и развитии методов специализации ведутся в ИПС РАН уже в течении не одного десятка лет. В данной статье специализация рассматривается не только как инструмент для оптимизации программ, но и как механизм создания специализированных вычислительных конфигураций в контексте кластерных и GRID-систем.

Разумеется, рамки данной статьи не позволяют рассмотреть всех известных практически интересных случаев, когда прямое или косвенное применение специализации может дать существенный выигрыш в эффективности высокопроизводительного и распределенного счета. К оставшемуся «за рамками» можно отнести частичное применение Т-функции к аргументам (карринг), а также содержательную параллельную специализацию, такую, например, как частичные вычисления (Partial Evaluation).

Отметим только, что тема эта достаточно интересная, и неоднократно поднимаемая в мировой научной литературе. Многое в этом области, скорее всего, даже еще не сформулировано, что объясняется относительной «молодостью» самих распределенных супервычислений в GRID-системах.

Необходимо производить дальнейшие исследования и апробации в этом направлении. Предстоит, в частности, произвести измерения коэффициентов ускорения динамически адаптируемых сервисов по сравнению с «неадаптируемыми».

Работы в этом направлении ведутся в рамках суперкомпьютерной программы «СКИФ» Союзного государства (разработка на базе кластеров семейства «СКИФ» экспериментальной GRID-системы), а также при поддержке Российской Академии Наук (программы фундаментальных исследований Президиума РАН «Разработка фундаментальных основ создания научной распределенной информационно-вычислительной среды на основе технологий GRID») и РФФИ (проект № 02-01-81024Бел2002_а).

Список литературы

[1] Ershov A. P., Bjørner D., Futamura Yo., Furukawa K., Haraldsson A., Scherlis W. L. *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation* // New Generation Computing. — **6**, № 2–3. ↑(document), 1, 3

[2] Futamura Yo. *Partial evaluation of computing process — an approach to a compiler-compiler* // Systems, Computers, Controls. — **2**, № 5, с. 45–50. ↑(document), 1, 2.2, 3

[3] Jones N. D., Sestoft P., Søndergaard H. *An experiment in partial evaluation: the generation of a compiler generator* // Rewriting Techniques and Applications Lecture Notes in Computer Science ред. Jouannaud J.–P. T. **202**: Springer-Verlag, 1985, с. 124–140. ↑2.2

[4] Wadler P. *Deforestation: transforming programs to eliminate trees* // Theoretical Computer Science. — **73**, с. 231–248. ↑(document), 1, 3

[5] Futamura Yo., Nogi K. *Generalized Partial Computation* // Partial Evaluation and Mixed Computation ред. Bjørner D., Ershov A. P., Jones N. D.: North-Holland, 1988, с. 133–151. ↑

[6] Futamura Yo., Nogi K., Takano A. *Essence of generalized partial computation* // Theoretical Computer Science, 90(1), 1991, с. 61–79. ↑(document), 1

[7] Romanenko S. A. *A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure* // Partial Evaluation and Mixed Computation ред. Bjørner D., Ershov A. P., Jones N. D.: North-Holland, 1988, с. 445–463. ↑2.2

[8] Turchin V. F. The Phenomenon of Science. — New York: Columbia University Press, 1977, Русскоязычный вариант: Турчин В.Ф. *Феномен науки: Кибернетический подход к эволюции* М., Наука, 1993, 296 с. ↑4

[9] Turchin V. F. *The concept of a supercompiler* // Transactions on Programming Languages and Systems. — **8**, № 3, с. 292–325. ↑(document), 1, 3

[10] Turchin V. F., Nemytykh A. P. *A self-applicable supercompiler*: Technical Report, № CSc. TR 95-010, City College of the City University of New York, 1995. ↑2.2

[11] Абрамов С. М., Адамович А. И., Коваленко М. Р. *Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ. Пример реализации алгоритма построения изображений методом трассировки лучей* // Программирование, № 25 (2), с. 100–107. ↑(document), 1, 3

[12] Абрамов С. М., Васенин В. А., Мамчиц Е. Е., Роганов В. А., Слепухин А. Ф. *Динамическое распараллеливание программ на базе параллельной редукции графов. Архитектура программного обеспечения новой версии Т-системы* // Научная сессия МИФИ, 22–26 января 2001: Сб. научных трудов. — Т. **2**. — М., 2001. ↑(document), 1, 3

[13] Pimenov S. P. *Parallel execution as partial evaluation* // CC'94. — Canada, 1994. ↑1

[14] Романенко С. А.. *Применение смешанных вычислений к ассемблерам и загрузчикам*: препринт.— Т. **27**.—М., ИПМ им. М.В.Келдыша АН СССР, 1983, с. 15. ↑2.3

[15] Chepovsky A. M., Klimov A. V., Klimov A. V., Klimov Yu. A., Mishchenko A. S., Romanenko S. A., Skorobogatov S. S. *Partial Evaluation for Common Intermediate Language* // Perspectives of System Informatics // Andrei Ershov Fifth International Conference, Novosibirsk, Russia, July 9–12, 2003: Proceedings of the Conference A.P.Ershov Institute of Informatics Systems, 2003, с. 120–124, Will be printed in LNCS, see also http://www.iis.nsk.su/psi03/programme_e.shtml. (english) ↑1

[16] Goertzel B., Klimov A. V., Klimov A. V. Supercompiling Java Programs: White Paper, 2002, http://www.supercompilers.com. ↑1

[17] The Grid: Blueprint for a New Computing Infrastructure / ред.Ian Foster, Carl Kesselman.—San Francisco, Calif.: Morgan Kaufmann, 1998.— ISBN 0-97028-467-5, с. 550. ↑(document), 1, 3

[18] Foster I., Kesselman C., Tuecke S. *The Anatomy of the GRID* // Int. J. High Perform. Comput. Appl.—**15**, № 3, Also available at http://www.globus.org/research/papers/anatomy.pdf. ↑(document), 1, 3

Исследовательский центр мультипроцессорных систем ИПС РАН

S. M. Abramov, V. A. Roganov. *A dynamic specialization as a tool for distributed computations optimization and as a method of creation GRID-systems adaptive services.* (in russian.)

Abstract. Specialization is a one of the most fundamental method for computational algorithm optimization. Partial evaluation, mixed evaluation, supercompilation and other specialization techniques give us sometimes wonderful solutions for hard practical problems not only from optimization area. This paper is dedicated to dynamic specialization approach to optimization of distributed computations and adaptive services creation for GRID-systems. As a base tool for parallelization we are considering a T-system [11, 12]. We assume a reader has a basic knowledge in the following areas: program specialization [1, 2, 4, 9], GRID-technology [17, 18], T-system [11, 12].