УДК 004.04

СОЗДАНИЕ ФУНКЦИОНАЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ ДЛЯ КЛАСТЕРА

В.А. Резниченко, Е.В. Резниченко

Томский политехнический университет
E-mail: vladimir.a.reznichenko@gmail.com

Показана возможность создания функционального языка программирования, организующего работу с вычислительными ресурсами как кластера в целом, так и ресурсами его отдельных узлов кластера, созданных на основе многоядерных процессоров. Предложен инструментарий, абстрагирующий программиста от низкоуровневых средств организации вычислений в кластерных и многоядерных системах.

Ключевые слова:
Параллельные вычисления, функциональные языки программирования, кластер.

Key words:
Parallel computing, functional programming languages, cluster.

Описывается функциональный язык программирования для создания программ выполняемых на вычислительных кластерах. Данный язык предоставляет иструментарий для использования как ресурсов вычислительного кластера в целом, балансируя нагрузку на его узлы, так и эффективного использования ресурсов отдельного узла, представляющего собой многоядерную систему.

Основное назначение разработанной системы – абстрагировать программиста от специфики взаимодействия узлов кластера между собой и организации вычислений на нескольких вычислительных процессорах/ядрах.

В научно-технической литературе понятие «кластер» употребляется в различных значениях, в частности, «кластерная» технология используется для повышения надежности серверов баз данных или web-серверов. В рамках данной статьи будем говорить только о *вычислительных кластерах*, используемых в качестве доступной альтернативы традиционным суперкомпьютерам.

Классические суперкомпьютеры всегда ассоциировались с чем-то большим: размеры, очень высокая производительность, большой объем памяти и высокая стоимость [1, 2]. Первым серьезным этапом в уменьшении отношения цена/производительность стало появление компьютеров с массовым параллелизмом [3]. Использование серийных микропроцессоров позволило не только гибко управлять мощностью компьютера в зависимости от потребностей и возможностей, но и значительно удешевить их производство. Примерами суперкомпьютеров этого класса могут служить Intel Paragon, IBM SP, Cray T3D/T3E и ряд других.

Единственным способом взаимодействия процессоров в рамках подобных систем было их общение через некоторую коммуникационную среду, объединяющую процессоры в единую вычислительную установку. Например, в компьютерах семейства Cray T3D/T3E все процессоры объединены специальными высокоскоростными каналами в трехмерный тор, в котором каждый вычислительный узел имеет шесть непосредственных соседей. В компьютере IBM SP/2 взаимодействие процессоров идет через иерархическую систему переключателей, потенциально обеспечивающей непосредственное соединение каждого процессора с любым другим.

Однако подобные уникальные решения, с учетом их высоких показателей, обычно недешевы, поэтому и стоимость подобных систем никак не могла быть сравнима со стоимостью систем, находящихся в массовом производстве. Шло время, и прогресс в области сетевых технологий сделал свое дело: на рынке появились недорогие, но эффективные коммуникационные решения. Это и предопределило появление кластерных вычислительных систем, фактически являющихся одним из направлений развития компьютеров с массовым параллелизмом.

Если говорить кратко, то вычислительный кластер – это совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи. В качестве вычислительных узлов обычно используют однопроцессорные компьютеры, двух- или четырехпроцессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаще всего используют стандартные операционные системы: Linux, NT, Solaris. Состав и мощность узлов может меняться даже в рамках одного кластера, давая возможность создавать неоднородные системы. Выбор конкретной коммуникационной среды определяется многими факторами: особенностями класса решаемых задач, доступным финансированием, необходимостью последующего расширения кластера. Возможно включение в конфигурацию специализированных компьютеров, например, файл-сервера, и, как правило, предоставлена возможность удаленного доступа на кластер через Internet.

Подобные возможности позволяют проектировать большое число разнообразных конфигураций кластера. Рассматривая крайние точки, кластером можно считать как пару персональных компьютеров, связанных локальной 10 Мбит/с сетью Ethernet, так и вычислительную систему, создаваемую

в рамках проекта Cplant в Национальной лаборатории Sandia: 1400 рабочих станций на базе процессоров Alpha, связанных высокоскоростной сетью Myrinet [4].

В последнее время направление параллельных вычислений интенсивно развивается [4] и наблюдается появление большого числа продуктов, решающих задачи организации параллельных вычислений. Это связано с появлением и активным использованием многоядерных процессоров, удешевлением кластерных решений для создания больших вычислительных мощностей.

Существующие решения, например, Parallel Haskell, Parallel Lisp, OpenCL ориентированы только на создание программ только для кластерных, либо только для многоядерных систем и они зачастую игнорируют [5, 6] тот факт, что многоядерные и многопроцессорные системы приходят на смену однопроцессорным.

Поэтому работу программы определяет сам программист [7]: он должен явно указать, какие блоки будут выполняться параллельно, а какие нет. Тоже самое происходит с организацией параллельных вычислений на одной машине.

Предпосылки создания

Проведенный анализ существующих решений позволил выделить следующие предпосылки для создания нашей системы:

- сложность существующих средств организации параллельных вычислений;
- узкая направленность развития отрасли на кластерные и многоядерные системы.

Сформулируем требования к создаваемой системе:

- встроенная поддержка управления взаимодействия процессов в пределах одного узла кластера;
- автоматизированное управление выполнением задач кластера;
- создание расширенного инструментария для управления параллельными вычислениями.

С учетом сложности поставленной задачи, прототипом нового языка программирования был выбран LISP-подобный язык Scheme. Это обусловленно тем, что функциональные языки программирования являются интерпретируемыми и, следовательно, нет необходимости перекомпиляции программ написанных на этом языке. Кроме того, большим преимуществом данного класса языков является динамическая типизация, которая также присутствует в системе, дополняя строгое типизирование. Ввиду того, что основной задачей ставилось исследование возможности использования средств организации параллельных вычислений разного уровня, не будем рассматривать систаксис языка, детали реализации и внутреннего представления структур данных, алгоритмов. Вместо этого будут описаны ключевые концепции созданного языка программирования и представлен тот инструментарий, который был разработан для решения поставленной задачи – показать возможность создания гибкой и мощной абстракции для организации параллельных вычислений.

Концепции системы

Первая концепция – связывание переменных. Любой идентификатор, который не является ключевым словом, является переменной. Переменная является именованным контейнером и может содержать некоторое значение. Если переменная содержит значение, то она называется связанной, иначе – неопределенной. В зависимости от контекста использования и логики приложения обращение к неопределенной переменной может как вызывать исключения, так и корректно обрабатываться.

Вторая концепция – окружение. *Окружение* – это контейнер, содержащий переменные для соответсвующего текущей точке исполнения программы контекста. Окружение может быть значением переменной. Для организации независимых вычислений в рамках одного окружения вводится ряд механизмов создания отношений между окружениями.

Новое окружение может быть создано как *расширение* уже существующего окружения (в окружение добавляются новые переменные) и состоит из трех этапов:

- создается *пустое окружение,* которое не содержит переменных;
- переменные и их значения *копируются* из базового окружения в новое;
- в новом окружении создаются переменные, которые, в случае конфликтов имен, используются для исполнения кода программы.

Окружение, от которого наследуется новое, будем называть *родительским*, вновь созданное – *дочерним*, а сам процесс расширения – *наследованием*. При этом не сохраняется связь с родительским окружением, что означает изолированность вновь созданного окружения. В процессе реализации данного механизма мы столкнулись с тем, что, если родительское окружение содержит достаточно большое число переменных, возникают проблемы с производительностью приложения. Решением стало введение понятия *замыкания окружений*, состоящего из трех этапов:

- создается пустое окружение, которое не содержит переменных;
- в созданном окружении помещается информация о родительском окружении;
- во вновь созданном окружении создаются новые переменные.

Механизмы расширения и замыкания не являются равноценными, поскольку накладываю определенные ограничения на возможность параллельного исполнения кода. Так, расширение позволяет исполнять участки кода на разных узлах вычислительно кластера, что обусловлено полной изоляцией вновь созданного окружения. Замыкание возможно для участков программы, которые выполня-

ются на одном узле вычислительного кластера. Эти свойства описанных механизмов использованы для создания специальных конструкций нашего языка программирования.

Третья концепция – контекст создания. Согласно этой концепции любая функция, определенная в системе, связана с окружением, в котором она была создана. Это означает, что это окружение всегда доступно из функции. При этом оно может быть недоступно из остальных участков программы.

Четвертая концепция – гибридная типизация. В зависимости от класса решаемых задач, разработчику доступны преимущества и строгой и динамической типизаций. Так, к примеру, для расчета математических моделей может быть использована строгая типизация. Для задач другого класса, например, подключения экспертных систем для решения отдельного класса задач, может быть использовано динамическое типизирование.

Пятая концепция – минимизация объектов системы. Поскольку программа является интерпретируемой, то имеет смысл выделить некоторые классы объектов (типы данных), которые могут быть ограничены по пространству значений. Возьмем, к примеру, переменные булевого типа. Они имеют всего два состояния (истина и ложь). Нет необходимости создавать несколько копий истинного или ложного значений. Достаточно создать всего два объекта и ссылаться на них на всех участках программы. Помимо уменьшения объема памяти, происходит ускорение ряда операций (например сравнения).

Шестая концепция – специальные формы. Данная концепция определяет основные элементы языка, тот инструментарий, который и делает организацию параллельных вычислений простым и эффективным одновременно.

Специальные формы

Lambda-форма. Используется для создания именованных и анонимных функций. Окружение, в котором была определена функция, запоминается и фиксируется внутри самой функции. Когда инициируется вызов функции, окружение наследуется от окружения, зафиксированного в функции и расширяется значениями аргументов, переданных с функцией при вызове. Созданное таким образом окружение будем называть *окружением вызова*.

Выражения в теле *lambda* будут выполнены в контексте окружения вызова. Если проводить аналогию с классическими императивными языками, этом механиз похож на область видимости внутри функции. Разница заключается в том, что императивные языки хранят все аргументы в регистрах процессора. Результат исполнения последнего выражения в теле возвращается как результат вызова всей функции. Структура вызова:

```
(lambda (formals) expressions)
```

Define-форма имеет два варианта вызова: определяющий переменную и объявляющий функцию, и позволяет создавать и манипулировать переменными окружения. Важно отметить, что первая форма может быть представлена через вторую форму с использованием *lambda*-формы. Структура вызова:

```
(define (formals) expressions)
(define variable expression)
```

Set-форма изменяет значение переменной окружения. Если *expression* указано, оно интерпретируется, и его значение становится значением переменной. В противном случае, пременная становится неопределенной, что может вызвать исключения при попытке изспользовать эту переменную. Разница между двумя формами состоит в том, что *set!* проверяет соотвествие типов и если типы старого и нового значений не совпадают, то вызывается исключение. Переменная должна быть определена на момент вызова выражения, иначе так же вызывается исключение. Структура вызова:

```
(set variable expression)
(set! variable expression)
```

Последовательные формы. К этой группе относятся *serial-форма* и *parallel-форма*. Они позволяют манипулировать порядком интерпретации выражений.

Serial-форма интерпретирует вложенные выражения последовательно – в порядке их следования. Синхронизация в многопроцессорных системах выполняется специальным монитором, рис. 1, *а*. Значение последнего из них возвращается как значение *serial*-формы. Зачастую, явное использование *serial*-формы не требуется, поскольку многие специальные формы имеют последовательный порядок интерпретации выражений. Например, *lambda*-форма. Структура вызова:

```
(serial (expressions))
```

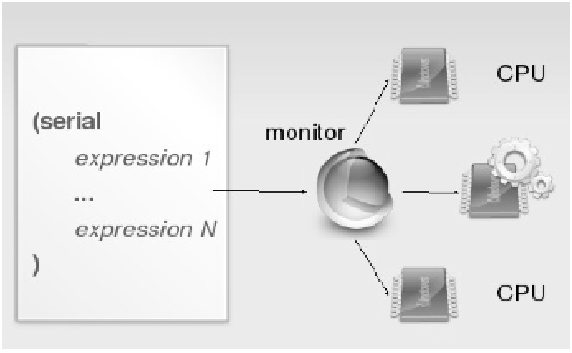

Рис. 1. *Использование монитора для реализации serial-формы*

Parallel-форма позволяет выполнять вложеные выражения параллельно, используя ресурсы многоядерных или многопроцессорных систем одновременно. Управление очередями инструкции осуществляется монитором, рис. 2. Результаты интерпретации выражений не определены, т. к. не определен порядок выполнения выражений. Структура вызова:

```
(parallel (expressions))
```

Atomic-форма позволяет выполнить набор инструкций как одну команду. При интерпретации данного выражения все другие потоки, обслужи-

вающие исполнение программы, приостанавливаются, рис. 3. При выходе из блока приостановленные потоки возобнавляют работу. Это гарантирует монополизацию памяти этим блоком, поскольку все управление передается ему. Структура вызова:

```
(atomic (expressions))
```

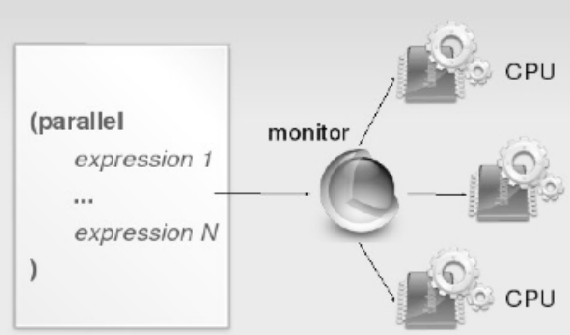

Рис. 2. *Использование монитора для реализации parallel-формы*

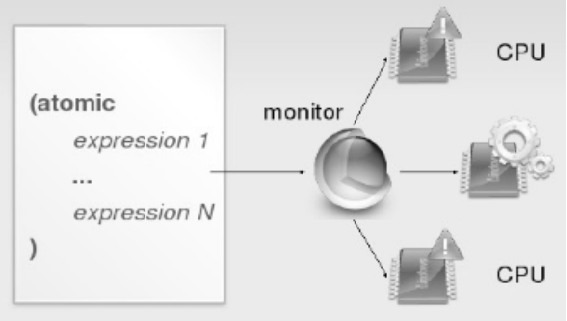

Рис. 3. *Использование монитора для реализации atomic-формы*

Synchro-форма позволяет синхронизировать работу нескольких потоков относительно заданных участков программы, рис. 4. Структура вызова:

```
(synchro (key) (expressions))
```

Как видно из описанного выше, возможно создание простого и гибкого интерфейса организации параллельных вычислений с использованием различных уровней параллеризма. Разработанный прототип позволяет использовать вычислительные мощности кластера и его узлов для создания сложных программ, а также абстрагироваться от низкоуровневых программных уровней.

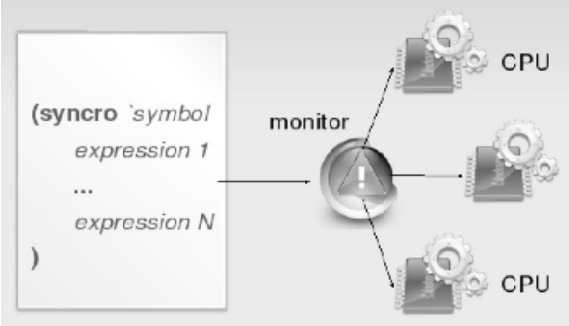

Рис. 4. *Использование монитора для реализации synchro-формы*

Выводы

Рассмотрена возможность создания функционального языка программирования, предоставляющего простой инструментарий организации параллельных вычислений. Показано, что

- возможно объединение различных аспектов параллеризма в терминах одного языка программирования;
- возможно создание упрощенного интерфейса для использования ресурсов разных уровней параллеризма через введение специальных конструкций языка;
- не представляется возможным полностью автоматизировать использование различных ресурсов для организации параллельных вычислений в рамках интерпретируемых языков программирования, поскольку снижается производительность программы.

СПИСОК ЛИТЕРАТУРЫ

1. Корнеев В.В. Параллельные вычислительные системы. – М.: Нолидж, 1999. – 320 с.
2. Хокни Р., Джасхоуп К. Параллельные ЭВМ. – М.: Радио и связь. 1986. – 510 с.
3. Flynn M. Some computer organizations and their effectiveness. – IEEE Trans. Comput., C-21:948, 1972. – 391 p.
4. Gropp W., Sterling T., Lusk E. Beowulf Cluster Computing with Linux, 2nd Edition. – MIT Press, 2003. – 496 p.
5. Головкин Б.А. Вычислительные системы с большим числом процессоров. – М.: Радио и связь, 1995. – 320 с.
6. Амамия М., Танака Ю. Архитектура ЭВМ и искусственный интеллект. – Мир, 1993. – 400 с.
7. Трахтенгерц Э.А. Введение в теорию анализа и распараллеливания программ ЭВМ в процессе трансляции. – М.: Наука, 1981. – 254 с.

Поступила 28.04.2010 г.