Эффективно-переносимое программирование параллельных архитектур

В. П. ИВАННИКОВ, А. Л. ЛАСТОВЕЦКИЙ
Институт системного программирования РАН

УДК 519.68

Ключевые слова: параллельные вычислительные системы, параллельное программирование, инструментальные средства программирования.

Аннотация

Статья представляет собой аналитический обзор основных инструментальных средств параллельного программирования, главным образом, с точки зрения эффективности и переносимости разрабатываемых параллельных программ. Прослеживается, как эволюция инструментальных средств программирования следует за эволюцией архитектуры вычислительных систем.

Abstract

V. P. Ivannikov, A. L. Lastovetsky, Efficiently portable programming parallel architectures, Fundamentalnaya i prikladnaya matematika vol. 4 (1998), № 3, p. 947–974.

The paper is an analytical review of basic parallel programming tools, mainly, from the standpoint of efficiency and portability of developed applications. It is traced how the evolution of programming tools follows the evolution of computer architecture.

В статье формулируются требования на инструментальные средства программирования вычислительных систем. Эти требования основаны на анализе наиболее успешных средств программирования традиционных последовательных скалярных компьютеров. Прослеживается, как эволюция средств программирования следует за эволюцией архитектуры вычислительных систем.

1. Требования к инструментальным средствам программирования

Существует огромное количество языков программирования и поддерживающих их систем программирования, предназначенных для разработки программного обеспечения традиционных последовательных скалярных вычислительных машин, однако, несомненно, Си и Фортран являются наиболее

Фундаментальная и прикладная математика 1998, том 4, № 3, с. 947–974.

популярными среди профессионалов. Какие же свойства этих двух языков сделали их столь успешными и общепризнанными инструментами для разработки последовательных программ?

В то время как Фортран используется, главным образом, для научного программирования, Си более универсален и широко используется для системного программирования. На Си можно программировать в стиле языка Фортран. Более того, любую программу на языке Фортран 77 можно легко конвертировать в эквивалентную Си-программу (в частности, компилятор GNU языка Фортран 77 реализован как такой конвертор). Так что если исключить традиционную привязанность научных программистов к Фортрану, то разумно предположить, что одни и те же свойства делают Фортран столь привлекательным для научного программирования, а Си — для универсального (в особенности, системного) программирования. Поэтому в дальнейшем при обосновании соответствующих требований к инструментальным средствам программирования последовательных скалярных компьютеров мы будем, главным образом, обращаться к языку Си.

Прежде всего, язык Си позволяет разрабатывать высокоэффективное программное обеспечение для каждого конкретного последовательного компьютера традиционной скалярной архитектуры. Это объясняется тем, что Си отражает все основные черты этой архитектуры, влияющие на эффективность программ, а именно, машинно-ориентированные типы данных (`short`, `char`, `unsigned` и т. д.), косвенную адресацию и адресную арифметику (массивы, указатели и их взаимосвязь), а также другие понятия машинного уровня (операции `++` и `--`, операция `sizeof`, операции преобразования типа, битовые поля, поразрядные операции, составное присваивание и т. д.). Традиционная архитектура последовательных компьютеров отражена в языке Си с такой полнотой, которая позволяет для каждого конкретного последовательного скалярного компьютера писать на Си практически столь же эффективные программы, что и на соответствующем языке ассемблера. Другими словами, Си поддерживает *эффективное программирование*.

Во-вторых, язык Си стандартизован как ANSI C [1] и все хорошие компиляторы поддерживают этот стандарт. Это позволяет пользователям писать на Си такие программы, которые, будучи разработанными и оттестированными для одного компьютера, будут без переработки правильно работать на других компьютерах. Другими словами, Си поддерживает *переносимое программирование*. Переносимость Си-программ зависит не только от переносимости их исходного кода, но и от переносимости использованных библиотек. Язык Си обеспечивает особенно высокую степень переносимости для компьютеров, работающих под управлением либо одинаковых операционных систем, либо операционных систем из одного семейства (например, различных версий ОС Unix). Дело в том, что, наряду со стандартными библиотеками ANSI C, большое количество библиотек являются стандартными de facto в рамках соответствующего семейства операционных систем. Если говорить о Unix-системах, то для них вместо «родных» компиляторов широко

используется высококачественный переносимый свободно-распространяемый GNU компилятор [2], позволяющий еще больше повысить степень переносимости исходного Си-кода.

В-третьих, язык Си позволяет пользователю разрабатывать отдельно компилируемые программные единицы, которые могут быть корректно использованы другими пользователями при создании своих программ без знание их внутренностей. Другими словами, Си поддерживает *модульное программирование*.

В-четвертых, язык Си предлагает пользователю *простую* и прозрачную *модель программирования*, обеспечивающую *надежность* разрабатываемых программ. Вместе с модульностью это облегчает разработку сложных и полезных программных систем. Очень непросто найти баланс между эффективностью и ясностью, а также совместить ясность и выразительность. Язык Си представляет собой довольно редкий пример такой гармонии.

Наконец, язык Си поддерживает не только эффективное и переносимое программирование, но и *эффективно-переносимое программирование* последовательных скалярных компьютеров. Мы уже отмечали, что Си отражает все основные черты этой архитектуры, влияющие на эффективность программ. С другой стороны, язык Си скрывает от программиста такие особенности каждого конкретного компьютера, которые не имеют аналогов в других компьютерах этого класса (например, особенности организации регистровой памяти, детали реализации стека, детали системы команд и т. д.). Это позволяет писать переносимые программы, выполняющиеся эффективно на любом конкретном последовательном скалярном компьютере, имеющем высококачественный компилятор и эффективно реализованные библиотеки. Это свойство эффективной переносимости, которое выглядит естественным и достаточно легко обеспечиваемым для последовательных скалярных компьютеров, становится слабым местом в инструментальных средствах программирования параллельных архитектур.

Заметим, что если какой-то инструментарий поддерживает эффективно-переносимое программирование, то он также поддерживает и эффективное, и переносимое программирование. Однако поддержка как эффективного, так и переносимого программирования некоторого класса компьютеров еще не обеспечивает поддержки эффективно-переносимого программирования. Действительно, если инструментарий позволяет написать переносимую программу, а также вручную прооптимизировать ее для каждого компьютера рассматриваемого класса, это не значит, что эта программа будет выполняться эффективно на каждом компьютере этого класса без каких-либо изменений ее исходного кода.

Сформулированные выше требования можно суммировать следующим образом: вряд ли большое число пользователей захотят применять инструментарий, который не позволяет использовать эффективно потенциал производительности их компьютеров, или не позволяет писать программы, которые можно выполнять на других компьютерах, или не позволяет писать модули,

которые могли бы использоваться другими программистами, или основан на изощренной системе понятий, провоцирующей ошибки и делающей трудоемкой разработку и отладку сложных программ, или не позволяет писать переносимые программы, эффективно выполняющиеся на компьютерах целевого класса.

2. Средства программирования векторных и суперскалярных компьютеров

Векторные и суперскалярные компьютеры обладают бóльшим потенциалом производительности, чем последовательные скалярные компьютеры традиционной неймановской архитектуры. Векторные и суперскалярные архитектуры являются развитием этой традиционной архитектуры и включают последнюю в качестве частного случая. Поэтому неудивительно, что инструментальные средства программирования и для этих архитектур базируются на языках Си и Фортран.

Прежде всего, эти инструментальные средства включают оптимизирующие компиляторы с языков Си и Фортран, поддерживающие в совокупности модульное, переносимое и надежное программирование. Ситуация с эффективностью и эффективной переносимостью менее очевидна. Основное специфическое оптимизирующее преобразование, выполняемое этими компиляторами, заключается в векторизации циклов (loop vectorization). Компиляторы пытаются распознать такие циклы, которые фактически представляют собой последовательную форму записи некоторых векторных операций, и оттранслировать их в целевой код наиболее эффективным образом. Например, следующий фрагмент на языке Си

```
double a[64], b[64], c[64];
int i;
Input(a, b);
for(i=0; i<64; i++)
   c[i]=a[i]+b[i];
```

содержит цикл, реализующий сумму двух векторов. На векторном компьютере этот цикл эффективно реализуется с помощью векторных команд. Для суперскалярного компьютера он может быть эффективно реализован путем развертывания цикла с тем, чтобы как можно лучше загрузить все конвейерные устройства компьютера. Некоторые компиляторы выполняют для суперскалярных компьютеров общее распараллеливание циклов, а именно, конвейеризацию циклов (loop pipelining), включающее векторизацию циклов в качестве частного случая.

Хотя оптимизирующие компиляторы языков Си и Фортран способны обеспечить эффективное программирование каждого конкретного компьютера

векторной или суперскалярной архитектуры, от программистов требуются определенные усилия для того, чтобы добиться эффективного выполнения их программ. Дело в том, что наиболее трудной проблемой, которую должен решить оптимизирующий компилятор, является проблема распознавания параллелизуемых циклов. В общем случае эта проблема неразрешима, поэтому каждый компилятор использует свой собственный эвристический распознающий алгоритм. При этом многие циклы, которые могли бы быть распараллелены, не распознаются как параллелизуемые. Более того, зачастую незначительное, чисто формальное изменение цикла, ранее распознаваемого компилятором как параллелизуемый, приводит к тому, что тот же компилятор перестает распознавать его как таковой. Поэтому для того, чтобы написать на Си или Фортране 77 эффективную программу для некоторого векторного или суперскалярного компьютера, программист должен хорошо знать особенности соответствующего оптимизирующего компилятора, с тем чтобы использовать такие формы параллелизуемых циклов, которые распознаются компилятором как таковые. Это приводит к тому, что программа, выполнявшаяся эффективно на одном векторном или суперскалярном компьютере (благодаря ее тщательной подгонке под соответствующий компилятор), может выполняться медленнее после переноса на другие (даже более мощные) компьютеры той же архитектуры. Таким образом, оптимизирующие компиляторы языков Си и Фортран 77, поддерживая (но отнюдь не облегчая) эффективное программирование, не поддерживает эффективно-переносимого программирования векторных и суперскалярных компьютеров.

Другим важным инструментом для эффективного программирования компьютеров рассматриваемого класса являются расширения языков Си и Фортран 77 с помощью библиотек векторных функций. Такие библиотеки эффективно реализуют векторные операции и позволяют пользователю самому писать эффективные программы, не полагаясь на оптимизирующие возможности используемых компиляторов. Векторные библиотеки облегчают эффективное программирования векторных и суперскалярных компьютеров, однако они не обеспечивают переносимости, так как не существует стандартной векторной библиотеки и каждая платформа предоставляет свой собственный, отличный от других вариант такой библиотеки.

Таким образом, Си и Фортран 77 не могут играть для векторных и суперскалярных компьютеров ту же роль, что они играли для последовательных скалярных компьютеров, поскольку они не отражают некоторых существенных черт, общих для всех компьютеров этой более сложной архитектуры, что не позволяет писать на них эффективно-переносимые программы. Поэтому были разработаны расширения этих языков, такие как Фортран 90 [3], Фортран 95 [4] и Си[] [5]. Эти расширенные языки предоставляют высокоуровневые понятия, отражающие специфику векторных и суперскалярных компьютеров, и позволяют пользователю явно выражать такие параллельные вычисления, которые могут быть эффективно реализованы для любого векторного или суперскалярного компьютера. В отличие от компиляторов с

языков Си и Фортран 77, компиляторам с языков Фортран 90, Фортран 95 и Си[] не нужно распознавать участки исходного кода, которые могут быть эффективно реализованы. Им нужно лишь сгенерировать эффективный целевой код для параллельных участков, явно специфицированных программистом.

Инструментальные средства, включающие параллельные расширения Фортрана 77 и Си и их (иногда переносимые) компиляторы, не только позволяют писать более эффективные программы для конкретных векторных и суперскалярных компьютеров, но и поддерживают эффективно-переносимое программирование таких компьютеров.

Рассмотрим в качестве примера следующую Си-программу:

```
#include "common.h"
double A[SIZE][SIZE];
main()
{
   double max;
   register int i, j, k;
   max=0.0;
   for(i=0; i<SIZE; i++)
      for(k=0; k<SIZE; k++)
         max=(max<A[i][k])?A[i][k]:max;
}
```

вычисляющую максимальное значение элементов матрицы A. Оптимизирующий компилятор языка Си для суперскалярного компьютера DEC Alpha 21064, работающего под управлением операционной системы OSF1, не распознает в качестве параллелизуемого цикл, вычисляющий максимум, и генерирует для него соответствующий ассемблерный код, не использующий эффективно конвейерные устройства этого процессора. В то же время следующая написанная на языке Си[] программа

```
#include "common.h"
double A[SIZE][SIZE];
main()
{
   double max;
   max=[?>]((*(double(*)[SIZE*SIZE])A)[]);
}
```

выражает те же вычисления, но с помощью векторных операций.

В языке Си[] вводится понятие *векторного значения*, или просто *вектора*. Вектор определяется как упорядоченная последовательность значений любого, но одинакового, типа (элементов вектора). В отличие от массива, вектор не является объектом данных (то есть областью памяти для хранения значений), но новым видом значения. В отличие от Си, в языке Си[] определено значение

массива, и этим значением является вектор (i-й элемент вектора есть значение i-го элемента соответствующего массива).

Для обеспечения доступа к массиву как к единому целому вводится унарная постфиксная операция [], применимая к адресному значению типа «массив», которая блокирует преобразование операнда к указателю. Таким образом, в Си[] допускается использование в выражениях адресных значений, обозначающих массив, на которые распространяются все правила языка Си обращения с адресными значениями скалярных типов. В частности, выражение `(*(double(*)[SIZE*SIZE])A)[]` обозначает массив из `SIZE*SIZE` элементов типа `double` как единое целое, а значением этого выражения является вектор из `SIZE*SIZE` элементов типа `double`. Унарная операция `[?>]` применима только к векторным операндам и вычисляет максимальное значение среди элементов операнда.

Компилятор языка Си[] для DEC Alpha 21064 реализует векторную операцию `[?>]` таким образом, чтобы как можно эффективнее использовать конвейерные устройства процессора. В результате целевой код, вырабатываемый этим компилятором, выполняется в два раза быстрее, чем код, выработанный оптимизирующим Си-компилятором для рассмотренной выше Си-программы.

Другими словами, языки Си[] и Фортран 90/95 обеспечивают бóльшую эффективность для рассматриваемых архитектур, чем языки Си и Фортран 77.

В то же время перенос рассмотренной Си[]-программы с процессора DEC Alpha 21064 на Cray 1, или UltraSPARC, или Intel i860 не приведет к потере эффективности, так как компилятор языка Си[] для каждой из этих машин обязан реализовывать векторные операции как можно эффективнее.

3. Средства программирования мультипроцессоров с общей памятью

Архитектура мультипроцессора с общей памятью, или SMP-архитектура (от shared-memory multiprocessor) предоставляет больше параллельных возможностей для увеличения производительности, чем векторные и суперскалярные архитектуры. Более того, как правило, процессоры SMP-компьютеров имеют векторную или суперскалярную архитектуру. В дополнение к параллелизму на уровне отдельной команды, предоставляемой векторной архитектурой, и параллельному выполнению команд в рамках единого потока управления, предоставляемого суперскалярной архитектурой, SMP-архитектура предоставляет параллельные нити (threads) управления, выполняющиеся на разных процессорах SMP-компьютера и разделяющие память с другими нитями в рамках одного процесса, а также параллельные процессы, выполняющиеся на разных процессорах, но не разделяющие память с другими процессами, а общающиеся посредством механизма передачи сообщений.

Как и в случае с векторными и суперскалярными архитектурами, наиболее часто используемыми инструментальными средствами программирования являются оптимизирующие компиляторы с языков Си и Фортран 77. В дополнение к оптимизирующим преобразованиям, унаследованным от векторных и суперскалярных архитектур, основное специфическое оптимизирующее преобразование заключается в таком распараллеливании цикла, при котором параллельные нити целевой программы выполняли бы различные итерации цикла. В реальных компиляторах это оптимизирующее преобразование применяется лишь к циклам, распознаваемым как не имеющим зависимостей по итерациям (loop-carried dependencies), то есть к циклам, все итерации которых независимы и могут выполняться параллельно. Таким образом, еще одна трудная и в общем случае неразрешимая проблема распознавания должна решаться оптимизирующими компиляторами с языков Си и Фортран 77 для SMP-компьютеров. Поэтому каждый такой компилятор способен генерировать эффективный целевой параллельный код лишь для исходного последовательного кода весьма специфического вида, а все вместе они не поддерживают эффективно-переносимого программирования SMP-компьютеров.

Более эффективные программы можно написать с помощью библиотек нитей (thread libraries), предоставляющих примитивы для порождения нитей управления и синхронизации их доступа к разделяемым данным. Однако низкий уровень этих библиотек и их машинная зависимость не позволяют использовать их для переносимого программирования SMP-компьютеров.

Средства программирования SMP-компьютеров более высокого уровня, чем библиотека нитей, обеспечивающим большую эффективность, чем оптимизирующие компиляторы с языков Си и Фортран 77, включают расширения этих языков директивами, позволяющими программисту предписывать компилятору либо распараллеливание отмеченного цикла без какой-либо проверки зависимости по итерациям, либо порождение отдельных нитей целевой программы для выполнения отмеченных участков исходной программы. Как правило, такие директивы реализуются в виде псевдокомментариев и различаются для разных компиляторов. Поэтому, в общем случае, директива, одним компилятором интерпретируемая как предписание оптимизирующего преобразования, может интерпретироваться другим компилятором как обычный комментарий. Другими словами, такие расширенные языки, поддерживая переносимость, не поддерживают эффективной переносимости в классе SMP-компьютеров.

Параллельные языки, используемые для эффективно-переносимого программирования векторных и суперскалярных компьютеров (такие как Фортран 90 или Фортран 95), также используются для программирования SMP-компьютеров. Высококачественные компиляторы с этих языков эффективно реализуют векторные операции и другие параллельные конструкции (такие как оператор FORALL языка Фортран 95) на SMP-компьютерах. В отличие от рассмотренных выше средств, эти расширенные языки вместе с их

компиляторами поддерживают эффективно-переносимое программирование SMP-систем.

Наконец, многие инструментальные средства, такие как коммуникационные пакеты PVM (Parallel Virtual Machine), MPI (Message-Passing Interface) или язык High Performance Fortran (HPF), первоначально предназначенные для программирования вычислительных систем с распределенной памятью, также используются для программирования SMP-машин. Фактически эти средства эмулируют для программиста мультипроцессор с распределенной памятью на мультипроцессоре с общей памятью.

Параллельный MP (message-passing)-алгоритм, выразимый на PVM / MPI не всегда столь же эффективно реализуем на SMP-компьютере, как более естественный для SMP-архитектуры PRAM (parallel random access memory)-алгоритм, решающий ту же задачу. Например, для неизменяемого объекта данных, значение которого требуется всем параллельным процессам, в MP-алгоритмах часто используются его копии во всех параллельных процессах, особенно если затраты на передачу данных значительны. Однако реализации PVM / MPI не могут автоматически распознать такой неизменяемый размноженный объект данных в MP-алгоритме и преобразовать его в разделяемый объект данных при реализации алгоритма на SMP-компьютере. Тем не менее, для широкого класса задач реализации PVM / MPI на SMP-компьютерах обеспечивают высокую эффективность.

Что касается языка HPF, то он позволяет компиляторам вырабатывать высокоэффективный код для SMP-компьютеров и поэтому способен поддерживать эффективно-переносимое модульное программирование, по крайней мере, при решении регулярных задач.

4. Средства программирования мультипроцессоров с распределенной памятью

Архитектура мультипроцессора с распределенной памятью, или MPP-архитектура (от massively-parallel processors), являясь, в отличие от SMP-архитектуры, масштабируемой параллельной архитектурой, обеспечивает теоретически неограниченный параллелизм для ускорения вычислений. Программа, выполняющаяся на MPP-машине, представляет собой множество параллельных процессов, взаимодействующих посредством передачи сообщений.

Поскольку MPP-архитектура намного дальше от традиционной последовательной скалярной архитектуры, чем векторные и суперскалярные архитектуры или даже SMP-архитектура, то проблема компиляции последовательного языка программирования, такого как Си или Фортран 77, для MPP-компьютера превращается фактически в проблему автоматического

синтеза эффективной MP-программы, при котором исходный последовательный код используется в качестве спецификации ее функциональной семантики. Хотя в этом направлении и получены некоторые интересные научные результаты (например, в процессе работы над исследовательским компилятором PARADIGM [6]), все они весьма далеки от практического использования. Поэтому основными инструментальными средствами для эффективного программирования MPP-компьютеров в настоящее время являются расширении языков C и Fortran 77 с помощью библиотек функций передачи сообщений, а также языки параллельного программирования высокого уровня.

4.1. Библиотеки передачи сообщений

Было разработано довольно много пакетов передачи сообщений (таких как PVM [7], Nexus [8], PARMACS [9], p4 [10], Chameleon [11], CHIMP [12], PICL [13], Zipcode [14] и т. п.), однако в настоящее время наиболее популярными являются PVM и MPI.

Как PVM, так и MPI позволяют писать не только эффективные и переносимые, но и эффективно-переносимые программы для MPP-компьютеров. Однако по сравнению с PVM MPI имеет два очень важных преимущества.

Во-первых, в отличие от PVM MPI стандартизован как MPI 1.1 [15] и широко реализован в соответствии с этим стандартом. Имеются как высококачественные свободно-распространяемые реализации, поддерживающие MPI 1.1 (такие как LAM MPI [16] от Огайского суперкомпьютерного центра или MPICH [17], являющийся совместной разработкой Аргоннской национальной лаборатории и Университета штата Миссисипи), так и коммерческие реализации практически от всех основных производителей высокопроизводительных компьютеров. Поэтому MPI обеспечивает более высокую степень переносимости программ, чем PVM.

Другое важное преимущество MPI заключается в том, что введенное в нем понятие коммуникатора позволяет разным пользователям независимо писать, компилировать и использовать различные модули единой параллельной программы. Другими словами, в отличие от PVM MPI поддерживает модульное параллельное программирование MPP-компьютеров и, следовательно, разработку параллельных библиотек для этой архитектуры.

Дело в том, что в PVM единственным уникальным атрибутом, характеризующим процесс, вовлеченный в какую-нибудь коммуникационную операцию, является идентификатор процесса, присваиваемый в период выполнения PVM-системой каждому из процессов, образующих выполняющуюся MP-программу. Все остальные коммуникационные атрибуты, такие как группы процессов и тэги сообщений, определяются пользователем и, следовательно, не обязаны быть уникальными во время выполнения программы, особенно если различные модули программы написаны разными программистами. Поэтому у программиста нет гарантированного способа локализовать коммуникации внутри параллельного модуля.

В качестве иллюстрации рассмотрим следующий схематический фрагмент MP-модуля

```
extern Proc();
if(мой идентификатор процесса равен A)
    Послать сообщение M с тэгом T процессу B
Proc(...);
if(мой идентификатор процесса равен B)
    Принять сообщение с тэгом T от процесса A
```

написанный для того, чтобы сначала процесс A послал сообщение M процессу B, затем оба эти процесса вместе с остальными процессами, составляющими программу, выполнили бы внешнюю процедуру `Proc`, а после того как процесс B покинет процедуру `Proc`, он бы принял сообщение, посланное ему процессом A. Однако нет никакой гарантии, что внутри процедуры `Proc` процесс A не посылает сообщений процессу B, а процесс B не принимает сообщений от процесса A. Поэтому, например, процедура `Proc` может перехватить сообщение M, посланное для того, чтобы быть полученным процессом B лишь после выхода последнего из процедуры `Proc`. Использование же определенного пользователем тэга T для установления связи между операцией посылки сообщений и операцией принятия сообщения не может решить эту проблему, так как нет никаких гарантий, что программист, разрабатывавший процедуру `Proc` не использовал тот же самый тэг.

MPI позволяет связать с любой группой взаимодействующих процессов дополнительный уникальный атрибут, называемый коммуникационным контекстом и присваиваемый MPI-системой во время выполнения программы. Этот атрибут играет роль обязательного уникального определяемого системой тэга, которым снабжается любое сообщение и которое может быть использовано для локализации коммуникаций. Так, MPI позволяет следующую безопасную реализацию рассмотренного выше фрагмента MP-программы:

```
extern Proc();
Создать новый коммуникационный контекст C
для группы процессов, включающей процессы A и B
if(мой идентификатор процесса равен A)
    Послать сообщение M с тэгом T
    и коммуникационным контекстом C процессу B
Proc(...);
if(мой идентификатор процесса равен B)
    Принять сообщение с тэгом T
    и коммуникационным контекстом C от процесса A
```

Здесь уникальный коммуникационный контекст C, добавленный к сообщению M, гарантирует, что это сообщение не будет перехвачено внутри процедуры `Proc`, поскольку коммуникационная операция посылки или принятия

сообщения с этим же коммуникационным контекстом C не может появиться в этой процедуре.

Таким образом, в настоящее время MPI является единственной библиотекой передачи сообщений, поддерживающей эффективно-переносимое модульное программирование MPP-компьютеров.

Главный недостаток MPI (как и других коммуникационных пакетов) заключается в низком уровне его системы понятий и параллельных примитивов. Очень непросто написать на MPI действительно сложную полезную и надежную параллельную программу. Поэтому, несмотря на великолепный дизайн, MPI является, по существу, средством параллельного программирования уровня ассемблера для MPP-компьютеров и может быть использован для практического (а не учебного) программирования лишь очень узким кругом высококвалифицированных специалистов.

4.2. Языки параллельного программирования высокого уровня

Хотя с помощью библиотек передачи сообщений и можно писать для MPP-компьютеров высокоэффективные параллельные программы, большинство пользователей находят такое программирование исключительно трудоемким и провоцирующим ошибки. Поэтому для того, чтобы облегчить программирование MPP-компьютеров и поддержать разработку для них сложных и надежных программ, было разработано довольно много систем программирования, базирующихся на параллельных языках высокого уровня. Подавляющее большинство этих систем могут быть помещены в один из двух классов, в зависимости используемой парадигмы параллельного программирования. Системы программирования, базирующиеся на параллелизме данных (data parallelism), и системы программирования, базирующиеся на функциональном параллелизме (task parallelism), покрывают разработку различных классов параллельных программ.

Было разработано несколько функционально-параллельных интерактивных графических систем программирования, таких как HeNCE [18], CODE [19] или Meander [20], которые позволяют пользователю специфицировать коммуникации и синхронизации между подпрограммами на Си или Фортране 77. Так, в системе HeNCE с этой целью используется граф, вершина которого представляет процесс, являющийся экземпляром некоторой процедуры, а направленная дуга представляет коммуникацию или синхронизацию. На основании этого графа система генерирует программу на Си с обращениями к PVM.

Другой подход к функционально-параллельным системам программирования базируется на языках параллельного программирования, таких как Оккам [21], Fortran M [22] или Compositional C++ (CC++) [23].

Язык программирования Оккам реализует разработанную Хоаром модель параллельного программирования, известную как модель взаимодействующих последовательных процессов [24].

Fortran M добавляет к языку Фортран 77 языковые конструкции для создания задач и каналов для передачи сообщений по каналам и для отображения задач на процессоры MPP-компьютера.

Язык CC++ представляет собой расширение C++ с помощью конструкций параллельного блока и параллельного цикла для явной спецификации структурированного параллельного выполнения, а также конструкции порождения процесса для выражения неструктурированного параллелизма. В CC++ имеются синхронизационные переменные и атомарные функции для синхронизации. В CC++ введено ключевое слово `global` для спецификации процессорных объектов, которые соответствуют выполнимым файлам CC++, а также введены глобальные указатели для ссылок между процессорными объектами.

Системы программирования для MPP-компьютеров, основанные на параллелизме данных, базируются на расширенных языках программирования, таких как Fortran D [25], Vienna Fortran [26], HPF, CM Fortran [27], C* [28], Dataparallel C [29], Modula-2* [30] и т. д.

Fortran D является типичным представителем расширяющих Фортран языков параллельного программирования. Фактически Fortran D расширяет последовательный Фортран 77 тремя новыми операторами для спецификации распределения данных. Оператор `DECOMPOSITION` специфицирует массив виртуальных процессоров. Оператор `ALIGN` специфицирует распределения элементов массива данных по этим виртуальным процессорам. Оператор `DISTRIBUTE` специфицирует отображение виртуальных процессоров на физические процессоры целевого MPP-компьютера. Некоторые дополнительные средства языка включают циклы `FORALL`, операции редукции и т. п. Компилятор имеет всю информацию для того, чтобы сгенерировать целевую программу на языке Фортран 77 с обращениями к примитивам передачи сообщений. Класс параллельных программ, которые можно разработать на языке Fortran D, состоит из распараллеленных последовательных программ.

Dataparallel C является расширяющим Си аналогом Fortran D. Он позволяет делать те же вещи, но только в духе языка Си. Он вводит оператор области, который

- специфицирует массив виртуальных процессоров;
- объявляет объекты данных, распределенные по этим виртуальным процессорам;
- специфицирует отображение виртуальных процессоров на физические.

Следуя духу Си, Dataparallel C не вводит глобальных указателей. Так что распределенный массив представляет собой не совокупность элементов, которые могут быть размещены на различных виртуальных процессорах, а распределенный объект данных, каждый компонент которого — массив.

Таким образом, имеется два основных класса систем программирования для MPP-компьютеров, каждый из которых позволяет реализовывать свой класс параллельных алгоритмов. С другой стороны, эффективная решение многих задач требует алгоритмов, которые нельзя выразить в каком-то одном

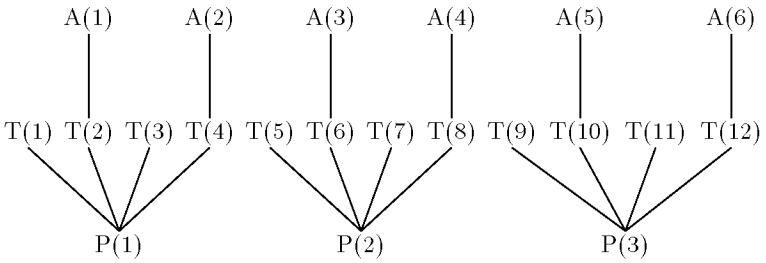
стиле параллельного программирования (то есть либо с помощью параллелизма данных, либо с помощью функционального параллелизма). Поэтому для для эффективного использования MPP-машин необходимы системы программирования, поддерживающие как параллелизм данных, так и функциональный параллелизм. Существуют различные подходы к решению этой задачи. Например, имеются идеи интегрировать HPF и Fortran M в рамках единой системы программирования или разработать привязку пакета MPI к языку HPF. Другой подход заключается в постепенном добавлении конструкций, основанных на параллелизме данных, таких как оператор `FORALL`, в функционально-параллельные языки, а средства функционального параллелизма, такие как узловые функции (nodal functions), — в языки параллелизма данных. Более решительные нововведения, направленные на интеграцию преимуществ функционально-параллельных языков и языков параллелизма данных, были введены в языке программирования mpC. Они включают развитые средства управления такими ресурсами MPP-компьютера, как процессоры и коммуникационные связи между ними (линки).

Хотя и существует довольно много параллельных языков, направленных на переносимое программирование MPP-компьютеров, только язык HPF стандартизован (как HPF 1.1 [31] в 1994 году). Уже имеется несколько коммерческих реализаций HPF для большинства MPP-компьютеров, декларирующих поддержку этого стандарта. Поэтому HPF обеспечивает более высокую степень переносимости для MPP-машин, чем любой другой язык параллельного программирования высокого уровня. HPF расширяет Фортран 90 главным образом средствами распределения данных, средствами параллельного выполнения на основе параллелизма данных, встроенными функциями и стандартной библиотекой.

Средства распределения данных включают директивы `DISTRIBUTE` и `REDISTRIBUTE`, `ALIGN` и `REALIGN`, `PROCESSORS`, `TEMPLATE`, `INHERIT`. Они позволяют программисту подсказать компилятору, как распределить элементы массивов между процессорами MPP-машины. Директива `PROCESSORS` объявляет массив абстрактных процессоров. Директива `TEMPLATE` объявляет абстрактный массив из пустых элементов (массив-образец), используемый как абстрактное пространство индексированных позиций при специфицировании с помощью директивы `ALIGN`, что определенные объекты данных должны быть отображены на абстрактные процессоры таким же образом, как некоторые другие объекты данных. Директива `DISTRIBUTE` специфицирует отображение объектов данных на абстрактные процессоры, в общем случае, путем отображения массива-образца на соответствующий массив абстрактных процессоров. Так, следующий фрагмент

```
!HPF$ PROCESSORS P(3)
!HPF$ TEMPLATE T(12)
      INTEGER, DIMENSION(6) :: A
!HPF$ ALIGN A(J) WITH T(J*2)
!HPF$ DISTRIBUTE T(BLOCK(4)) ONTO P
```

специфицирует, что элементы `A(2*I-1)`, `A(2*I)` массива A должны быть отображены на один и тот же абстрактный процессор `P(I)`, $\mathtt{I} = 1, \ldots, 4$, следующим образом



Директива `REDISTRIBUTE` аналогична директиве `DISTRIBUTE`, но рассматривается не как объявление, а как выполнимый оператор. Массив-образец, объявленный как `DYNAMIC`, может быть перераспределен в любое время, и все массивы данных, выровненные на момент перераспределения с этим массивом-образцом, также переотображаются на абстрактные процессоры, так чтобы сохранилось отношение выравнивания. Директива `REALIGN` аналогична `ALIGN`, но рассматривается не как объявление, а как выполнимый оператор. Любой массив данных, объявленный как `DYNAMIC`, может быть перевыровнен в любой момент времени. В отличие от перераспределения, перевыравнивание одного объекта данных не вызывает переотображения других объектов данных.

Средства параллельного выполнения на основе параллелизма данных включают оператор `FORALL` и конструкцию для одновременного присваивания больших групп элементов массивов, а также директиву `INDEPENDENT`, позволяющую программисту сообщать компилятору, например, об отсутствии объекта данных, определяемого одной итерацией цикла `DO` и используемого (для записи или чтения) другой итерацией (такая информация позволяет компилятору применять оптимизирующие преобразования в случаях, когда он сам не способен распознать их допустимость).

Процессоры конкретного MPP-компьютера могут быть упорядочены в виде многомерного массива процессоров, отражающего структуру коммуникационных связей между ними. Встроенные функции `NUMBER_OF_PROCESSORS` и `PROCESSORS_SHAPE` возвращают значения, связанные с этим конкретным MPP-компьютером и конфигурацией процессоров. `NUMBER_OF_PROCESSORS` в зависимости от параметров возвращает либо общее число процессоров, доступное программе, либо доступное число процессоров по специфицированному измерению процессорного массива. `PROCESSORS_SHAPE` возвращает форму процессорного массива. Значения, возвращаемые этими встроенными функциями, остаются неизменными в течение всего времени одного выполнения программы. Эти функции, а также подпрограммы-запросы, определяющие в период выполнения программы, каким образом в действительности тот

или иной массив данных отображен на исполняющий MPP-компьютер, позволяют писать переносимые программы, адаптирующиеся к конкретному MPP-компьютеру.

Таким образом, в общем случае, HPF поддерживает эффективно-переносимое модульное надежное программирования MPP-машин, по крайней мере, при решении регулярных задач.

Как уже отмечалось, основным недостатком HPF является то, что класс выразимых на нем параллельных алгоритмов весьма ограничен и состоит из регулярным образом параллелизуемых последовательных алгоритмов. Даже модульность в языке HPF ограничена, так как две процедуры, имеющие дело с распределенными данными, не могут выполняться параллельно. Такая ситуация не удовлетворяет программистов, использующих HPF для научного программирования. Поэтому работа над этим языком продолжается, а на его будущую улучшенную версию (хотя она и весьма далека от завершения) уже ссылаются как на HPF-2.

Эта работа прежде всего направлена на

- развитие возможностей пользователя по управлению отображением распределенных данных;
- предоставление пользователю возможности планировать параллельное выполнение различных участков программы;
- включение в язык средств для выражения элементов функционального параллелизма.

Необходимость в средствах для выражения функционального параллелизма мотивируется тем, что некоторые научные параллельные вычисления естественным образом описываются в терминах взаимодействующих задач среднего размера, поэтому реализация этих вычислений с помощью подходящих структур может оказаться и более простой, и более модульной, чем реализация через чистый параллелизм данных.

Необходимость предоставления пользователю средств для планирования параллельного выполнения программы мотивируется тем, что он может знать, что какие-то компоненты программы, отличные от итераций цикла `DO`, могут выполняться параллельно (например, рекурсивные вызовы в алгоритме «разделяй и властвуй» (divide-and-conquer) или последовательные стадии конвейера обработки изображения). Хотя этот вид параллелизма и может быть иногда выражен на языке HPF-1, получающийся при этом исходный код не является элегантным и с трудом может быть откомпилирован в эффективный целевой код. Кроме того, при определенных условиях различные итерации цикла `DO` могут выполняться параллельно. Пользователю может быть известно о выполнении этих условий, в то время как компилятору может быть очень непросто определить, выполняются ли они. Поэтому разумно предоставить пользователю возможность и в этом случае сообщать необходимую информацию компилятору.

Что касается управления отображением распределенных данных, то хотя HPF-1 и предоставляет для этого весьма разнообразные возможности, все они приводят к регулярным схемам. Этого недостаточно для программирования (по крайней мере, некоторых) сложных алгоритмов. В частности, следующие аспекты параллельных алгоритмов нуждаются в более сложном разбиении данных.

- *Несбалансированная нагрузка.* В случае, когда объем вычислений, связанный с обработкой различных элементов массива, варьирует, может оказаться, что лучшая сбалансированность вычислительной нагрузки обеспечивается неравномерным распределением элементов. Распределения HPF-1 можно сделать несбалансированными, если число элементов не делится нацело на число процессоров, однако такой способ решения проблемы считается нежелательным и неэлегантным.
- *Неструктурированный доступ к данным.* Когда структура данных имеет сложные взаимосвязи, может оказаться полезным разместить близкие по связям секции на один и тот же процессор. Такое размещение, как правило, приводит к разбиению массива на секции с несколько нерегулярными границами, а не на допускаемые HPF-1 секции, получающимся разрезанием массива ортогональными плоскостями. Проблема может стать особенно трудной, когда индексирование элементов не соответствует связям между ними.
- *Нелинейные (базирующиеся на указателях) структуры данных.* При отсутствии аналога индекса трудно компактно описать правило выравнивания соответствующих структур. По аналогичной причине недостаток априорной информации об организации структуры (то есть тот факт, что структура не является массивом) затрудняет описание общей схемы распределения.
- *Динамические структуры данных.* Если структура данных строится постепенно, то неуместным (если не сказать путающим) выглядит описание отображения этой структуры до ее создания. По-видимому, имеет смысл отложить отображение такой структуры данных до тех пор, пока ее создание не завершено.

Кроме перечисленных трех направлений, обсуждаются улучшения в области ввода-вывода, оптимизации коммуникаций и внешнего окружения языка.

В целом, основным направлением эволюции языка HPF является предоставление программисту больших возможностей как с точки зрения выразимости параллельных алгоритмов, так и с точки зрения управления эффективностью целевого кода.

4.3. Пакеты

Кроме универсальных средств программирования MPP-машин, таких как библиотеки передачи сообщений и языки параллельного программирования

высокого уровня, имеются специализированные средства, предназначенные для решения определенных задач на MPP-машинах. Среди этих средств в настоящее время наиболее продвинутым и известным является ScaLAPACK [32].

ScaLAPACK — это переносимый пакет для решения задач линейной алгебры на массивно-параллельных вычислительных системах с распределенной памятью. Он был разработан как основанная на парадигме передачи сообщений версия пакета LAPACK. Он базируется на двух дополнительных пакетах — BLACS (Basic Linear Algebra Communication Subprograms) и PBLAS (Parallel BLAS). BLACS обеспечивает коммуникации между процессами на двумерной решетке процессов для целей линейной алгебры и инкапсулирует конкретную коммуникационную платформу (такую как MPI или PVM). PBLAS представляет собой расширенное подмножество пакета BLAS для вычислительных систем с распределенной памятью и оперирует над матрицами, распределенными по блочно-циклической схеме. Он вызывает BLAS для вычислений и BLACS для коммуникаций. Заметим, что поддерживается лишь равномерное распределение данных, достаточное для эффективного решения регулярных задач линейной алгебры на MPP-машинах.

Высокоуровневые программы пакета ScaLAPACK позволяют, в частности, решать плотные системы линейных уравнений путем LU и QR-разложений, а также методом Холесского.

В целом, ScaLAPACK поддерживает разработку эффективно-переносимых программ, решающих задачи линейной алгебра на MPP-компьютерах.

5. Средства программирования локальных сетей компьютеров

Главной особенностью локальной сети компьютеров, отличающей ее от MPP-компьютера, является неоднородность. Типичный MPP-компьютер состоит из идентичных вычислительных процессорных узлов, связанных специальным оборудованием, обеспечивающим быстрые коммуникации между узлами. При этом производительность сетевого оборудования обычно сбалансирована с производительностью и числом вычислительных процессорных узлов, так чтобы обеспечивать приемлемое ускорение при увеличении числа участвующих процессоров для типичных масштабируемых и квази-масштабируемых MP-алгоритмов. В то же время локальные сети компьютеров являются неоднородными по своей природе и состоят из разнообразных рабочих станций, персональных компьютеров, серверов и, иногда, специализированных параллельных компьютеров, связанных между собой с помощью разнородного сетевого оборудования. В частности, это приводит к неоднородности с точки зрения производительности отдельных вычислительных узлов и скорости обмена сообщениями между ними. Как правило, производительность процессоров и сетевого оборудования в локальных сетях не сбалансирована

для высокопроизводительных распределенных вычислений. Обычно локальная сеть компьютеров развивается довольно спонтанно и на ее эволюцию влияет много факторов, среди которых высокопроизводительные распределенные вычисления редко имеют первостепенное значение.

Тем не менее, для переносимых высокопроизводительных вычислений на локальных сетях компьютеров используются, главным образом, те же инструменты, что и для переносимого программирования MPP и, частично, SMP-компьютеров, а именно PVM, MPI и HPF.

Для того чтобы понять, насколько адекватны эти инструменты для локальных сетей компьютеров, посмотрим, в какой степени они отвечают требованиям к инструментальным средствам программирования, сформулированным в разделе 1 этой статьи. Для этого переформулируем эти требования применительно к неоднородным вычислительным сетям (НВС).

Эффективное программирование НВС означает разработку программы, эффективно использующей вычислительную мощность конкретной сети компьютеров.

Переносимое программирование НВС означает создание программы, которая, будучи написанной и оттестированной для одной НВС, будет без переработки правильно работать на других НВС.

Модульное программирование НВС означает разработку отдельно компилируемой программной единицы, корректное использование которой другими пользователями при создании своих программ не требует знания ее внутренностей.

Эффективно-переносимое программирование НВС означает разработку переносимой программы, адаптирующейся к особенностям каждой конкретной целевой НВС (в частности, к числу и производительностям процессоров, скорости обмена данными между процессорами) с целью эффективного использования потенциала ее производительности.

Простое и надежное программирование НВС означает использование такой модели параллельного программирования, которая не делает разработку сложных программ слишком трудной и не провоцирует появления ошибок в программах.

И PVM, и MPI, и HPF поддерживают переносимое программирование НВС.

В отличие от PVM, MPI и HPF, кроме того, поддерживают модульное программирование НВС.

Для того чтобы разобраться, поддерживают ли MPI и HPF эффективное и эффективно-переносимое программирование НВС, рассмотрим несколько типичных параллельных алгоритмов, допускающих как эффективную, так и эффективно-переносимую реализацию для НВС, и проанализируем, как они могут быть выражены на MPI и HPF.

Сначала рассмотрим задачу моделирования эволюции системы звезд в галактике (или некоторого множества галактик) под воздействием гравитационного притяжения. Пусть моделируемая система состоит из некоторого чи-

сла больших групп тел. Известно, что поскольку сила взаимодействия между телами быстро падает с расстоянием, то воздействие большой группы тел может быть приближено воздействием одного эквивалентного тела, если эта группа тел находится достаточно далеко от точки, в которой вычисляется ее воздействие. Пусть это предположение выполняется в нашем случае, то есть пусть моделируемые группы тел находятся достаточно далеко друг от друга.

В этом случае мы можем естественным образом распараллелить задачу, и соответствующий MP-алгоритм будет использовать несколько процессов, каждый из которых занимается обновлением данных, характеризующих одну группу тел. Каждый процесс хранит атрибуты всех тел, образующих соответствующую группу, а также массы и центры тяжести остальных групп. Каждое тело представляется своими координатами, вектором скорости и массой. Имеется один процесс (назовем его хост-процессом), который хранит информацию о всех телах, составляющих галактику, и выводит (например, в графическом виде на экран) последовательные состояния галактики.

Схема работы этого MP-алгоритма выглядит следующим образом:

```
Инициализировать хост-процесс
Инициализировать галактику на хост-процессе
Инициализировать сбалансированное множество процессов
    для групп тел, составляющих галактику
    (из расчета один процесс на одну группу тел)
Разослать группы тел по процессам
Вычислить (параллельно) массы групп тел
Обменяться массами групп тел между процессами
while(1) {
    Вывести состояние галактики на хост-процессе
    Вычислить (параллельно) центры масс групп тел
    Обменяться центрами масс между процессами
    Параллельно обновить атрибуты групп тел
    Собрать группы тел на хост-процессе
}
```

Здесь «сбалансированное множество процессов» означает, что процессы отображаются на процессоры исполняющей НВС таким образом, чтобы обеспечить как баланс между скоростью различных процессов, так и баланс между скоростью процессов и скоростью передачи данных между процессами с тем, чтобы минимизировать общее время работы программы.

MPI не позволяет создать группу процессов, основываясь на таких ее свойствах, как относительные скорости процессов или скорости передачи данных между процессами создаваемой группы. Основной способ создания группы процессов заключается в явном перечислении тех процессов всего упорядоченного множества однородных MPI-процессов, составляющих выполняющуюся на НВС параллельную программу, которые включаются в создаваемую

группу. Единственным исключением является операция одновременного создания n новых подгрупп уже созданной группы, при которой каждый процесс группы сам указывает идентификатор подгруппы, в которую он хочет войти. Однако и этот способ создания групп процессов не решает проблемы создания группы процессов, основываясь на их относительных количественных характеристиках. Процесс в MPI идентифицируется лишь порядковым номером и не имеет никаких дополнительных свойств (атрибутов), отличающих его от других процессов. Поэтому программист, пишущий соответствующую MPI-программу, не может повлиять на сбалансированность процессов в создаваемой группе. С точки зрения MPI все группы процессов одинаково сбалансированы. Эта точка зрения объяснима, если вспомнить, что MPI разрабатывался прежде всего для переносимого программирования MPP-машин, состоящих из идентичных процессоров, связанных с помощью очень быстрого сетевого оборудования, и любые отображения совокупности процессов, составляющих выполняющуюся MPI-программу, приписывающие разные процессы разным процессорам MPP-компьютера, считаются эквивалентными и оптимальными. Поэтому, по существу, и не возникала проблема, каким образом запустить MPI-программу на конкретной MPP-машине с тем, чтобы обеспечить ее эффективное выполнение.

Ситуация резко меняется в случае HBC. Пусть, к примеру, моделируемая система тел включает пять групп g1, g2, g3, g4 и g5, состоящих из 100, 200, 300, 400 и 500 тел соответственно, а целевая HBC состоит из пяти однопроцессорных рабочих станций w1, w2, w3, w4 и w5, относительная производительность которых характеризуется числами 1, 2, 3, 4 и 5 соответственно. Упрощая ситуацию, предположим, что рабочие станции связаны сетевым оборудованием, достаточно производительным, чтобы в рассматриваемом случае пренебречь накладными расходами на передачу данных. Очевидно, что в этом случае отображение, приписывающее g1 к w5, g2 к w4, g3 к w3, g4 к w2 и g5 к w1, приведет к существенно более медленному выполнению программы, чем отображение, приписывающее g1 к w1, g2 к w2, g3 к w3, g4 к w4 и g5 к w5. Поэтому, для того чтобы запустить эту программу на HBC таким образом, чтобы обеспечить ее эффективное выполнение, пользователь должен хорошо знать как характеристики целевой HBC, так и внутренности запускаемой программы и решить нетривиальную оптимизационную задачу. Эта задача становится намного более сложной, если целевая HBC состоит из большого числа одно- и многопроцессорных компьютеров, связанных разнородным сетевым оборудованием, и / или если задача моделирования эволюции системы тел является лишь частью большей MPI-программы.

Таким образом, при программировании HBC невозможно эффективно использовать MPI-модули без знания их внутренностей, то есть MPI не может обеспечить одновременно модульное и эффективное программирование HBC.

Более того, если число групп тел и число тел в каждой группе определяется лишь в период выполнения программы, то у пользователя вообще отсутствует информация для эффективного запуска программы на HBC. Поэтому, в об-

щем случае, MPI не поддерживает эффективного программирования НВС, а следовательно, и их эффективно-переносимого программирования.

Как и MPI, HPF не позволяет выразить описанный выше алгоритм. Главной помехой является то, что единственной параллельной машиной, видимой при программировании на HPF, является однородный мультипроцессор, обеспечивающий очень быстрые коммуникации между его процессорами (то есть MPP-машина). Поэтому программист, пишущий соответствующую HPF-программу, не может повлиять на степень сбалансированности процессов целевой MP-программы. Кроме того, HPF не поддерживает ни нерегулярного и / или неравномерного распределения данных, ни среднеблочного параллелизма, чтобы выразить этот (более близкий к среднеблочному функциональному параллелизму, чем к чистому параллелизму данных) алгоритм. Таким образом, HPF также не поддерживает эффективное (и, следовательно, эффективно-переносимое) программирование НВС.

Все сказанное приводит к заключению, что MPI и HPF не подходят для эффективно-переносимого модульного программирования НВС. Подкрепим это заключение еще одним типичным параллельным алгоритмом решения задачи линейной алгебры, допускающим эффективно-переносимую реализацию для НВС. Задача заключается в умножении двух плотных квадратных $n \times n$ матриц X и Y. Алгоритм использует некоторое число процессов, каждый из которых выполняется на отдельном процессоре и вычисляет свое число строк результирующей матрицы Z. Число процессов зависит от характеристик целевой НВС, а число строк, вычисляемое процессом, зависит от мощности процессора, на котором выполняется процесс. Все эти числа определяются в период выполнения программы таким образом, чтобы обеспечить минимальное время выполнения программы. Описанный алгоритм не может быть выражен ни на MPI, ни на HPF (HPF не поддерживает неравномерного распределения строк матрицы между процессами, и ни MPI, ни HPF не могут обеспечить сбалансированную группу процессов, чтобы выполнить этот алгоритм).

Начинают появляться новые инструментальные средства программирования, ориентированные на эффективно-переносимое модульное программирование НВС. В частности, разработанная в Институте системного программирования РАН исследовательская система программирования mpC [33, 34] базируется на расширении языка Си, предоставляющем высокоуровневые средства, позволяющие программисту явно специфицировать свойства различных групп процессов, участвующих в выполнении различных частей единой распределенной программы. Соответствующая абстракция называется *сетевым объектом*. В языке mpC программист может специфицировать топологию сетевых объектов и определять (в частности, динамически) соответствующие сетевые объекты, а также распределять данные и вычисления между ними. Система программирования языка mpC использует эту информацию в период выполнения программы для того, чтобы отобразить сетевые объекты языка mpC на целевую НВС таким образом, чтобы обеспечить эффективное

выполнение программы на этой НВС.

Например, пользователь может написать на языке mpC следующий код

```
/*1*/     nettype HeteroNet(n, p[n]) {
/*2*/        coord I=n;
/*3*/        node { I>=0: p[I]; };
/*4*/     };
/*5*/     ...
/*6*/        {
/*7*/           repl int m, q[N];
/*8*/           ... /* Вычисление m и q[0],...,q[m-1] */
/*9*/           {
/*10*/             net HeteroNet(m,q) r;
/*11*/             ...
/*12*/          }
/*13*/       }
```

для того чтобы определить автоматический сетевой объект r, состоящий из m виртуальных процессоров, причем относительная производительность i-го виртуального процессора характеризуется значением q[i].

Здесь строки 1–4 объявляют сетевую топологию `HeteroNet`, параметризованную целым параметром n и векторным параметром p, состоящим из n целых. Строка 2 — это объявление системы координат, к которой привязываются виртуальные процессоры. Оно вводит координатную переменную I со значениями от 0 до $n-1$. Строка 3 — это объявление процессорных узлов. Оно привязывает виртуальные процессоры к введенной системе координат и объявляет их типы и производительности. В данном случае объявляется, что «для всех $I < n$ если $I \geqslant 0$, то виртуальные процессор, относительная производительность которого задается значением p[I], связывается с точкой с координатой [I]».

В строке 7 объявляются переменная m и массив q, размазанные по всему вычислительному пространству. По определению объект данных, *распределенный* по какой-нибудь области вычислительного пространства (в частности, по всему вычислительному пространству), состоит из набора компонент одного типа, так что каждый виртуальный процессор области содержит в точности одну компоненту. А *размазанный* объект данных определяется как распределенный объект, все компоненты которого равны между собой.

Концептуально, создание нового сетевого объекта инициируется виртуальным процессором, принадлежащим уже созданному сетевому объекту. Этот виртуальный процессор называется *родителем* создаваемого сетевого объекта. Родитель принадлежит созданному сетевому объекту. В нашем случае родителем сетевого объекта r является так называемый *хост-процессор* — единственный виртуальный процессор, определенный от начала и до конца выполнения программы.

Предположим теперь, что мы моделируем эволюцию m групп тел под воздействием силы притяжения и наша параллельная программа использует для обновления состояния каждой из групп отдельный виртуальный процессор. Предположим также, что значение q[i] равняется квадрату числа тел в i-ой группе. Тогда сетевой объект r, выполняющий основную часть вычислений и обменов данными, определяется в строке 10 таким образом, что он состоит из m виртуальных процессоров, а относительная производительность каждого виртуального процессора характеризуется объемом вычислений, необходимым для обновления состояния группы тел, которую он обсчитывает. Таким образом, чем более мощным объявлен виртуальный процессор, тем большую группу тел он обсчитывает.

Система программирования mpC использует эту информацию, а также информацию о производительностях физических процессоров и топологии конкретной НВС, на которой выполняется программа, чтобы отобразить как можно более подходящим образом эти виртуальные процессоры во множество процессов, составляющих целевую параллельную программу и представляющих вычислительное пространство. Так как это делается во время выполнения программы, то пользователю не требуется перекомпилировать эту mpC программу при переносе на другие НВС.

Дополнительно, для разработки программ, адаптирующихся к особенностям конкретной исполняющей НВС, можно использовать обращения к следующим библиотечным функциям:

```
int MPC_Processors(int* num_of_procs,
                   double** relative_performance);
int MPC_Links(MPC_Links ** links);
```

Первая функция возвращает число доступных физических процессоров исполняющей НВС и их относительные производительности. Вторая функция возвращает информацию о сетевой структуре исполняющей НВС и ее коммуникационных характеристиках. Предположим, что нам нужно перемножить две плотные $k \times k$-матрицы X и Y и наша параллельная программа использует некоторое число виртуальных процессоров, каждый из которых вычисляет свое число строк результирующей матрицы Z. Наконец, пусть мы используем в строке 8 следующий код

```
repl double *powers;\
repl MPC_Link *links;\
external repl k;\
MPC_Processors(&m,&powers);\
MPC_Links(&links);\
Partition(&m, powers, links, q, k);
```

для вычисления m, q[0], ... ,q[m-1]. Основываясь на числе и производительностях физических процессоров и сетевых характеристиках исполняющей НВС,

определенная пользователем функция `Partition` вычисляет, сколько физических процессоров будет участвовать в умножении матриц и сколько строк результирующей матрицы будет вычисляться каждым из участвующих процессоров (заметим, что число физических процессоров, участвующих в умножении матриц, может оказаться меньше общего числа доступных физических процессоров, так как накладные расходы на коммуникации могут превысить выигрыш от распараллеливания). Таким образом, после вызова этой функции переменная m будет содержать число участвующих физических процессоров, а q[i] будет содержать число строк результирующей матрицы, вычисляемое i-м из участвующих процессоров. Сетевой объект r, выполняющий все остальные вычисления и коммуникации, определяется таким образом, что чем мощнее виртуальный процессор, тем больше строк он вычисляет. Система программирования mpC будет пытаться обеспечить оптимальное отображение виртуальных процессоров, составляющих r, во множество процессов, представляющих все вычислительное пространство. Таким образом, не более одного процесса из процессов, выполняющихся на каждом из физических процессоров, будет вовлечено в умножение матриц и чем мощнее физический процессор, тем большее число строк будет вычислять выполняющийся на нем процесс.

Таким образом, в отличие от MPI и HPF, поддерживающих эффективно-переносимое модульное программирование MPP-машин (включая однородные кластеры рабочих станций на основе очень быстрого сетевого оборудования), mpC, кроме того, поддерживает эффективно-переносимое модульное программирование неоднородных вычислительных сетей, включающих MPP-машины в качестве частного случая.

6. Заключение

В статье мы попытались проанализировать основные инструментальные средства разработки параллельных программ, главным образом, с точки зрения эффективности и переносимости последних. За ее рамками остались средства отладки, верификации, визуализации параллельных программ, средства программирования, направленные на разработку безопасных и отказоустойчивых параллельных программ, а также многие другие средства параллельного программирования, напрямую не адресующиеся к проблемам эффективности и переносимости.

Если обратиться к принятой в настоящее время классификации средств разработки программ для параллельных архитектур, включающей библиотечные расширения последовательных языков программирования, параллелизующие компиляторы, параллельные языки программирования, а также координационные языки в сочетании с последовательными языками программирования, то мы практически не затрагивали последнего из перечисленных направлений. Дело в том, что так называемое координационное программирование не стало

пока достаточно четко очерченной областью параллельного программирования. Многие относят к координационным языкам и языки управления заданиями (такие как язык shell операционной системы Unix), и языки формирования конфигурационных файлов (например, управляющие загрузкой параллельной MPI-программы в системах MPICH и LAM), графические языки типа HeNCE или Meander и многое другое. Что же касается одного из самых известных представителей этого направления — языка Linda, то он не ориентирован на разработку эффективных параллельных программ.

Литература

[1] ANSI, Programming Language C. — X3.159-1989. –American National Standard Institute, 1989.

[2] R. M. Stallman. Using and Porting GNU CC. — Cambridge, MA: Free Software Foundation, May 1992.

[3] ISO, Fortran 90. — May 1991.

[4] M. Metcalf, and J. Reid. Fortran 90/95 Explained. — Oxford and New York: Oxford University Press, 1996.

[5] С. С. Гайсарян, А. Л. Ластовецкий, И. Н. Ледовских, Д. А. Халецкий. Расширение ANSI C для векторных и суперскалярных компьютеров // Программирование. — 1995. — Т. 21, № 1. — С. 26–36.

[6] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers // Proceedings of the First International Workshop on Parallel Processing. Bagalore, India, December 1994. — P. 322–330.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jlang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing. — Cambridge, MA: MIT Press, 1994.

[8] The Nexus Multithreaded Runtime System. — `http://www.mcs.anl.gov/nexus/index.html`.

[9] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable Programming with the Parmacs message-passing library // Parallel Computing. Special issue on message-passing interface (to appear).

[10] R. Butler and E. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System // Journal of Parallel Computing. — 1994. — V. 20. — P. 547–568.

[11] W. D. Gropp, and B. Smith. Chameleon Parallel Programming Tools Users Manual. — Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[12] CHIMP Version 1.0 Interface. — Edinburgh Parallel Computing Centre, University of Edinburgh, May 1992.

[13] G. Geist, M. Heath, B. Peyton, and P. Worley. A User's Guide To PICL: A Portable Instrumented Communication Library. — Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[14] A. Skjellum, and A. Leung. Zipcode: A Portable Multicomputer Communication Library // Proceedings of the 5-th Distributed Memory Concurrent Computing Conference. — IEEE Computer Society Press, 1990. — P. 767–776.

[15] Message Passing Interface Forum, MPI: A Message-passing Interface Standard, version 1.1, June 1995.

[16] LAM/MPI Parallel Computing. — `http://www.osc.edu/lam.html`.

[17] MPICH — A Portable Implementation of MPI. — `http://www.mcs.anl.gov/mpi/mpich/`.

[18] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing // Proceedings of Supercomputing 91. Albuquerque, New Mexico, November 1991. — P. 435–444.

[19] J. C. Browne. Software engineering of parallel programs in a computationally oriented display environment // Languages and Compilers for Parallel Computing. — Cambridge, MA: MIT Press, 1990. — P. 75–94.

[20] Г. Виртц. Язык MEANDER и его программное окружение // Программирование. — 1995. — Т. 21, № 1. — С. 15–25.

[21] INMOS Ltd., Occam Programming Manual. — Prentice-Hall International, 1984.

[22] I. Foster, and K. M. Chandy. Fortran M: A language for modular parallel programming // Journal of Parallel and Distributed Computing. — 1995. — V. 26, No. 1. — P. 24–35.

[23] K. M. Chandy, and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01, California Institute of Technology, Pasadena, CA, 1992.

[24] Ч. Хоар. Взаимодействующие последовательные процессы. — М.: Мир, 1989.

[25] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y.Wu. Fortran D Language Specification. Houston, TX: Center for Research on Parallel Computation, Rice University, October 1993.

[26] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran // Scientific Programming. — 1992. — V. 1, No. 1. — P. 31–50.

[27] The CM Fortran Programming Language. CM-5 Technical Summary. Cambridge, MA: Thinking Machines Corporation, November 1992. — P. 61–67.

[28] The C* Programming Language. CM-5 Technical Summary. — Cambridge, MA: Thinking Machines Corporation, November 1992. — P. 69–75.

[29] P. J. Hatcher, and M. J. Quinn. Data-Parallel Programming on MIMD Computers. — Cambridge, MA: The MIT Press, 1991.

[30] M. Philippsen, and W. Tichy. Modula-2* and its compilation // First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991.

[31] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.1. — Houston, TX: Rice University, November 10, 1994.

[32] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers — Design Issues and Performance. UT, CS-95-283, March 1995.

[33] D. Arapov, V. Ivannikov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis. Modular Parallel Programming in mpC for Distributed Memory Machines // Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms / Architectures Synthesis (pAs'97). — Aizu-Wakamatsu, Japan: IEEE CS Press, March 1997. — P. 248–255.

[34] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis. A Programming Environment for Heterogeneous Distributed Memory Machines // Proceedings of the 7-th Heterogeneous Computing Workshop (HCW'97) of the 11th International Parallel Processing Symposium (IPPS'97). — Geneva, Switzerland: IEEE CS Press, April 1997. — P. 32–45.

Статья поступила в редакцию в мае 1998 г.