Язык и система программирования для высокопроизводительных параллельных вычислений на неоднородных сетях

А.Л. Ластовецкий, А.Я. Калинов, И.Н. Ледовских,
Д.М. Арапов, М.А. Посыпкин

Аннотация

В статье описывается язык *mpC*, разработанный специально для программирования высокопроизводительных вычислений на неоднородных сетях компьютеров. Программа на *mpC* явно определяет абстрактную вычислительную сеть и распределяет данные, вычисления и коммуникации по этой сети. Система программирования *mpC* в период выполнения программы использует эту информацию и информацию о реальной сети, выполняющей программу, для такого размещения процессов программы на физической сети, которое обеспечивало бы эффективное выполнение программы. Описывается опыт использования *mpC* для решения задач на локальных сетях неоднородных рабочих станций.

1. Введение

Еще 10 лет назад к высокопроизводительным параллельным вычислительным системам относили лишь так называемые суперкомпьютеры - мультипроцессоры с общей памятью (SMP) и мультипроцессоры с распределенной памятью (MPP). Параллельные вычисления на обычных сетях рабочих станций и персональных компьютеров не имели смысла, поскольку не могли ускорить решение большинства задач из-за низкой производительности типового сетевого оборудования. Однако с начала 1990х годов рост производительности типового сетевого оборудования уже устойчиво обгоняет рост производительности процессоров [1], (pp.6-7). Современные сетевые технологии, такие как Fast Ethernet, ATM, Myrinet и другие, уже обеспечивают обмен данными между компьютерами на уровне сотен мегабит и даже нескольких гигабит в секунду. Это привело к ситуации, когда не только специализированные параллельные компьютеры, но и обычные локальные и даже глобальные сети можно использовать в качестве параллельных вычислительных систем для высокопроизводительных вычислений. Объявленная президентом США стратегическая инициатива, направленная на тысячекратное ускорение обмена данными в сети Internet и поддержанная ведущими компаниями (телекоммуникационными и компьютерными), ясно указывает на наметившийся сдвиг высокопроизводительных вычислений в сторону сетевых вычислений.

Таким образом, в настоящее время сети компьютеров являются самой доступной и распространенной параллельной архитектурой и, зачастую, необходимое ускорение решения тех или иных задач может быть достигнуто не путем приобретения нового более мощного компьютера, а за счет использования вычислительного потенциала уже имеющихся компьютеров, связанных с помощью современного сетевого оборудования.

Использование сетей для параллельных высокопроизводительных вычислений сдерживается лишь отсутствием подходящего программного обеспечения. Дело в том, что, в отличие от суперкомпьютеров, сети по своей природе неоднородны и состоят из разнообразных компьютеров обычно различной производительности, связанных в сеть иногда неоднородным сетевым оборудованием, предоставляющим разную скорость обмена данными между разными процессорными узлами. Как правило, параллельная программа, перенесенная с (однородного) суперкомпьютера, выполняется на неоднородной сети с той же скоростью, с какой она выполнялась бы на однородной сети, состоящей из процессоров эквивалентных по производительности слабейшему процессору исходной неоднородной сети, число которых равно числу процессоров исходной неоднородной сети. Объясняется это тем, что такие параллельные программы распределяют данные, вычисления и коммуникации без учета различий в производительности процессоров и коммуникационных линков. Это резко снижает эффективность использования производительного потенциала сетей и приводит к крайне незначительному их использованию для высокопроизводительных параллельных вычислений.

В настоящее время основными средствами параллельного программирования для сетей являются MPI [2], PVM [3] и HPF [4].

PVM (Parallel Virtual Machine) и MPI (Message Passing Interface) - это библиотеки передачи сообщений, являющиеся, по сути, средствами уровня ассемблера для параллельного программирования. Из-за низкого уровня их параллельных примитивов разработка на *PVM/MPI* не учебных, а практически полезных и сложных программ представляет собой крайне трудную задачу и требует очень высокой квалификации разработчика. Кроме того, эти библиотеки не проектировались для разработки адаптивных параллельных программ, то есть программ, распределяющих вычисления и коммуникации в зависимости от входных данных и особенностей конкретной исполняющей неоднородной сети. Конечно, благодаря их низкому уровню, у пользователя есть возможность написать специальную систему времени выполнения, обеспечивающую адаптируемость его программы, однако такая система, как правило, настолько сложна, что необходимость ее разработки способна отпугнуть большинство нормальных пользователей.

HPF (High Performance Fortran) представляет собой параллельный язык высокого уровня, первоначально спроектированный для программирования (однородных) суперкомпьютеров. Поэтому единственной параллельной архитектурой, видимой при программировании на HPF, является однородный мультипроцессор с очень быстрыми коммуникационными линками между его процессорами. HPF не поддерживает нерегулярное распределение данных, неоднородное распределение данных, а также среднеблочный параллелизм. Типичный компилятор языка HPF транслирует программу на HPF в программу на PVM или MPI, и программист не может влиять на сбалансированность процессов целевой программы передачи сообщений. Кроме того, HPF очень труден для компиляции. Даже лучшие компиляторы HPF генерируют код, в среднем в 2-3 раза медленнее выполняемый на однородных кластерах рабочих станций, чем вручную написанный на MPI (по материалам конференции пользователей HPF [5], состоявшейся в июне 1998 года в Порто, Португалия). Поэтому HPF также не вполне подходит для программирования высокопроизводительных вычислений на сетях.

Таким образом, нужны новые, специальные инструментальные средства, которые бы позволили эффективно использовать неоднородные сети компьютеров в качестве параллельных вычислительных систем с распределенной памятью.

В статье описывается первый параллельный язык, специально спроектированный для программирования неоднородных сетей, а также поддерживающая его система программирования. Этот язык, названный *mpC*, представляет собой расширение ANSI C. Подобно HPF, он включает векторное подмножество [6]. Известные языки параллельного программирования ориентированы на программирование регулярных параллельных архитектур, параметризуемых с помощью небольшого числа параметров. Регулярность целевых архитектур позволяет неявно встраивать их в эти языки. Этот традиционный подход неприменим к проектированию параллельного языка, ориентированного на неоднородные сети, поскольку эта архитектура не имеет регулярной структуры. Основная идея, лежащая в основе языка *mpC*, заключается в предоставлении пользователю языковых конструкций, позволяющих ему определять абстрактную неоднородную параллельную машину, наиболее подходящую для выполнения его алгоритма. Эта информация вместе с информацией о физической параллельной системе используется системой программирования *mpC* для обеспечения эффективного выполнения соответствующей программы на этой физической параллельной системе.

Статья построена следующим образом. Раздел 2 дает введение в *mpC*. Раздел 3 кратко описывает принципы реализации. В разделе 4 обсуждаются проблемы оценки характеристик параллельной вычислительной системы, на которой выполняется программа. Раздел 5 описывает опыт использования *mpC* для разработки реальных приложений. В разделе 6 описываются похожие работы. Раздел 7 содержит заключение.

2. Введение в *mpC*

В языке *mpC* вводится понятие вычислительного пространства, которое определяется как некоторое доступное для управления множество виртуальных процессоров различной производительности, связанных линками с различной скоростью передачи.

Основным понятием *mpC* является понятие сетевого объекта или просто сети. Сеть состоит из виртуальных процессоров различной производительности, связанных между собой линками с различной скоростью передачи. Сеть является областью вычислительного пространства, которая может быть использована для вычисления выражений и выполнения операторов.

Размещение и освобождение сетевых объектов в вычислительном пространстве выполняется в аналогичной манере с размещением и освобождением объектов данных в памяти в языке C. Концептуально, создание новой сети

инициируется процессором уже существующей сети. Этот процессор называется родителем создаваемой сети. Родитель всегда принадлежит создаваемой сети. Единственным процессором, определенным от начала и до конца выполнения программы, является предопределенный виртуальный хост-процессор.

Любой сетевой объект, объявленный в программе, имеет тип. Тип специфицирует число, типы и производительности процессоров, линки между процессорами и их скорости, а также родителя сети. Например, объявление

```
/* Строка 1 */   nettype Rectangle {
/* Строка 2 */     coord I=4;
/* Строка 3 */     node { I>=0 : I+1; };
/* Строка 4 */     link {
/* Строка 5 */       I>0:  [I]<->[I-1];
/* Строка 6 */       I==0:  [I]<->[3];
/* Строка 7 */     };
/* Строка 8 */     parent [0];
/* Строка 9 */   };
```

вводит сетевой тип с именем Rectangle, соответствующий сетям, состоящим из четырех виртуальных процессоров различной производительности, связанных в прямоугольник ненаправленными линками с обычной скоростью.

Здесь строка 1 содержит заголовок объявления сетевого типа. Она вводит имя сетевого типа.

Строка 2 содержит объявление системы координат, к которой привязываются процессоры. Она вводит целую координатную переменную I со значениями от 0 до 3.

Строка 3 содержит объявление процессорных узлов. Оно привязывает процессоры к системе координат и объявляет их типы и производительности. Строка 3 соответствует предикату ***для всех*** *I<4* ***если*** *I>=0,* ***то*** *виртуальный процессор с относительной производительностью, определяемой величиной I+1, привязывается к точке с координатой [I]*. Выражение I+1 называется спецификатором производительности. Большее число задает большую производительность. В примере 0-й виртуальный процессор в два раза медленней, чем 1-й, в три раза медленнее, чем 2-й и в четыре раза медленнее, чем 3-й виртуальный процессор. Для любой сети такого типа эта информация позволяет компилятору приписать каждому виртуальному процессору его вес, нормализованный относительно родительского узла сети.

Строки 4-7 содержат объявление линков. Оно специфицирует линки между процессорами. Строка 5 соответствует предикату ***для всех*** *I<4* ***если*** *I>0* ***то*** *имеется ненаправленный линк с обычной скоростью, связывающий процессоры с координатами [I] и [I-1]*, а строка 6 соответствует предикату ***для всех*** *I<4* ***если*** *I==0* ***то*** *имеется ненаправленный линк с обычной скоростью, связывающий процессоры с координатами [I] и [3]*. Отметим, что если линк между двумя процессорами не специфицирован явно, то это означает наличие между ними линка с минимальной для данной сети скоростью.

Строка 8 содержит объявление родителя. Она специфицирует, что родительский процессор имеет координату [0] в создаваемой сети.

Имея объявление сетевого типа, можно объявить идентификатор сетевого объекта этого типа. Например, объявление

```
net Rectangle r1;
```

вводит идентификатор r1 сетевого объекта типа Rectangle.

Понятие распределенного объекта вводится в стиле языков C* [7] и Dataparallel C [8]. По определению, объект данных, распределенный по некоторой области вычислительного пространства, составляется из обычных (нераспределенных) объектов одного типа (называемых компонентами распределенного объекта данных), размещенных в процессорных узлах этой области таким образом, что каждый процессорный узел содержит одну и только одну компоненту. Например, объявления

```
net Rectangle r2;
int [*]Derror, [r2]Da[10];
float [host]f, [r2:I<2]Df;
repl [*]di;
```

вводят:

- целую переменную *Derror*, распределенную по всему вычислительному пространству;
- массив *Da* из десяти целых, распределенный по сети *r2*;
- нераспределенную вещественную переменную *f*, принадлежащую виртуальному хост-процессору;
- вещественную переменную *Df*, распределенную по подсети сети *r2*;
- целую переменную *di*, размазанную по всему вычислительному пространству.

По определению, распределенный объект является размазанным, если все его компоненты равны между собой.

Понятие распределенного значения вводится аналогично понятию распределенного объекта данных.

Кроме простого сетевого типа, можно объявлять параметризованное семейство сетевых типов, называемое топологией или параметризованным сетевым типом. Например, объявление

```
/* Строка 1 */     nettype Ring(n, p[n])
{
/* Строка 2 */     coord I=n;
/* Строка 3 */     node {
/* Строка 4 */     I>=0: p[I];
/* Строка 5 */     };
/* Строка 6 */     link {
/* Строка 7 */     I>0:  [I]<->[I-1];
```

```
/* Строка 8 */      I==0: [I]<->[n-1];
/* Строка 9 */      };
/* Строка 10 */     parent [0];
/* Строка 11 */     ;
```

вводит топологию с именем `Ring`, соответствующую сетям, состоящим из *n* виртуальных процессоров, связанных в кольцо с помощью ненаправленных линков с нормальной скоростью.

Заголовок объявления (строка 1) вводит параметры топологии `Ring`, а именно, целый параметр `n` и векторный параметр `p`, состоящий из `n` целых. Соответственно, координатная переменная `I` пробегает значения от 0 до `n`-1, строка 4 соответствует предикату ***для всех*** `I<n` ***если*** `I>=0`, ***то*** *виртуальный процессор с относительная производительностью, специфицируемой значением* `p[I]`, *привязывается к точке с координатами* `[I]` и т.п.

Имея объявление топологии, можно объявить идентификатор сетевого объекта подходящего типа. Например, фрагмент

```
     repl [*]m,[*]n[100];
     /* Вычисление m, n[0], ..., n[m-1] */
     net Ring(m,n) r;
```

вводит идентификатор `rr` сетевого объекта, чей тип определяется полностью только во время выполнения программы. Сеть `rr` состоит из *m* виртуальных процессоров. Относительная производительность `i`-го виртуального процессора определяется значением `n[i]`.

Сетевой объект характеризуется классом вычислительного пространства, выделенного для него и определяющего время его жизни. Область вычислительного пространства может выделяться статически или автоматически. Вычислительное пространство под статическую сеть отводится один раз. Будучи созданной, сеть существует до конца выполнения программы. Новый экземпляр сети, описанной как автоматическая, создается при каждом входе в блок, в котором сеть описана, и уничтожается при каждом выходе из блока.

Рассмотрим простую *mpC* программу умножения двух плотных квадратных матриц *X* и `Y`, использующую несколько виртуальных процессоров, каждый из которых вычисляет часть строк результирующей матрицы `Z`.

```
/* 1 */  #include <stdio.h>
/* 2 */  #include <stdlib.h>
/* 3 */  #include <mpc.h>
/* 4 */  #define N 1000
/* 5 */  void [host]Input(), [host]Output();
/* 6 */  nettype Star(m, n[m]) {
/* 7 */   coord I=m;
/* 8 */   node { I>=0: n[I]; };
/* 9 */   link { I>0:  [0]<->[I]; };
/* 10*/  };
/* 11*/  void [*]main()
/* 12*/  {
/* 13*/   double [host]x[N][N], [host]y[N][N], [host]z[N][N];
/* 14*/   repl int nprocs;
/* 15*/   repl double *powers;
/* 16*/   Input(x, y);
/* 17*/   MPC_Processors(&nprocs, &powers);
/* 18*/   {
/* 19*/    repl int ns[nprocs];
/* 20*/    MPC_Partition_1b(nprocs, powers, ns, N);
/* 21*/    {
/* 22*/     net Star(nprocs, ns) w;
/* 23*/     int [w]myn;
/* 24*/     myn=([w]ns)[I coordof myn];
/* 25*/     {
/* 26*/      repl int [w]i, [w]j;
/* 27*/      double [w]dx[myn][N],[w]dy[N][N], [w]dz[myn][N];
/* 28*/      dy[]=y[];
/* 29*/      dx[]=::x[];
/* 30*/      for(i=0; i<myn; i++)
/* 31*/       for(j=0; j<N; j++)
/* 32*/ dz[i][j]=[+](dx[i][]*(double[*][N:N])(dy[0]+j)[]);
/* 33*/      z[]::=dz[];
/* 34*/     }
/* 35*/    }
/* 36*/    Output(z);
/* 37*/   }
/* 38*/  }
```

Программа включает в себя 5 функций: `main`, приведенную выше, `Input` и `Output`, определенные в других исходных файлах, и библиотечные функции `MPC_Processors` и `MPC_Partition_1b`. Функции `Input` и `Output` объявляются в строке 5, а функции `MPC_Processors` и `MPC_Partition_1b` объявляются в файле `mpc.h`.

В общем случае, *mpC* допускает 3 класса функций. В этом примере используются функции всех трех видов: `main` относятся к классу базовых функций, `Input` и `Output` к классу сетевых функций и `MPC_Processors` и `MPC_Partition_1b` к классу узловых функций.

Вызов базовой функции всегда является тотальным выражением(то есть оно вычисляется на всем вычислительном пространстве; никакие другие вычисления не могут выполняться параллельно с вычислением тотального выражения). Ее аргументы, если таковые имеются, либо принадлежат хост-процессору, либо распределены по всему вычислительному пространству, а возвращаемое значение (если таковое имеется) распределено по всему вычислительному пространству. В отличие от других видов функций, базовая функция может содержать определения сетей. В строке 11 конструкция `[*]`, помещенная перед

идентификатором функции main, специфицирует, что объявляется идентификатор базовой функции.

Узловая функция может быть полностью выполнена на любом одном процессоре вычислительного пространства. В ней могут создаваться только локальные объекты данных виртуального процессора, на котором она вызывается, кроме которых могут использоваться и компоненты внешних объектов данных, принадлежащие этому процессору. Объявление идентификатора узловой функции не требует никаких дополнительных спецификаторов. С точки зрения *mpC* все обычные функции языка Си являются узловыми.

В общем случае, сетевая функция вызывается и выполняется на некоторой области вычислительного пространства, и ее аргументы и возвращаемое значение, если таковые имеются, также распределены по этой же области. Две сетевые функции могут выполняться параллельно, если области, на которых они вызваны, не пересекаются. Функции Input и Output представляют собой простейшую форму сетевой функции, которая может быть вызвана только на статически определенной области вычислительного пространства. Они объявлены в строке 5 как сетевые функции, которые могут быть вызваны и выполнены только на виртуальном хост-процессоре (это задается конструкцией [host], помещенной непосредственно перед идентификаторами функций). Поэтому вызовы функций в строках 16 и 36 выполняются на виртуальном хост-процессоре.

Строки 11-38 содержат определение функции main. Строка 13 содержит определение массивов *x*, *y* и *z*, принадлежащих виртуальному хост-процессору.

Строка 14 определяет целую переменную nprocs, размазанную по всему вычислительному пространству. Ее распределение определяется правилом умолчания без помощи конструкции [*]. В общем случае, распределение по умолчанию задается распределением наименьшего блока, объемлющего соответствующее объявление и имеющего явно определенное распределение. Определение в стоке 14 содержится в теле функции main, для которой явно задано распределение по всему вычислительному пространству.

Строка 15 определяет распределенную по всему вычислительному пространству переменную *powers* типа указатель на вещественное число. Объявление определяет, что все распределенные объекты данных, на которые указывает *powers*, являются размазанными.

В строке 17 библиотечная узловая функция MPC_Processors, возвращая число физических процессоров и их производительности вызывается на всем вычислительном пространстве (механизм оценки производительности процессоров описан в разделе 4). Таким образом, сразу после этого вызова размазанная переменная *nprocs* содержит число физических процессоров, а размазанный массив *powers* содержит их производительности.

Строка 19 определяет динамический целый массив *ns*, размазанный по всему вычислительному пространству. Все компоненты массива состоят из одинакового числа элементов *nprocs*.

В строке 20 библиотечная узловая функция MPC_Partition_1b вызывается на всем вычислительном пространстве. Основываясь на производительностях физических процессоров, эта функция вычисляет, сколько строк результирующей матрицы будет вычисляться каждым из физических процессоров. Таким образом, сразу после этого вызова *ns*[*i*] содержит число строк, вычисляемое i-ым физическим процессором. MPC_Partition_1b разбивает данное целое (*N* в приведенном выше вызове) на части в соответствии с заданной пропорцией.

В строке 22 определяется автоматическая сеть *w*, состоящая из *nprocs* виртуальных процессоров. Относительная производительность *i*-го виртуального процессора определяется значением *ns*[*i*]. Таким образом, тип этой сети определяется полностью только в период выполнения. Эта сеть, выполняющая остальные вычисления и обмены данными, определяется таким образом, что чем мощнее виртуальный процессор, тем большее число строк он вычисляет. Система программирования *mpC* будет обеспечивать оптимальное отображение виртуальных процессоров, образующих сеть *w*, во множество процессов, представляющих вычислительное пространство. Таким образом, в точности один процесс из процессов, выполняющихся на каждом из физических процессоров, будет вовлечен в умножение матриц, и чем мощнее будет соответствующий физический процессор, тем большее число строк он будет вычислять.

Строка 23 определяет переменную *myn*, распределенную по w.

Результатом бинарной операции coordof в строке 24 будет целое значение, распределенное по *w*, каждая компонента которого равна значению координаты *I* виртуального процессора, которому эта компонента принадлежит. Правый операнд операции не вычисляется, а используется для спецификации

области вычислительного пространства. Заметим, что координатная переменная `I` интерпретируется как целая переменная, распределенная по области вычислительного пространства. Таким образом, после выполнения оператора в строке 24 каждая компонента `myn` содержит число строк результирующей матрицы, вычисляемое виртуальным процессором, которому она принадлежит.

Строка 26 определяет целые переменные `i` и `j` размазанные по сети `w`.

Строка 27 определяет три массива, распределенных по сети `w`. Тип `dy` определен статически как массив из `N` массивов из `N` вещественных чисел. Тип `dx` и `dz` динамически определяется как массив из `myn` массивов из `N` вещественных чисел. Заметим, что размерность `myn` массивов `dx` и `dz` не одинакова для различных компонент этих массивов.

Строка 28 содержит необычный унарный постфиксный оператор `[]`. Дело в том, что, строго говоря, *mpC* является расширением векторного расширения ANSI C называемого C[] [14], в котором введено понятие вектора как упорядоченной последовательности значений некоторого одного типа. В отличие от массива, вектор является не объектом данных, а новым видом значения. В частности, значением массива является вектор. Оператор `[]` был введен для доступа к массиву как целому. Он имеет операнд типа "массив" и блокирует преобразование операнда к указателю. Таким образом, `y[]` обозначает массив `y` как один объект, а `dy[]` обозначает распределенный массив `dy` как один объект.

Оператор в строке 28 рассылает матрицу `Y` от родителя сети `w` всем виртуальным процессорам этой сети. В результате каждая компонента распределенного массива, на который указывает `dy`, будет содержать эту матрицу. В общем случае, если левый операнд оператора `=` распределен по некоторой области вычислительного пространства `R`, значение правого операнда принадлежит некоторой области вычислительного пространства, объемлющей `R` и присвоение может быть произведено без преобразования типов, то выполнение оператора заключается в посылке значения правого операнда к каждому виртуальному процессору области `R`, где оно присваивается соответствующей компоненте левого операнда.

Оператор в строке 29 рассылает матрицу `X` с виртуального хост-процессора по всем виртуальным процессорам сети `w`. В результате каждая компонента `dx` содержит соответствующую порцию матрицы `X`.

В общем случае первым операндом 4-хместной операции `=::` должен быть массив, распределенный по некоторой области `R`, состоящей из `NP` виртуальных процессоров. Остальные операнды (второй и третий могут отсутствовать) - нераспределенные и принадлежат одному процессору. Второй операнд - необязательный, но если имеется, то должен либо указывать на начальный элемент *NP*-элементного целого массива, либо на *NP*-элементный целый массив. Третий операнд тоже необязательный и должен либо указывать на указатель на начальный элемент *NP*-элементного целого массива, значение *i*-го элемента которого не должно быть больше числа элементов *i*-й компоненты первого операнда, либо указывать на такой массив. Четвертый операнд должен быть массивом, значения элементов которого могут быть без преобразования типа присвоены любому элементу любой компоненты первого операнда.

Выполнение `e1=e2:e3:e4` заключается в вырезании `NP` подмассивов (возможно, перекрывающихся) из массива `e4` и посылке значения *i*-го подмассива *i*-му виртуальному процессору области `R`, где оно присваивается соответствующей компоненте распределенного массива `e1`. Смещение *i*-го подмассива относительно начального элемента массива `e4` задается значением *i*-го элемента массива `e2`, а его длина - значением *i*-го элемента массива `e3`.

Если `e3` указывает на пустой указатель (имеющий значение `NULL`), то операция выполняется, как если бы `*e3` указывало бы на начальный элемент *N*-элементного целого массива, значение *i*-го элемента которого равняется длине *i*-ой компоненты распределенного массива `e1`. Более того, в этом случае такой массив действительно создается в результате выполнения операции, а указатель на его начальный элемент присваивается `*e3`.

Если `e2` указывает на пустой указатель (имеющий значение `NULL`), то операция выполняется, как если бы `*e2` указывало бы на начальный элемент *N*-элементного целого массива, значение нулевого элемента которого равняется 0, а значение *i*-го элемента равняется сумме значения его (*i*-1)-го элемента и значения i-го элемента массива `e3`. Более того, в этом случае такой массив действительно создается в результате выполнения операции, а указатель на его начальный элемент присваивается `*e2`.

Второй операнд операции может быть опущен. В этом случае операция выполняется, как если бы `e2` указывало на пустой указатель. Единственное различие заключается в том, что созданный в ходе выполнения *NP*-элементный массив освобождается.

Третий операнд операции может быть опущен. В этом случае операция выполняется, как если бы

e3 указывало на пустой указатель. Единственное различие заключается в том, что созданный в ходе выполнения *N*-элементный массив освобождается.

Таким образом, `dx[]=::x[]` в строке 29 приводит к разделению *N*-элементного массива `x` массивов из *N* вещественных чисел на *nprocs* подмассивов таким образом, что длина *i*-го подмассива равна значению компоненты `myn`, принадлежащей *i*-му виртуальному процессору (то есть значению `[w:I==i]myn`). Эта же операция будет выполнена быстрее, если использовать форму `dx=:&ns:x`, что позволит избежать дополнительных (и излишних в этом случае) коммуникаций и вычислений необходимых для формирования опущенных операндов.

Асинхронный оператор (то есть оператор не требующий коммуникаций между виртуальными процессорами) в строках 30-32 параллельно вычисляет соответствующую порцию результирующей матрицы *Z* на каждом из виртуальных процессоров сети `w`.

И, наконец, оператор в строке 32 собирает эти порции на виртуальном хост-процессоре, формируя результирующий массив *z*, с помощью оператора сборки `::=`. Эта 4-х местная операция соответствует операции рассылки и выполняет подобным образом обратные коммуникационные операции.

3. Реализация *mpC*

В настоящее время, система программирования *mpC* включает компилятор, систему поддержки времени выполнения, библиотеку и командный пользовательский интерфейс.

Компилятор транслирует текст исходной программы на *mpC* в ANSI C программу с обращениями к функциям системы поддержки времени выполнения. Используется либо SPMD модель целевого кода, когда все процессы, составляющие параллельную программу выполняют одинаковый код или квази-SPMD модель, когда исходный *mpC* файл транслируется в два различных файла - один для виртуального хост-процессора и второй для остальных виртуальных процессоров.

Система поддержки времени выполнения управляет вычислительным пространством, которое состоит из некоторого числа процессов, выполняющихся на целевой вычислительной машине с распределенной памятью (например, на сети ПК и рабочих станций), а также обеспечивает передачу сообщений. Она полностью инкапсулирует конкретный коммуникационный пакет (в настоящее время, MPI 1.1) и обеспечивает независимость компилятора от конкретной целевой платформы.

Библиотека состоит из функций, поддерживающих отладку программ, доступ к характеристикам вычислительного пространства, а также доступ к эффективным средствам низкого уровня.

Командный пользовательский интерфейс состоит из некоторого числа команд для создания виртуальной вычислительной машины с распределенной памятью и выполнении *mpC*-программ на этой машине. При создании виртуальной параллельной машины ее топология (в частности, число и производительности процессоров, характеристики коммуникационных связей между процессорами) определяется автоматически в процессе выполнения специальной тестовой программы и полученная информация сохраняется в специальном файле, используемом в дальнейшем системой поддержки времени выполнения.

3.1 Модель целевой программы

Все процессы, составляющие выполняющуюся целевую программу, разбиваются на две группы - специальный процесс, называемый диспетчером и играющий роль управляющего вычислительным пространством, и обычные процессы, называемые узлами и играющие роль виртуальных процессоров вычислительного пространства. Диспетчер работает как сервер, принимая запросы от узлов. Диспетчер не принадлежит вычислительному пространству.

В целевой программе каждая сеть или подсеть исходной *mpC*-программы представляется некоторым множеством узлов, называемым областью. В любой момент времени выполнения целевой программы каждый узел либо является свободным, либо включенным в одну или несколько областей. Узел, включенный в какую-либо область, называется занятым. Включением узлов в область и их освобождением занимается диспетчер. Единственным исключением является специальный узел, называемый хостом и играющий роль виртуального хост-процессора. Непосредственно после инициализации вычислительное пространство представляется хостом и множеством свободных узлов.

Основная проблема в управлении процессами состоит в их включении в область и освобождении. Решение этой проблемы определяет общую структуру целевого кода и формирует требования к функциям системы поддержки времени выполнения.

Для создания области, представляющей сеть, (сетевой области) родительский узел вычисляет, если надо, топологические характеристики сети и передает диспетчеру запрос на создание области. Запрос содержит число узлов и их относительные производительности. Со своей стороны,

диспетчер хранит информацию относительно целевой сети компьютеров, включающую число актуальных процессоров, их относительные производительности и число узлов на актуальных процессорах. Базируясь на этой топологической информации, диспетчер выбирает множество свободных узлов, наиболее подходящих для создаваемой сетевой области. После этого диспетчер посылает всем свободным узлам сообщение, говорящее, включены они в создаваемую область или нет.

Для освобождения сетевой области ее родительский узел посылает запрос диспетчеру на освобождение области. Заметим, что этот родительский узел остается включенным в родительскую сетевую область освобождаемой области. Остальные узлы освобождаемой области становятся свободными и начинают ожидать команду от диспетчера.

Каждый узел может определить, занят ли он. Узел является занятым, если вызов на нем функции `MPC_Is_busy()` возвращает 1, и свободным, если вызов возвращает 0.

Каждый узел может определить, занят ли он в конкретной области или нет. Доступ к области осуществляется через ее дескриптор. Если дескриптор *rd* соответствует некоторой области, то любой узел вычислительного пространства принадлежит этой области тогда и только тогда, когда вызов функции `MPC_Is_member(&rd)` на этом узле возвращает значение 1. В этом случае дескриптор *rd* позволяет узлу получать всю информацию о соответствующей области, а также идентифицировать себя внутри нее.

Когда свободный узел включается в сетевую область, диспетчер должен сказать ему, в какую именно область он включается, то есть указать ее дескриптор. Простейший способ - разослать указатель на дескриптор от родительского узла не применим для машин с распределенной памятью, не имеющих общего адресного пространства. Поэтому необходим дополнительный идентификатор создаваемой сети, имеющий одинаковое значение на родительском и свободных узлах и имеющий форму, допускающую пересылку от родительского узла к свободным через диспетчер.

В исходной *mpC* программе сеть указывается своим именем, которое является обычным идентификатором и подчиняется обычным правилам видимости языка C. Поэтому имя сети не может служить уникальным идентификатором сети даже внутри одного файла. Можно перенумеровать все сети внутри файла и использовать этот номер как уникальный идентификатор сети в файле. Однако такой уникальный внутри файла идентификатор не будет уникальным во всей программе, которая может состоять из нескольких исходных файлов. Однако он может использоваться при создании сети, если все узлы, участвующие в создании сети выполняют целевой код, соответствующий одному и тому же исходному файлу *mpC* программы. Такая схема и используется в нашем компиляторе. Все сети в файле нумеруются, а структура программы гарантирует, что при создании сети все участвующие узлы выполняют код, содержащийся в одном и том же файле.

В создании области, представляющей сеть (сетевой области), участвуют родительский узел, диспетчер и все свободные узлы. При этом родительский узел вызывает функцию

```
MPC_Net_create(MPC_Name name, MPC_Net* net);
```

где *name* содержит уникальный номер создаваемой сети в файле и *net* указывает на соответствующий дескриптор области. Функция вычисляет всю необходимую топологическую информацию и посылает запрос на создание сети диспетчеру.

В это время свободные узлы ожидают команды от диспетчера в так называемой точке ожидания, вызвав функцию

```
MPC_Offer(MPC_Names* names, MPC_Net** nets_voted, int voted_count);
```

где *names* это массив номеров сетей, создание которых возможно в этой точке ожидания, *nets_voted* - массив указателей на дескрипторы областей, создание которых возможно в этой точке ожидания, и *voted_count* содержит число таких областей.

Соответствие между номером сети и дескриптором области устанавливается следующим образом. Если свободный узел получил сообщение, что он включен в сеть с номером, равным *names*[*i*], то он включен в сетевую область, на дескриптор которой указывает *nets_voted*[*i*].

Свободные узлы возвращаются их функции ожидания `MPC_Offer` либо после того, как их включают в сетевую область, либо после того, как они получают от диспетчера команду покинуть точку ожидания.

3.2 Структура целевого кода для *mpC* блока

В общем случае, в коде целевого блока с определениями сети имеются две точки ожидания. В первой точке, называемой создающей точкой ожидания, свободные узлы ожидают команд диспетчера, связанных с созданием сетевых областей, а во второй, называемой освобождающей, - команд, связанных с их освобождением. В промежутке между этими точками, свободные узлы выполняют код, не связанный с созданием или освобождением сетей, определенных в исходном *mpC*-блоке, а именно, участвуют в тотальных вычислениях и/или в создании и освобождении сетей, определенных во вложенных *mpC*-блоках. Первый оператор

исходного *mpC*-блока, требующий участия в его выполнении свободных узлов, называется оператором разрыва точки ожидания. Таким образом, в общем случае целевой блок имеет следующий вид:

```
{
   объявления, соответствующие
объявлениям переменных в исходном
   mpC-блоке
   {
     if(!MPC_Is_busy()) {
       целевой код, выполняемый
       свободными узлами для создания
       сетевых областей для сетей,
       определенных в исходном mpC-блоке
     }
     if(MPC_Is_busy()) {
       целевой код, выполняемый занятыми
       узлами для создания областей для
       сетей и подсетей, определенных
       в исходном mpC-блоке, а также
       целевой код для операторов
       исходного mpC-блока,
       предшествующих оператору разрыва
       точки ожидания
     }
     эпилог создающей точки ожидания
   }
   целевой код для операторов исходного
   mpC-блока, начинающихся с оператора
   разрыва точки ожидания
   {
     целевой код, выполняемый занятыми
     узлами для освобождения сетевых
     областей для сетей и подсетей,
     определенных в исходном mpC-блоке

     метка освобождающей точки ожидания:

     if(!MPC_Is_busy()) {
       целевой код, выполняемый
       свободными узлами для
       освобождения сетевых областей для
       сетей, определенных в исходном
       mpC-блоке
     }
     эпилог освобождающей точки
     ожидания
   }
}
```

Если же в исходном *mpC*-блоке нет оператора разрыва точки ожидания, то есть, если внутри него (и во вложенных в него блоках) нет тотальных операций и нет вложенных блоков, в которых бы были определения сетей, то в целевом блоке удается совместить создающую и освобождающую точки ожидания, слив их в одну общую точку ожидания. В этом случае целевой блок имеет следующий вид:

```
{
  объявления, соответствующие
  объявлениям переменных в исходном
  mpC-блоке
  {
    пролог общей точки ожидания

    метка общей точки ожидания> 0  2:

    if(!MPC_Is_busy()) {
      целевой код, выполняемый
      свободными узлами для создания и
      освобождения сетевых областей для
      сетей, определенных в исходном
      mpC-блоке
    }
    if(MPC_Is_busy()) {
      целевой код, выполняемый занятыми
      узлами для создания и освобождения
      областей для сетей и подсетей,
      определенных в исходном mpC-блоке,
      а также целевой код для операторов
      исходного mpC-блока
    }
    эпилог общей точки ожидания
  }
}
```

Для того чтобы во время создания сетевой области все вовлеченные узлы выполняли код, расположенный в одном файле, компилятор помещает в эпилог точки ожидания глобальный барьер.

Согласованное прибытие узлов в эпилоге точки ожидания обеспечивается следующим сценарием:

- хост убеждается, что все другие занятые узлы, которые могли послать запрос на создание или освобождение, ожидаемый в точке ожидания, уже достигли эпилог;
- после этого хост посылает диспетчеру сообщение о том, что больше не будет запросов на создание или освобождение, ожидаемых в точке ожидания, и достигает эпилог;
- после получения этого сообщения диспетчер посылает всем свободным узлам команду покинуть точку ожидания;
- после получения этой команды все свободные узлы выходят из функции ожидания и достигают эпилог.

3.3 Алгоритм отображения

Выше мы отмечали, что для создание сетевой области диспетчер выбирает свободные узлы, наиболее подходящие для создаваемой области. В данном разделе объясняется как диспетчер осуществляет этот выбор.

Для каждого сетевого типа в исходной *mpC* программе компилятор генерирует 6 топологических функций, которые используются во время выполнения программы для вычисления различной топологической информации о сетевом объекте данного типа. Первая функция возвращает общее число узлов в сетевом объекте. Так как система поддержки времени выполнения использует линейную нумерацию узлов от 0 до *n*-1, где *n* - общее число узлов, то вторая и третья функции преобразуют координаты узлов в

линейный номер и наоборот. Четвертая функция возвращает линейный номер родительского узла. Пятая функция возвращает относительную производительность указанного узла. И, наконец, шестая функция возвращает длину направленного линка, соединяющего указанную пару узлов.

Когда диспетчер отображает виртуальные процессоры создаваемой сети во множество свободных узлов, он, прежде всего, вызывает соответствующие топологические функции для вычисления относительных производительностей виртуальных процессоров создаваемой сети, а также длины линков между ними. Относительная производительность виртуального процессора в создаваемой сети представляется положительным вещественным числом, нормированным относительно родительского виртуального процессора (относительная производительность которого в создаваемой сети полагается равной 1). Затем диспетчер вычисляет абсолютную производительность каждого виртуального процессора создаваемой сети, умножая его относительную производительность на абсолютную производительность родительского виртуального процессора.

С другой стороны, диспетчер хранит карту вычислительного пространства, отражающую его топологические свойства. Начальное состояние этой карты формируется во время инициализации диспетчера и содержит следующую информацию:

- число физических вычислительных узлов, составляющих вычислительную машину с распределенной памятью, и их производительности;
- число узлов вычислительного пространства (процессов), связанных с каждым из физических вычислительных узлов;
- для каждой пары физических узлов время инициализации обмена сообщениями и время передачи одного байта;
- для каждого физического узла время инициализации обмена сообщениями и время передачи одного байта между парой процессов, выполняющихся на этом физическом узле.

В целом, производительность физического вычислительного узла характеризуется двумя атрибутами. Первый атрибут задает скорость выполнения одного процесса на данном физическом вычислительном узле, а второй задает максимальное число невзаимодействующих процессов, которые можгут выполняться одновременно на данном физическом вычислительном узле без потери в скорости (то есть задает масштабируемость физического вычислительного узла). Например, если в качестве физического вычислительного узла используется мультипроцессор, то значение второго атрибута для него будет равно числу процессоров.

В настоящее время процедура выбора узлов вычислительного пространства для размещения на них виртуальных процессоров создаваемой сети основывается на следующей простейшей схеме.

Диспетчер приписывает вес, являющийся положительным вещественным числом, каждому виртуальному процессору размещаемой сети таким образом, что:

- чем мощнее виртуальный процессор, тем больший вес он получает;
- чем короче линки, выходящие из виртуального процессора, тем больший вес он получает.

Эти веса нормализуются таким образом, чтобы вес виртуального хост-процессора был равен 1.

Аналогично, диспетчер модифицирует карту вычислительного пространства таким образом, чтобы каждый физический вычислительный узел получил вес, являющийся положительным вещественным числом, характеризующим его вычислительную и коммуникационную мощность при выполнении одного процесса.

Диспетчер выбирает свободные узлы вычислительного пространства для размещения на них виртуальных процессоров последовательно, начиная с виртуального процессора, имеющего наибольший вес. Он старается выбрать наиболее мощный свободный узел для этого виртуального процессора. Для этого диспетчер оценивает мощность свободного узла для каждого физического вычислительного узла следующим образом. Пусть w - это вес физического вычислительного узла P, а N - его масштабируемость. Пусть множество H несвободных узлов, связанных с P, разбито на N подмножеств $h_1,...,h_N$; v_{ij} - вес виртуального процессора, размещенного в данный момент на j-ом узле множества h_i;. Тогда оценка мощности свободного узла, связанного с P, вычисляется по формуле

$$\max_i \left\{ \frac{w}{v + \sum_j v_{ij}} \right\}$$

Таким образом, диспетчер выбирает один из свободных узлов того физического вычислительного узла, для которого эта оценка оказывается максимальной, для размещения на нем очередного виртуального процессора (максимального по мощности из еще не размещенных), и добавляет этот узел к подмножеству h_k множества занятых узлов соответствующего вычислительного узла, где значение k определяется из следующего соотношения:

$$\frac{w}{v + \sum_j v_{kj}} = \max_i \left\{ \frac{w}{v + \sum_j v_{ij}} \right\}$$

Описанная процедура оказывается достаточно хорошей для сетей рабочих станций и персональных компьютеров и обеспечивает оптимальное размещение виртуальных процессоров сети, если в точности один узел вычислительного пространства связан с каждым из физических процессоров.

4. Оценка производительности

Как было показано выше, язык *mpC* позволяет программисту определить абстрактную неоднородную сеть, наиболее подходящую для выполнения соответствующего параллельного алгоритма, и система программирования языка *mpC* обеспечивает отображение абстрактной сети на реально существующую. Отображение осуществляется во время выполнения и основывается на информации об относительной производительности процессоров и линий связи реальной сети. Эффективность использования потенциала производительности существующей сети параллельной программой напрямую зависит от качества этого отображения, которое в свою очередь зависит от точности оценки производительности процессоров и сетевого оборудования.

Первоначально поддерживался только один способ оценки производительности процессоров и оборудования. Оценка производилась путем выполнения некоторой специальной оценочной программы и являлась частью выполнения одной из команд пользовательского интерфейса, внешнего по отношению к языку *mpC*.

Основная слабость такого механизма заключается в использовании статической интегральной оценки производительности оборудования. Эта оценка, используемая при создании сетей, определенных внутри программы, не зависит от кода программы и является постоянной во время ее выполнения. Однако, как правило, параллельный код, выполняемый на каждой сети, существенно отличается от смеси команд в оценочной программе. Это различие не очень важно при оценке производительности микропроцессоров одинаковой архитектуры, но для микропроцессоров различной архитектуры оценки производительности, полученные с помощью различной смеси команд, могут различаться очень существенно. Как результат, в каждом конкретном случае создания сети используемая оценка становится достаточно приблизительной, что приводит к недостаточной балансировке загрузки процессоров и, как следствие, к снижению эффективности программы.

Мы исследовали 4 подхода к улучшению точности оценки производительности процессоров. Первый подход заключался в классификации *mpC* приложений и использовании отдельной оценочной программы для каждого класса приложений. Этот подход был отвергнут, так как эксперименты показали, что производительность процессоров может быть существенно разной даже для приложений одного класса. Рассмотрим два фрагмента программы на C

```
     for(k=0; k<500; k++) {
      for(i=k, lkk=sqrt(a[k][k]); i<500;
i++)
       a[i][k]/=lkk;
      for(j=k+1; j<500; j++)
       for(i=j; i<500; i++)
        a[i][j]-=a[i][k]*a[j][k];
     }
```

и

```
     for(k=0; k<500; k++) {
      for(j=k, lkk=sqrt(a[k][k]); j<500;
j++)
       a[k][j]/=lkk;
      for(i=k+1; j<500; i++)
       for(j=i; j<500; j++)
        a[i][j]-=a[k][j]*a[k][i]
     }
```

оба реализующие разложение Холесского для матрицы 500x500. Если использовать их в качестве оценочного кода, то первый оценивает относительную производительность SPARCstation-5 и SPARCstation-20 как 10:9, а второй - как 10:14. Заметим, что функция dpotf2 пакета LAPACK, решающая эту же проблему, оценивает их относительную производительность как 10:10.

Второй подход лежал в использовании специального оценочного кода для каждого *mpC* приложения. В частности, рассматривалась проблема автоматической генерации оценочного кода по исходному коду приложения. Такой подход требует очень значительного усложнения системы программирования языка *mpC*, однако он не работает, если задача, решаемая программой, распадается на несколько подзадач, каждая из которых решается на отдельной сети. В этом случае оценка производительности процессоров получается усреднением оценок, полученных на различных параллельных частях программы, и может сильно отличаться от них, как и в первом подходе.

Третий подход заключается в автоматической генерации для каждой *mpC* программы такого оценочного кода, который давал бы векторную оценку производительности процессоров, при котором каждая параллельная часть программы, выполняемая на отдельной сети, характеризовалась бы своей оценкой, которую бы системы программирования использовала для создания этой сети. Этот подход очень сложен в реализации и не работает в том случае, если реальная сеть, используемая для вычислений, активно используется также и для других

вычислений. В этом случае, с точки зрения *mpC* программы, реальная производительность процессора является функцией времени. Поэтому использование сколь угодно сложной статической оценки, не изменяемой во время выполнения программы, часто ведет к искажению реальной производительности процессоров и, следовательно, к замедлению программы.

Поэтому для реализации в системе программирования *mpC* был выбран четвертый подход, который заключается во введении в язык новой конструкции, которая позволяет программисту обновить во время выполнения программы оценку производительности процессоров с помощью наиболее подходящего (с его точки зрения) оценочного кода.

Таким образом, для наиболее точной оценки производительности процессоров в языке *mpC* был введен оператор

```
recon benchmark
```

Здесь `benchmark` либо пустой оператор состоящий только из точки с запятой, либо оператор общего вида, распределенный по всему вычислительному пространству, который определяет выполнение одинакового кода на всех виртуальных процессорах и не задает коммуникаций между ними. С помощью этого предложения обновляется информация об относительной производительности процессоров сети, используемая системой программирования *mpC*. Если `benchmark` является пустым оператором, то используется стандартный оценочный код, в противном случае в качестве оценочного кода используется код, указанный пользователем в `benchmark`.

Кроме того, в библиотеке стандартных функций *mpC* есть несколько функций, которые позволяют получить текущую оценку производительности процессоров и явно задать эту оценку (без выполнения оценочного кода).

Оператор `recon` и соответствующие библиотечные функции достаточно просты как в использовании, так и в реализации. Для демонстрации его использования рассмотрим следующую подпрограмму

```
/*1 */ #define N 100
/*2 */ nettype Grid(P,Q) {
/*3 */   coord I=P, J=Q;
/*4 */ };
/*5 */ int [net Grid(p,q) v]
mpC2Cblacs_gridinit(int*, char*);
/*6 */ void [*]Cholesky(repl int P, repl
int Q) {
/*7 */    {
/*8 */     int n=N,info;
/*9 */     double a[N][N];
/*10*/     init(a);
/*11*/     recon
dpotf2_("U",&n,a,&n,&info);
/*12*/    }
/*13*/    {
/*14*/      net Grid(P,Q) w;
/*15*/      [w]: {
/*16*/        int ConTxt;
/*17*/
([(P,Q)w])mpC2Cblacs_gridinit(&ConTxt,"R
");
/*18*/        pdlltdriver1_(&ConTxt);
/*19*/
mpC2Cblacs_gridexit(ConTxt);
/*20*/      }
/*21*/    }
/*22*/ }
```

реализующую разложение Холесского квадратной матрицы. В ней неоднородное распределение процессов, участвующих в вычислениях, по процессорам выполняется средствами *mpC*, а собственно параллельное разложение Холесского выполняется средствами пакета ScaLAPACK [10].

Оператор `recon`, определенный в строке 11, определяет производительности реальных процессоров с помощью разложения Холлеского подпрограммой пакета LAPACK `dpotrf`. Код этой подпрограммы является хорошим приближением для кода, который будет выполняться на каждом из узлов. Он обновляет во время выполнения информацию о производительности процессоров с помощью выполнения указанных вычислений (вызов функции `dpotrf2`) в качестве оценочного кода.

Сеть *w*, выполняющая соответствующие параллельные вычисления, определена в строке 14 (ее тип определен в строках 2-4). Она состоит из `PxQ` виртуальных процессоров по умолчанию одинаковой производительности. Ее родитель - виртуальный хост-процессор имеет по умолчанию координаты I=0, J=0. Во время выполнения это определение сети *w* приводит к такому отображению ее виртуальных процессоров на процессы выполняющейся параллельной программы, что число процессов, участвующих в вычислениях на каждом реальном процессоре, пропорционально его производительности, только что оцененной во время выполнения.

Немного модифицированный тестовый драйвер пакета ScaLAPACK для разложения Холесского вызывается на сети `w` (строки 15-20). Этот драйвер читает из файла параметры задачи (размеры матрицы и блоков), формирует тестовую матрицу и проводит ее разложение.

В *mpC* есть три вида функций: базовые, сетевые и узловые.

Базовые функции вызываются и выполняются на всем вычислительном пространстве. Только в них могут создаваться сети. Примером базовой функции является функция `Cholesky`. Это определяется конструкцией `[*]` в строке 6, помещенным непосредственно перед именем функции.

Сетевая функция вызывается и выполняется на сети. Функция `mpC2Cblacs_gridinit`, определенная в строке 5, является примером сетевой функции. Она имеет три специальных формальных параметра: `v`, `p`, `q`. Так называемый сетевой формальный параметр `v` соответствует сети, на которой функция выполняется. Специальные формальные параметры `p` и `q` рассматриваются как целые переменные, размазанные по сети `v`. Они являются параметрами сетевого типа `Grid(p,q)` сети `v`. В строке 17 эта функция вызывается с сетью `w` и параметрами ее типа `P` и `Q` как аргументами, соответствующими описанным выше специальным формальным параметрам.

Узловая функция может быть выполнена на любом одном виртуальном процессоре. В *mpC* функции языка С рассматриваются как узловые.

Если переменная определена без специального определителя распределения, то в базовой функции она рассматривается распределенной по всему вычислительному пространству, а в сетевой функции - по соответствующей сети. Определитель распределения `[w]` в строке 15 задает сеть, выполняющую составной оператор в строках 15-20.

5. Опыт использования mpC

Первая реализация системы программирования *mpC* появилась в конце 1996 года. Уже более двух лет она свободно доступна в Internet (по адресу (http://www.isptas.ru/ mpc). За это время было проведено более 400 инсталляций системы программирования *mpC* во многих странах мира и ряд организаций использует ее, в основном, для проведения научных расчетов на сетях рабочих станций и ПК. *mpC* используется для умножения матриц, решения проблемы N тел, решения задач линейной алгебры (LU разложение, разложение Холесского и т.д.), численного интегрирования, моделирования добычи нефти, расчетов прочности строительных конструкций и многих других. Опыт использования показал, что *mpC* позволяет разрабатывать переносимые модульные параллельные программы, значительно ускоряющие решение как регулярных, так и нерегулярных задач на неоднородных сетях. Кроме того, *mpC* позволяет решать нерегулярные задачи на однородных сетях гораздо быстрее, чем традиционные средства.

В этом параграфе представлены несколько типичных приложений, написанных на *mpC*.

5.1 Нерегулярные приложения, реализованные на *mpC*

Рассмотрим в качестве примера нерегулярной задачи задачу моделирования эволюции системы звезд в галактике (или некоторого множества галактик) под воздействием гравитационного притяжения.

Пусть моделируемая система состоит из некоторого числа больших групп тел. Известно, что поскольку сила взаимодействия между телами быстро уменьшается 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, а наша моделирующая *mpC* программа будет использовать несколько виртуальных процессоров, каждый из которых будет заниматься обновлением данных, характеризующих одну группу тел. Каждый виртуальный процессор должен хранить атрибуты всех тел, образующих соответствующую группу, а также массы и центры тяжести остальных групп. Каждое тело представляется своими координатами, вектором скорости и массой.

Наконец, пусть как число групп, так и количество тел в каждой из групп становятся известны только в период выполнения программы.

mpC программа для решения этой задачи реализует следующую схему:

```
Инициализировать галактику на
виртуальном хост-процессоре
Создать сеть для дальнейших
вычислений и обменов данными
Разослать группы тел по виртуальным
процессорам сети
Вычислить (параллельно) массы групп
Обменяться массами групп между
виртуальными процессорами
while(1) {
  Визуализировать галактику на
  виртуальном хост-процессоре
  Вычислить (параллельно) центры
  масс групп
  Обменяться центрами масс между
  виртуальными процессорами
  Параллельно обновить атрибуты
  групп
  Собрать группы на виртуальном
  хост-процессоре
}
```

Соответствующий *mpC* код выглядит следующим образом:

```
#define MaxGs 30 /* Максимальное число
групп */
#define MaxBs 600 /* Максимальное число
тел в группе */

typedef double Triplet[3];
typedef struct {Triplet pos; Triplet v;
double m;} Body;
```

```
int [host]M; /* Число групп */
int [host]N[MaxGs]; /* Массив количества
тел в группах */
repl dM, dN[MaxGs];
double [host]t; /* Галактические часы */
 /* Массив атрибутов тел галактики*/
Body (*[host]Galaxy[MaxGs])[MaxBs];
nettype GalaxyNet(m, n[m]) {
  coord I=m;
  node { I>=0: n[I]*n[I];};
};

void [host]Input(), UpdateGroup()б
[host]VisualizeGalaxy();
void [*]Nbody(char *[host]infile)
{
  Input(infile); /* Инициализация
Galaxy, M и N */
  dM=M; /* Рассылка числа групп */
 /* Рассылка массива, содержащего числа
тел в группах */
  dN[]=N[];  {
    /* Создание сети g */
    net GalaxyNet(dM,dN) g;
    int [g]myN, [g]mycoord;
    Body [g]Group[MaxBs];
    Triplet [g]Centers[MaxGs];
    double [g]Masses[MaxGs];
    repl [g]i;
    void [net GalaxyNet(m,
n[m])]Mint(double (*)[MaxGs]);
    void [net GalaxyNet(m,
n[m])]Cint(Triplet (*)[MaxGs]);

    mycoord = I coordof body_count;
    myN = dN[mycoord];
    for(i=0; i<[g]dM; i++) /* Рассылка
групп */
      [g:I==i]Group[] = (*Galaxy[i])[];
    for(i=0; i<myN; i++)
        Masses[mycoord]+=Group[i].m;
    ([([g]dM,[g]dN)g])Mint(Masses);

    while(1) {
      VisualizeGalaxy();
      Centers[mycoord][]=0.0;
      for(i=0; i<myN; i++)
        Centers[mycoord][] +=

(Group[i].m/Masses[mycoord])*(Group[i].p
os)[];
      ([([g]dM,[g]dN)g])Cint(Centers);
      ([g]UpdateGroup)(Centers, Masses,
Group, [g]dM);
      for(i=0; i<[g]dM; i++) /* Сбор
групп */
        (*Galaxy[i])[]=[g:I==i]Group[];
    }
  }
}
void [net GalaxyNet(m,n[m]) p]
Mint(double (*Masses)[MaxGs])
{
  double MassOfMyGroup;
  repl i, j;
  MassOfMyGroup=(*Masses)[I coordof i];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Masses)[j] =
        [p:I==j]MassOfMyGroup;
}
void [net GalaxyNet(m,n[m]) p]
Cint(Triplet (*Centers)[MaxGs])
{
  Triplet MyCenter;
  repl i, j;
  MyCenter[] = (*Centers)[I coordof
i][];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Centers)[j][] =
        [p:I==j]MyCenter[];
}
```

Этот код содержит следующие внешние определения:

1. переменных `M`, `t`, и массивов `N`, `Galaxy`, принадлежащих виртуальному хост-процесору;
2. переменной *dM* и массива *dN*, размазанных по всему вычислительному пространству;
3. сетевого типа `Galaxynet`;
4. базовой функции `Nbody` с одним формальным параметром `infile`, принадлежащим виртуальному хост-процессору;
5. сетевых функций `Mint` и `Cint`.

В общем случае, сетевая функция вызывается и выполняется на некоторой сети или подсети, и ее аргументы и возвращаемое значение, если таковые имеются, также распределены по этой же сети или подсети. Заголовок определения сетевой функции либо специфицирует видимый в файле идентификатор статической сети или подсети, либо объявляет идентификатор сети, являющейся специальным формальным параметром функции. В первом случае функция может вызываться только на специфицированной сети или подсети, а во втором - на любой сети или подсети подходящего вида. В любом случае, никакие другие сети, кроме указанной в заголовке определения, в теле сетевой функции создаваться или использоваться не могут. В теле функции могут создаваться только объекты данных, размещенные в связанной с ней сети или подсети, кроме которых могут использоваться и компоненты внешних объектов данных, лежащие в этой области вычислительного пространства. В отличие от базовых функций, сетевые функции (также как и узловые) могут вызываться параллельно.

В коде определяются узловые функции `Input` и `VisualizeGalaxy`, связанные с виртуальным хост-процессором, и узловая функция `UpdateGroup`.

Автоматическая сеть `g`, осуществляющая большую часть вычислений и коммуникаций, определена таким образом, что она состоит из `M` виртуальных процессоров, относительная

производительность которых характеризуется квадратом числа тел в группе, которую он должен обсчитывать.

Таким образом, более мощный виртуальный процессор будет обсчитывать большую группу тел. Система программирования *mpC*, основываясь на этой информации, отобразит виртуальные процессоры, образующие сеть g, в процессы, образующие вычислительное пространство, наиболее подходящим образом. Так как это производится во время выполнения программы, то нет необходимости перетранслировать эту *mpC* программу при переносе ее на другую сеть компьютеров.

Вызов `([`*g*`]`*UpdateGroup*`)(...)` вызывает параллельное выполнение узловой функции *UpdateGroup* на каждом виртуальном процессоре сети *g*. Это означает, что имя функции *UpdateGroup* преобразуется в указатель на функцию, распределенный по всему вычислительному пространству, и оператор `[`*g*`]` вырезает из этого указателя, распределенного по всему вычислительному пространству, указатель, распределенный по сети *g*. Таким образом значением выражения `[`*g*`]`*UpdateGroup* является указатель на функцию, распределенный по *g*, и выражение `([`*g*`]`*UpdateGroup*`)(...)` обозначает распределенный вызов множества нераспределенных функций.

Сетевые функции *Mint* и *Cint* имеют три специальных формальных параметра. Сетевой параметр *p* обозначает сеть, на которой выполняется функция. Параметр *m* рассматривается как целая переменная, размазанная по *p*. Параметр *n* рассматривается как указатель на начальный элемент массива неизменяемых целых, размазанных по *p*. Фактические аргументы, соответствующие описанным формальным параметрам, задаются с помощью синтаксической конструкции `([([`*g*`]`*dM*`,[`*g*`]`*dN*`)`*g*`])`, размещенной слева от имени вызываемой функции в операторе вызова в функции *Nbody*.

Время выполнения *mpC* программы сравнивалось с ее тщательно написанным *MPI* аналогом. В качестве сети компьютеров использовались три рабочих станции SPARCstation 5(`gamma`), SPARCclassic(`omega`) и SPARCstation 20(alpha), соединенные 10Mbits Ethernet. Кроме этих трех, в нашем сегменте локальной сети было еще около 23 рабочих станции. В качестве коммуникационной платформы использовался *LAM MPI* [12] версия 5.2.

Вычислительное пространство системы программирования *mpC* состояло из 15 процессов - по 5 на каждой рабочей станции. Диспетчер выполнялся на `gamma` и использовал следующие относительные производительности, полученные автоматически при создании виртуальной параллельной машины: 1150(`gamma`), 331(`omega`) и 1662(alpha).

Программа на *MPI* была написана таким образом, чтобы минимизировать накладные расходы на коммуникации. Во всех экспериментах рассматривались 9 групп тел. Три процесса MPI программы выполнялись на `gamma`, один на `omega` и пять на `alpha`. Такое отображение является оптимальным, если число тел во всех группах одинаково.

Было проведено два эксперимента. Первый эксперимент сравнивал *mpC* и *MPI* программы для однородных входных данных, когда число тел во всех группах одинаково. Фактически, он показывал сколько мы платим за использование *mpC* вместо *MPI*. Оказалось, что время выполнения *MPI* программы составляет примерно 95% от времени выполнения *mpC* программы. То есть, в этом случае мы теряем только около 5% от производительности.

Второй эксперимент сравнивал эти программы для неоднородных исходных данных. Группы состояли из 10, 10, 10, 100, 100, 100, 600, 600, 600 тел. Время выполнения *mpC* программы не зависит от порядка чисел. Во всех случаях диспетчер отображает:

- на `gamma`, 4 процесса для виртуальных процессоров сети `g`, обсчитывающих две группы из 10 тел, одну группу их 100 тел и одну группу из 600 тел;
- на `omega`, 3 процесса для виртуальных процессоров сети `g`, обсчитывающих одну группу из 10 тел, две группы из 100 тел;
- на `alpha`, 2 процесса для виртуальных процессоров сети `g`, обсчитывающих две группы из 600 тел.

Время моделирования 15 часов эволюции галактики *mpC* программой составляет 94 сек.

Время выполнения *MPI* программы существенно зависит от порядка числа тел в группах и меняется от 88 сек до 391 сек при моделировании 15 часов эволюции галактики. Рис. 1 показывает отношение времен выполнения *MPI* и *mpC* программ для различных перестановок этих чисел. Все возможные перестановки могут быть разбиты на 24 непересекающихся подмножества одинаковой мощности таким образом, что две перестановки принадлежат одному подмножеству, если время их выполнения совпадает. Пусть эти подмножества пронумерованы таким образом, что подмножество с большим номером соответствует перестановкам с большим временем выполнения *MPI* программы. На рис. 1 каждое такое подмножество представлено столбцом, высота которого эквивалентна

соответствующему отношению времен t_{MPI} / t_{mpC}.

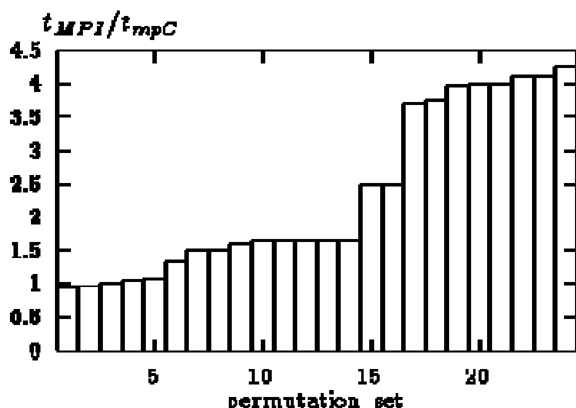

Рис. 1. Ускорение, полученное для различных сочетаний числа тел в группах.

Видно, что почти для всех исходных данных время выполнения MPI программы превышает (и часто, значительно) время выполнения mpC программы.

5.2 Регулярные приложения, реализованные на mpC

Отличительной особенностью нерегулярной задачи является то, что она естественным образом разбивается на сравнительно небольшое число разных по объему вычислений подзадач. Это естественное разбиение, в свою очередь, приводит к некоторому естественному параллелизму при решении задачи, когда вся программа представляет собой небольшое число параллельных взаимодействующих процессов, каждый из которых решает свою подзадачу. Примером нерегулярной задачи служит рассмотренная в разделе 5.1 задача моделирования эволюции галактики.

Регулярная задача отличается тем, что она естественным образом разбивается на сравнительно большое число одинаковых по объему вычислений небольших однородных задач. Это разбиение приводит к такому естественному параллелизму при решении регулярной задачи, при котором вся программа разбивается на большое число небольших одинаковых программ, выполняющихся параллельно и взаимодействующих через обмен данными. Примером регулярной задачи является рассмотренная в разд. 2 задача умножения плотных матриц. Основная идея эффективного решения регулярной задачи на неоднородной вычислительной системе состоит в ее сведении к нерегулярной задаче, структура которой определяется топологией вычислительной системы, на которой решается задача, а не естественной топологией задачи. Это достигается склеиванием небольших, одинаковых по объему вычислений однородных подзадач в более крупные подзадачи, общее число которых не превышает числа доступных физических процессоров, а объемы вычислений пропорциональны мощностям этих процессоров. Поскольку mpC позволяет определять топологию исполняющей вычислительной системы в период выполнения программы, то соответствующая программа может быть написана таким образом, чтобы выполняться эффективно на любой целевой неоднородной вычислительной сети, не требуя изменений исходного кода программы и даже ее перекомпиляции.

В этом подразделе представлен пример решения сложной прикладной регулярной задачи - моделирование процесса нефтедобычи - на неоднородных вычислительных сетях, а более точно, опыт использования *mpC* для переноса соответствующего приложения, написанного на Фортране 77 с использованием *PVM* (около 3000 строк исходного кода), с суперкомпьютера Parsytec PowerXplorer на сеть неоднородных рабочих станций.

Процесс нефтедобычи при заводнении моделировался следующей системой дифференциальных уравнений [12]:

$$m\frac{\partial S_w}{\partial t} + \mathrm{div}(\mathbf{u}_w) = q_w, \qquad q_w = \begin{cases} q_+ \times F_w(\bar{S}) & \text{- на источниках,} \\ q_- \times F_w(S_w) & \text{- на стоках,} \\ 0 & \text{- вне скважин.} \end{cases} \qquad (1)$$

$$\mathrm{div}\big(\boldsymbol{K}(S)\mathrm{grad}\boldsymbol{P}\big) = q. \qquad (2)$$

где

$$\boldsymbol{F}(S) = \frac{k_1(S)/\mu_1}{k_1(S)/\mu_1 + k_2(S)/\mu_2}, \qquad (4)$$

$$\boldsymbol{K}(S) = -k\left(\frac{k_1(S)}{\mu_1} + \frac{k_2(S)}{\mu_2}\right). \qquad (3)$$

Задаются начальные условия (во всем нефтяном пласте)

$$S_\omega|_{t=0} = \underline{S}, \qquad P|_{t=0} = P_0. \qquad (5)$$

$$\left.\frac{\partial S_\omega}{\partial n}\right|_\Gamma = 0, \qquad \left.\frac{\partial P}{\partial n}\right|_\Gamma = 0. \qquad (6)$$

Здесь S_w - водонасыщенность, S_o - нефтенасыщенность, $\underline{S}$ - связанная водонасыщенность, $\bar{S}$ - критическая водонасыщенность, u_w и u_o - скорости фильтрации воды и нефти соответственно, m - коэффициент пористости, k - абсолютная проницаемость пористой среды, $k_w(S_w)$ - коэффициент относительной фазовой проницаемости воды, $k_o(S_w)$ - коэффициент относительной фазовой проницаемости нефти (относительные фазовые проницаемости являются экспериментально измеряемыми функциями насыщенности вытесняющей фазы), $F_w(S_w)$ - функция Баклея-Леверетта для вытесняющей фазы, μ_w и μ_o - коэффициенты динамической вязкости воды и нефти, q (различают q-и $q+$ - объемные источники/стоки жидкости; $q-$ - объем жидкости, извлекаемой в единицу времени на скважине добычи, $q+$ - объем воды, закачиваемой в

единицу времени на скважине накачки, вне скважин q=0), q_w - источники/стоки водонасыщенности, q_o - источники/стоки нефтенасыщенности, t - время, P - давление в пласте (одинаковое для обеих фаз, поскольку капиллярные силы не учитываются).

Уравнения (1)-(2) моделируют фильтрацию двухфазной жидкости, состоящей из нефти и воды, сквозь пористую среду в водонапорном режиме. Уравнение (1) - это уравнение переноса для водонасыщенности, а уравнение (2) - это уравнение диффузии (эллиптического типа) для давления в пласте. Решениями этой системы являются водонасыщенность S_w (фракция воды в двухфазной жидкости) и давление P в пласте (одинаковое для обеих фаз, поскольку капиллярные силы не учитываются). Эти уравнения включают коэффициенты для характеристики среды: коэффициент пористости m, абсолютную проницаемость пористой среды k, коэффициенты относительной фазовой проницаемости воды $k_w(S_w)$ и нефти $k_o(S_w)$, коэффициенты динамической вязкости воды μ_w и нефти μ_o, объемные источники/стоки жидкости q, критическая и связанная водонасыщенности S_ и S , а также существенно нелинейная функция Баклея-Леверетта.

Численное решение искалось в некоторой области симметрии, выделенной из неограниченного и однородного нефтяного пласта, на границах которой ставились естественные условия отсутствия потоков (6). Численный алгоритм основывался на полностью явных методах решения уравнений (1)-(2), а именно, уравнение (1) решалось с помощью итерационного метода секущих, а для решения эллиптического уравнения (2) использовался итерационный (α-β)-алгоритм [12]. С целью ускорения сходимости (α-β)-алгоритма в уравнения для некоторых прогоночных коэффициентов был включен параметр релаксации.

Параллельная реализация алгоритма для выполнения на однородных мультипроцессорных системах с распределенной памятью основывалась на декомпозиции расчетной области (параллелизм данных): область разбивалась на подобласти одинакового размера вдоль Y-координаты, обсчитываемые параллельно разными процессорами исполняющего суперкомпьютера. Такое разбиение оказалось эффективнее как разбиения вдоль X-координаты, так и разбиения по двум направлениям сразу, поскольку обеспечивает меньшую интенсивность обменов данными между процессорами при выполнении параллельного (α-β)-алгоритма. В каждой подобласти уравнения (1)-(2) решались следующим образом. Для каждого временного слоя, водонасыщенность получается решением уравнения (1) с использованием значений давления, полученных для предыдущего временного слоя. Затем, с использованием найденной таким образом водонасыщенность, рассчитывается новое давление для текущего временного слоя путем решения уравнения (2). Потом эта процедура повторяется для следующего временного слоя.

Основная трудность этого параллельного алгоритма заключалась в определении оптимального параметра релаксации омега для параллельного (α-β)-алгоритма, поскольку этот параметр меняется при разбиении расчетной области на разное число подобластей одинакового размера. Использование неоптимальных параметров релаксации приводило к значительному росту числа итераций или даже к потере сходимости параллельного (α-β)-алгоритма. Многочисленные эксперименты позволили найти оптимальное значение параметра релаксации для различного числа подобластей.

Описанный алгоритм был реализован на языке Fortran 77 с использованием коммуникационной библиотеки PVM и продемонстрировал хорошие масштабируемость, ускорение и эффективность распараллеливания при выполнении на параллельном компьютере Parsytec PowerXplorer - многопроцессорной вычислительной системе, использующей процессоры PowerPC 601 в качестве вычислительных узлов и транспьютеры T800 в качестве коммуникационных узлов (один T800 обеспечивает передачу данных со скоростью 20 Мбит/с по 4 двунаправленным линкам).

В табл. 1 представлены результаты, полученные при расчете первого временного слоя. Здесь эффективность распараллеливания определяется как $(S_{real}/S_{ideal}) \times 100\%$, где S_{real} - это фактическое ускорение, достигнутое параллельной программой, а S_{ideal} - идеальное ускорение, которое могло бы быть достигнуто при решении задачи на данной параллельной вычислительной системе. Последнее вычисляется как отношение суммы производительностей процессоров системы к производительности базового процессора. Все ускорения вычисляются относительно времени выполнения исходной последовательной программы на базовом процессоре. Заметим, что эффективность распараллеливания тем выше, чем быстрее коммуникационные линки и слабее процессоры.

Вообще говоря, предполагалось, что параллельная программа моделирования процесса нефтедобычи должна быть частью переносимой программной системы, способной работать на только на суперкомпьютерах, но и на локальных сетях неоднородных компьютеров и обеспечивающей автоматизированное рабочее

место эксперта в области нефтедобычи. Поэтому потребовалась переносимая версия программы, эффективно решающая задачу моделирования нефтедобычи на сетях компьютеров.

Табл. 1. Производительность параллельной Fortran-программы, реализующей неявный алгоритм решения задачи нефтедобычи, при расчетах на многопроцессорной системе Parsytec PowerXplorer в среде программирования PARIX

Число проц.	ω	Число итераций	Время	Ускорение	Эффектив-ность
1	1.197	205	120	1	100%
2	1.2009	211	64	1.875	94%
4	1.208	214	38	3.158	79%
8	1.22175	226	26	4.615	58%

Табл 2. Относительная производительность рабочих станций

Номер рабочей станции	1	2	3-4	5 - 7	8 - 9
Производ тельность	1150	575	460	325	170

В качестве первого шага эта параллельная Fortran/PVM-программа безо всяких изменений была перенесена на локальную сеть, основанную на 10 Mbits Ethernet и состоящую из 9 однопроцессорных рабочих станций SUN. Для сравнения с производительностью процессоров PowerPC-601, на которых базируется Parsytec PowerXplorer, отметим, что самая "слабая" рабочая станция используемой сети (SPARCclassic) выполняет последовательную программу моделирования процессов нефтедобычи немного медленнее, чем PowerPC-601, а самая мощная (UltraSPARC-1) выполняет эту программу более чем в 6 раз быстрее. Относительные производительности рабочих станций при решении задачи нефтедобычи приведены в табл. 2 (станциям присвоены номера, используемые в последующих таблицах).

Табл. 3. Производительность параллельной Fortran/PVM - программы, моделирующей процессы нефтедобычи, при расчетах на подсетях рабочих станций

Подсеть (номера станций)	ω	Число итер.	Время (сек)	Идеал. ускор	Реал. ускор.	Эффек-тивность
{2,5}	1.2009	211	46	1.57	0.88	0.56
{5,6}	1.2009	211	47	2.0	1.52	0.76
{2,5 -7}	1.208	214	36	2.7	1.13	0.42
{2-7}	1.21485	216	32	4.3	1.27	0.30
{2,3,5-8}	1.21485	216	47	3.8	0.87	0.23
{1-8}	1.22175	226	46	3.3	0.41	0.12

В таблице 3 представлены результаты расчетов одного временного слоя с помощью этой Fortran/PVM-программы на различных подсетях этой неоднородной сети, состоящих из двух, четырех, шести и восьми рабочих станций. Эти результаты включают значения параметра релаксации и соответствующие количества β-итераций, время расчета, идеальное и реальное ускорение, а также эффективность использования подсети. Здесь ускорение вычисляется относительно времени выполнения последовательной программы на самой мощной рабочей станции подсети (время выполнения последовательной программы на различных рабочих станциях приведено в таблице 4). Заметное падение эффективности распараллеливания по сравнению с Parsytec PowerXplorer объясняется тремя причинами - более медленными коммуникационными каналами, более быстрыми процессорами и несбалансированной загрузкой процессоров с различной производительностью.

В то время как первые две причины неустранимы, последнюю можно преодолеть путем небольшой модификации параллельного алгоритма, реализуемого Fortran/*PVM*-программой. А именно, для оптимальной загрузки процессоров, расчетная область разбивается на подобласти разного размера, пропорциональные относительным производительностями участвующих в вычислениях процессоров. Точнее говоря, в результате такой неравномерной декомпозиции в каждой подобласти оказывается равное число столбцов расчетной сетки, но разное число строк. Что же касается параметра релаксации, разумно предположить, что его оптимальное значение является функцией числа строк расчетной сетки, и использовать для каждой подобласти свое значение $\omega = \omega(N_{row})$. Экспериментально была найдена последовательность оптимальных значений параметра релаксации для некоторых фиксированных значений N_{row} по которым с использованием кусочно-линейной интерполяции определялось приближенно значение ω для любого числа строк N_{row} расчетной сетки в каждой подобласти. Заметим, что такой подход обеспечил достаточно высокую скорость сходимости параллельного (α-β)-алгоритма с релаксацией (см. "Число итераций" в табл. 5).

Этот модифицированный алгоритм очень непросто выразить средствами *PVM* в переносимой форме. Дело в том, что *PVM*, подобно другим библиотекам передачи сообщений или *HPF*, не поддерживает создание группы процессов, основываясь на таком свойстве создаваемой группы, как относительные скорости процессов. Поэтому, программа, реализующая неоднородный параллельный

алгоритм моделирования процесса нефтедобычи, была написана на языке *mpC*. Эта *mpC*-программа определяет во время выполнения число и относительные производительности доступных процессоров, создает группу процессов таким образом, чтобы каждый процесс выполнялся на отдельном процессоре, и распределяет данные и вычисления пропорционально относительным производительностям процессоров. Заметим, что mpC-программа не только обеспечивает новые функциональные возможности, но и позволяет более чем в 3 раза сократить исходный код (по сравнению с исходной Fortran/*PVM*-программой).

Табл. 4. Производительность программы (последовательной), моделирующей процессы нефтедобычи, при расчетах на рабочих станциях

Процессор	Ultra SPARC-1	SPARC 20	SPARC station 4		SPARC 5			SPARC classic	
Станция	1	2	3	4	5	6	7	8	9
Итерации	205								
Время (сек)	18.7	40.7	51.2	51.2	71.4	71.4	71.4	133	133

Табл. 5. Производительность параллельной *mpC*-программы, моделирующей процессы нефтедобычи, при расчетах на подсетях рабочих станций

Подсеть (Номера станций)	Число Итер.	Время (сек)	Реал. ускор	Эффект.	Время 205 ит. (сек)	Ускор. 205 ит.	Эффект. 205 ит.
{2,5}	324	41.6	0.98	0.63	28.2	1.44	0.92
{5,6}	225	38.8	1.84	0.92	36.4	1.96	0.98
{2,5-7}	279	26	1.57	0.58	19.7	2.07	0.77
{2-7}	245	17.9	2.27	0.54	15	2.71	0.63
{2-8}	248	20.2	2.01	0.54	17	2.39	0.64
{2-8}*	260	32.8	1.24	0.33	26.8	1.52	0.40
{2-9}	268	21	1.94	0.40	16	2.54	0.53

расчетная область была насильно поделена на равные подобласти

Разработанная *mpC*-программа на гетерогенных сетях компьютеров дает умеренные ускорение и эффективность распараллеливания (см. табл. 5), которые, однако, значительно выше аналогичных характеристик Fortran/PVM-программы табл. 3). Несмотря на увеличение числа итераций, mpC-программа позволяет сократить время вычислений за счет оптимизации обменов и, что главное, сбалансированности загрузки процессоров (ср. "время" в табл. 5 и 3). Для оценки чистого выигрыша от балансировки загрузки *mpC*-программа запускалась на одной и той же подсети, состоящей из рабочих станций с номерами {2,3,5,6,7,8}, дважды - с автоматическим распределением данных в соответствии с относительными производительностями процессоров и с насильственным равномерным распределением данных. В последнем случае наблюдались возрастание времени вычислений более чем в 1.5 раза и соответствующая деградация ускорения и эффективности распараллеливания (в табл. 5, помечено *).

Умеренная эффективность распараллеливания mpC-программы во многом объясняется спецификой адаптации (α-β)-алгоритма с релаксацией к гетерогенным сетям. Этот алгоритм очень чутко реагирует на точность определения параметров ω, а описанная выше процедура приближенного вычисления параметров при неравномерной декомпозиции расчетной области дает удовлетворительный, но не наилучший результат. Число итераций, необходимых для сходимости параллельного алгоритма, существенно отличается от числа итераций в последовательном варианте. Поэтому интересно было измерить время, за которое на подсетях совершается 205 итераций -- то количество итераций, за которое β-процесс сходится с указанной выше точностью на одном процессоре. Соответствующие данные представлены в табл. 5, из которой следует, что если бы при распараллеливании удалось избежать возрастания числа итераций (например, за счет более точного определения параметров релаксации), на гетерогенных сетях можно было бы достичь весьма высоких ускорения и эффективности параллельной *mpC*-программы, моделирующей процессы нефтедобычи.

6. Другие работы в этом направлении

Все известные нам программные системы для параллельного программирования на сетях имеют одно общее свойство, которое заключается в том, что при разработке параллельной программы либо программист не имеет средств для описания виртуальной параллельной системы, выполняющей программу, либо эти средства настолько бедны, что не позволяют определить эффективное распределение вычислений и коммуникаций по целевой сети. Даже топологические возможности *MPI* (включая *MPI*-2 [13]) недостаточны для решения этой проблемы. Поэтому для обеспечения эффективного выполнения программы на конкретной сети пользователь должен использовать средства, внешние по отношению к языку, такие как схема загрузки или схема приложения [14]. Если пользователь знаком и с топологией целевой сети (то есть ее структурой и производительностями процессоров и линий связи), и с топологией приложения (то есть ее параллельной структурой), то с помощью конфигурационных файлов он может так отобразить процессы, составляющие программу, на процессоры сети, чтобы обеспечить ее наиболее эффективное

выполнение. Есть системы, которые поддерживают такое статическое отображение [15]. Однако если топология приложения определяется во время выполнения (например, зависит от исходных данных), то такой подход перестает работать.

Существуют системы [16, 17], которые реализуют некоторые функции распределенной операционной системы и стараются учесть неоднородность производительности процессоров сетей компьютеров при управлении задачами с целью максимизации пропускной способности сети компьютеров как одного компьютера. В отличие от этих систем, mpC предназначен для минимизации времени выполнения отдельного параллельного приложения на сети, что и является наиболее важным для конечного пользователя.

7. Заключение

Наиболее общей параллельной архитектурой является сеть компьютеров. В статье описывается язык mpC, предназначенный для эффективно-переносимого модульного параллельного программирования сетей компьютеров, и его система программирования.

К наиболее важными свойствам *mpC* можно отнести следующие:

- однажды разработанное *mpC* приложение будет эффективно выполняться на любой сети компьютеров без какого-либо изменения исходного текста (мы называем это свойство эффективной переносимостью);
- *mpC* позволяет писать приложения, адаптирующиеся не только к номинальной производительности процессоров, но и перераспределяющие вычисления и коммуникации в соответствии с динамическими изменениями загрузки компьютеров сети.

Более двух лет мы экспериментируем с *mpC* и разработали технологию его использования для высокопроизводительных вычислений на неоднородных сетях. Эта технология была успешно применена для решения следующих задач.

- Эффективное использование на неоднородных сетях компьютеров традиционного параллельного программного обеспечения, перенесенного с суперкомпьютеров.
  Примером является интерфейс между *mpC* и ScaLAPACK, позволяющий использовать последний на неоднородных сетях (более детально описано в разделе 4); потребовалась примерно неделя для разработки интерфейса и несколько дней для переноса сложного ScaLAPACK приложения на неоднородные сети с помощью интерфейса.
- Перекодирование на *mpC* с изменениями, обеспечивающими эффективное выполнение на неоднородной сети, суперкомпьютерных параллельных приложение.
  Примером является перенос приложения для моделирования добычи нефти, написанного на Фортране 77 с вызовами *PVM*, с суперкомпьютера Parsytec на сеть неоднородных рабочих станций (более детально описано в подразделе 5.2); на разработку соответствующего *mpC* приложения потребовалось около двух недель; на сети из 8 рабочих станций *mpC* приложение выполняется в три раза быстрее, чем его Фортран/PVM аналог на этой же сети и в два раза быстрее чем Фортран/PVM приложение на 8 процессорном сегменте суперкомпьютера Parcytec.
- Распараллеливанию последовательных программ для выполнения на неоднородных сетях.
  Например, на языке *mpC* была разработана параллельная версия классической адаптивной фортранной программы численного интегрирования quanc8 [13], основанной на квадратурной формуле Ньютона-Кортеса 8-го порядка; в случае вычислительно сложных подынтегральных функций эта *mpC*-программа позволяет существенно ускорить вычисление определенных интегралов, используя доступные компьютеры локальной сети; отметим, что эта программа во время выполнения автоматически перераспределяет объемы вычислений, выполняемые разными компьютерами, в зависимости от их текущей загрузки; разработка программы заняла 2 дня.
- Разработке оригинальных *mpC*-программ. Например, была разработана параллельная программа моделирования на неоднородных сетях эволюции систем тел под воздействием гравитационного притяжения (более детально описано в подразделе 5.1), которая продемонстрировала многократное ускорение по сравнению в тщательно написанной, но не учитывающей неоднородности MPI-программой.

Мы продолжаем работать над *mpC* и его системой программирования. Наша работа направлена как на обеспечение высокой эффективности для широкого диапазона сетей компьютеров (включая кластеры суперкомпьютеров и глобальные сети), так и на улучшение программной модели.

Библиография

1. H.El-Rewini, and T.Lewis, Introduction To Distributed Computing, Manning Publications Co., 1997.
2. Message Passing Interface Forum, MPI: A Message-passing Interface Standard, ver. 1.1, June 1995.

3. A.Geist, A.Beguelin, J.Dongarra, W.Jlang, R.Manchek, V.Sunderam, PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.
4. High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1, Rice University, Houston TX, November 10, 1994.
5. C.Koelbel, "Conferences for Scientific Applications", IEEE Computational Science & Engineering, 5(3), pp.91-95, 1998.
6. S. S. Gaissaryan, and A. L. Lastovetsky, "An ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler", The Journal of C Language Translation, 5(3), March 1994, pp.183-198.
7. Thinking Machines Corporation, "The C* Programming Language", CM-5 Technical Summary, pp. 69-75, November 1992.
8. P. J. Hatcher, and M. J. Quinn, Data-Parallel Programming on MIMD Computers, The MIT Press, Cambridge, MA, 1991.
9. D.Arapov, A.Kalinov, A.Lastovetsky, and I.Ledovskih "Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization", Proceedings of the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98), LNCS 1457, Berkley, CA, USA, August 9-11, 1998, pp.332-343.
10. B.Chetverushkin, N.Churbanova, A.Lastovetsky, and M.Trapeznikova, "Parallel Simulation of Oil Extraction on Heterogeneous Networks of Computers", Proceedings of the 1998 Conference on Simulation Methods and Applications (CSMA'98), the Society for Computer Simulation, November 1-3, 1998, Orlando, Florida, USA, pp. 53-59.
11. A.Kalinov, and A.Lastovetsky, "Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers", Proceedings of the 7'th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99), LNCS 1593, Springer-Verlag, April 12-14, 1999, Amsterdam, the Netherlands pp.191-200.
12. B. N. Chetverushkin, N. G. Churbanova, and M. A. Trapeznikova,"Simulation of oil production on parallel computing systems", Proceedings of Simulation MultiConference HPC'97: Grand Challenges in Computer Simulation, Ed. A.Tentner, Atlanta, USA, April 6-10, 1997, pp.122-127.
13. MPI-2: Extensions to the Message-Passing Interface, http://www.mcs.anl.gov.
14. Trollius LAM Implementation of MPI. Version 6.1, Ohio State University, 1997.
15. F.Heinze, L.Schaefers, C.Scheidler, and W.Obeloeer, "Trapper: Eliminating performance bottlenecks in a parallel embedded application", IEEE Concurrency, 5(3), pp.28-37, 1997.
16. Dome: Distributed Object Migration Environment, http://www.cs.cmu.edu/ Dome/.
17. Hector: A Heterogeneous Task Allocator, http://www.erc.msstate.edu/ russ/hpcc/.