Команда "шаг" в параллельных отладчиках

А.Я. Калинов, К.А. Карганов, К.В. Хоренко

Аннотация. В статье рассматривается новая схема команд перемещения в параллельных отладчиках на примере команды «шаг». Основное отличие предложенной схемы от существующих заключается в том, что отладчик на основе модели параллельной программы анализирует момент завершения «шага» программы, чем приближает отладку параллельной программы к отладке последовательной программы. Рассмотрены проблемы, возникающие при реализации данной схемы для отладчика MPI-программ, и описана ее реализация в отладчике mpC-программ.

1. Введение

Одной из важнейших возможностей, предоставляемых отладчиком, является возможность управления выполнением программы. Семантика команд управления выполнением в последовательных отладчиках (когда отлаживается один процесс с одним потоком управления) достаточно ясна. Например, семантика команды пошагового выполнения последовательной программы заключается в следующем – выполнить операторы текущей строки и прервать выполнение отлаживаемой программы. Если операторы текущей строки не выполнены, то пользователь анализирует причину (например, ожидание взаимодействия с внешним миром, длинный счет и т.д.) и предпринимает или не предпринимает какие-либо действия. Причины невыполнения шага последовательной программы назовем «*последовательными»*. Естественное расширение этой семантики на случай параллельного отладчика (когда отлаживается несколько потоков управления, возможно, в нескольких процессах) – выполнить операторы текущих строк для выделенных потоков управления и прервать их выполнение. Однако, в случае параллельного отладчика, некоторые потоки могут не выполнить шаг не только по *«последовательным»*, но и по *«параллельным»* причинам, связанным с взаимодействием между потоками/процессами параллельной программы.

Большинство существующих параллельных отладчиков являются многоцелевыми, не ориентированными на поддержку какого-либо конкретного параллельного языка программирования или коммуникационного пакета. В них применяется комбинация двух основных схем реализации команды "шаг":

- синхронная – команда выдается для всех процессов группы, ожидается завершение шага для всех процессов группы, имеется возможность принудительного прерывания – основная схема в Mantis [1,2];
- асинхронная – команда выдается для всех процессов группы завершивших предыдущий шаг на данный момент, управление в отладчике передается пользователю без ожидания окончания выполнения команды - основная схема в TotalView [3].

При этом ответственность за анализ и обработку всех «параллельных» причин незавершения шага возлагается на пользователя.

В тоже время, если рассматривать не параллельную программу вообще, а MPI-программы или программы на параллельном языке высокого уровня, то среди *«параллельных»* причин незавершения шага большинство является *«тривиальными»*, обработку которых может производить отладчик, приближая тем самым, насколько это возможно, отладку параллельной программы к отладке последовательной программы. Конечно, это возможно только в том случае, если отладчик понимает семантику параллельной программы и имеет соответствующую поддержку со стороны реализации библиотеки или языка.

В данной статье обсуждается возможность реализации синхронной модели выполнения команды "шаг", которая отличается от традиционной тем, что:

- последовательная команда "шаг" выполняется только теми процессами, которые завершили выполнение предыдущей команды перемещения;
- «тривиальные параллельные» причины незавершения шага обрабатываются отладчиком.

Рассматриваются проблемы, возникающие при реализации такой модели для отладчика MPI-программ, и описывается ее реализация для параллельного отладчика для языка mpC [4,5], который является составной частью интегрированной системы разработки и выполнения mpC программ mpC Workshop. Статья организована следующим образом: в первом параграфе дается определение параллельной команды "шаг", которое демонстрируется на примере простой MPI-программы. Во втором параграфе анализируется возможность учета «тривиальных причин» незавершения шага для MPI-программ. Третий параграф дает введение в язык параллельного программирования mpC. В четвёртом параграфе описывается mpC-отладчик.

2. Синхронная модель команды «шаг» с обработкой «тривиальных причин» незавершения шага на примере MPI-программ

Введём определение команды синхронной команды «параллельного шага». Примем, что для выполнения команды «шаг параллельной программы» группой процессов каждому процессу из группы, завершившему предыдущие команды перемещения, необходимо выполнить действия, определяемые командой «шаг» последовательного отладчика (“последовательной” команды «шаг»). Шаг параллельной программы будем считать завершённым, если каждый процесс отлаживаемой программы:

а) выполнил действия, предписанные последней данной ему “последовательной” командой перемещения (возможно, не командой «шаг»)

или

б) остановился и не может продолжить выполнение без дополнительных действий со стороны пользователя по отношению к другим процессам отлаживаемой программы.

Назовем «тривиальной параллельной» причиной незавершения параллельной команды шаг процессом ожидание синхронизации с другими процессами, про которую известно, что она не произойдет на данном шаге параллельной программы.

Пример 1. Рассмотрим пошаговое выполнение следующей MPI-программы:

```
/* 0*/ #include "mpi.h"
/* 1*/ int main(int argc, char **argv){
/* 2*/    int rank, i;
/* 3*/    MPI_Init(&argc, &argv);
/* 4*/    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* 5*/    if (rank == 1) {
/* 6*/       MPI_Status stat;
/* 7*/
MPI_Recv(&i,1,MPI_INT,0,0,MPI_COMM_WORLD,&stat);
/* 8*/    }
/* 9*/    if (rank == 0) {
/*10*/       i = 0;
/*11*/       MPI_Send(&i,1,MPI_INT,1,0,MPI_COMM_WORLD);
/*12*/    }
/*13*/    MPI_Finalize();
/*14*/    return 0;
/*15*/ }
```

Пусть отладчик является «умным», то есть обрабатывающим «тривиальные параллельные» причины незавершения шага и интерпретирует вычисление управляющего выражения условного оператора как самостоятельный шаг. Пусть, кроме того, команды выдаются для всех процессоров отлаживаемой

программы, и текущей строкой для всех процессов является строка 3. Тогда для последовательности шагов параллельной программы для различных процессов текущими будут следующие строки:

Шаг	Текущие позиции для процессов с rank			Примечание
	0	1	остальные	
1	3	3	3	
2	4	4	4	
3	5	5	5	
4	9	7	9	Процесс с rank равным 1 не может закончить свой шаг по «тривиальной параллельной» причине - управление не может вернуться из функции `MPI_Recv` так как процесс с rank равным 0 еще не вызвал функцию `MPI_Send` в строке 11 и не вызовет ее на этом шаге.
5	10	7	13	Остальные процессы не могут завершить шаг по «тривиальной параллельной» причине - для завершения работы `MPI_Finalize` ее должны вызвать все процессы.
6	11	7	13	Процесс с rank равным 1 процесс должен закончить выполнение `MPI_Recv.`
7	13	9	13	
8	13	13	13	

Таблица 1. Текущие позиции для различных процессов при пошаговом выполнении MPI программы.

В данном примере за восемь шагов параллельной программы процесс с rank равным 0 сделал 7 последовательных шагов, процесс с rank равным 1 сделал 6 последовательных шагов, остальные процессы сделали 5 последовательных

шагов, и пользователю не потребовалось никаких дополнительных действий для этого.

Основные отличия предложенной схемы выполнения команды «шаг» от существующих схем:

1. Первые три шага в параллельном отладчике с синхронной схемой команды «шаг» будут идентичны приведённым выше. Однако пользователь не дождётся завершения шага 4 в таком отладчике, поскольку процесс с rank равным 1 не сможет завершить свою часть команды. Пользователю придётся прервать выполнение команды. В результате он получит набор процессов, остановленных в разных точках программы. Он не будет знать, какой из процессов остановился самостоятельно, выполнив свой элементарный шаг, а какой был прерван пользователем.
2. Если в отладчике реализована асинхронная модель параллельной команды «шаг», то пользователь, после того, как выдал команду и получил управление в отладчике (сразу же после выдачи команды), не может быть уверен, что все процессы завершили выполнение своих частей общей параллельной команды «шаг». Пользователь может убедиться в полном завершении (или незавершении) общей команды лишь исследовав статус каждого процесса вручную и проанализировав их.

Важно отметить, что в обоих рассмотренных случаях после четвёртого шага процесс с rank 1 будет находиться где-то внутри функции `MPI_Recv`. В тоже время отладчик уровня исходного текста языка параллельного программирования должен обеспечивать отладку в терминах самого языка. Отладчик MPI-программ также должен поддерживать отладку в терминах MPI и языка последовательного программирования, из которого вызываются функции MPI. Соответственно, для него функции MPI должны являться неделимыми примитивами. Для пользователя, отлаживающего MPI-программу, получить информацию о том, что один из процессов его программы находится в какой-то внутренней функции, вызванной, например, внутри `MPI_Recv` также неестес-твенно, как для пользователя, отлаживающего последовательную программу на языке высокого уровня, получить информацию, что программа выполняет какую-то из ассемблерных операций. Мы считаем это существенным недостатком существующих схем параллельных команд перемещения, который устраняется в предлагаемом подходе.

3. Подход к учету «тривиальных причин» незавершения шага для MPI-программ

Предлагаемая реализация шага параллельной программы возможна только в том случае, если отладчик понимает семантику функций MPI и имеет модель выполнения MPI-программы. Это невозможно без поддержки отладчиков со

стороны MPI, так же как отладка программ на языке высокого уровня невозможна без поддержки со стороны компилятора. В настоящее время поддержка параллельных отладчиков реализациями MPI ограничивается интерфейсом, разработанным для TotalView [7], который ориентирован только на просмотр очередей сообщений и не позволяет обрабатывать «тривиальные параллельные» причины незавершения шага в MPI-программе.

Современные параллельные отладчики (например, p2d2[6]) имеют следующую клиент-серверную архитектуру:

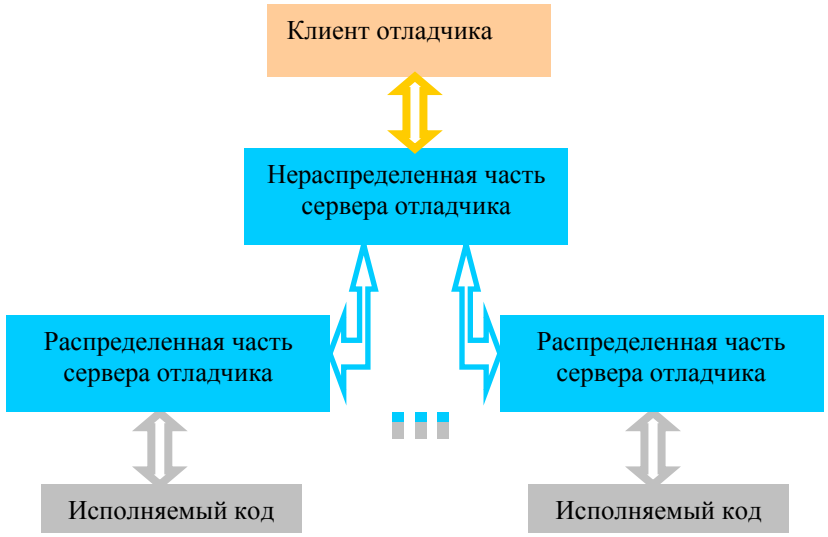


Рис. 1

В ней сервер состоит из двух частей: нераспределенной, которая отвечает за управление параллельной программы как целым, и распределенной, которая отвечает за непосредственное управление выполнением каждого из процессов параллельной программы. Соответственно, нераспределенная часть должна поддерживать модель текущего состояния параллельной программы и, на основании информации, получаемой от распределенных частей обрабатывать «тривиальные параллельные» причины незавершения шага.

Рассмотрим, какой информации не хватает в текущем интерфейсе «MPI – отладчик» [7] на примере функции `MPI_Send`. С помощью этого интерфейса доступна очередь незаконченных посылок для каждого коммуникатора. Элемент этой очереди имеет тип структуры mqs_pending_operation[8] и содержит большое количество информации о посылке. Известно, что реализация MPI самостоятельно принимает решение о том буферизировать, или нет пересылаемое сообщение. В случае буферизации сообщения операция пересылки является локальной и заканчивается вне зависимости от соответс-

твующего запроса на прием. В противном случае она не является локальной и не может закончиться, если не был выдан соответствующий запрос на прием. Для того, чтобы передать эту информацию компоненте отладчика, связанной с конкретным процессом, достаточно просто добавить в структуру mqs_pending_operation еще одно поле, которое говорило бы о том было ли сообщение буферизировано или нет.

Построение полного интерфейса «MPI – отладчик», удовлетворяющего требованиям предлагаемого подхода, не является целью данной работы. Мы просто хотим обратить внимание на то, что он может быть получен путем незначительной доработки уже имеющегося интерфейса.

Мы реализовали предлагаемый подход в отладчике для языка параллельного программирования mpC, который описан ниже.

4. Язык параллельного программирования mpC

Язык параллельного программирования mpC был разработан в конце 90х в Институте системного программирования РАН. В языке mpC, который является расширением языка C, вводится понятие вычислительного пространства, которое определяется как некоторое доступное для управления множество виртуальных процессоров. Основным понятием mpC является понятие сетевого объекта или просто сети. Сеть является областью вычислительного пространства, которая может быть использована для вычисления выражений и выполнения операторов.

Размещение и освобождение сетевых объектов в вычислительном пространстве выполняется аналогично размещению и освобождению объектов данных в памяти. Концептуально, создание новой сети инициируется процессором уже существующей сети. Этот процессор называется родителем создаваемой сети. Родитель всегда принадлежит создаваемой сети. Единственным виртуальным 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 и производительности процессоров. Например, объявление

```
nettype SimpleNet(n) {
      coord I=n;
};
```

вводит параметризованный сетевой тип с именем `SimpleNet`, соответствующий сетям, состоящим из `n` виртуальных процессоров. Виртуальные процессоры сети связаны с координатной переменной `I`, значение которой изменяется от `0` до `n-1`. Родитель сети имеет по умолчанию координату равную 0. Это простейшая сеть, определение которой содержится в стандартном файле `mpc.h`.

Пример 2. Рассмотрим mpC-программу эквивалентную MPI-программе, приведенной в **примере 1**.

```
/* 0*/ #include “mpc.h”
/* 1*/  int [*]main(int argc, char **argv) {
/* 2*/      net SimpleNet(2) w;
/* 3*/      int [w]i;
/* 4*/      [host]i=0;
/* 5*/      [w:I==1]i=[host]i;
/* 6*/      return 0;
/* 7*/ }
```

Имея объявление сетевого типа, можно объявить идентификатор сетевого объекта этого типа. Например, объявление в строке 2 вводит идентификатор `w` сетевого объекта состоящего из `2` виртуальных процессоров.

Объект данных, распределенный по некоторой области вычислительного пространства, составляется из обычных (нераспределенных) объектов одного типа, размещенных в процессорных узлах этой области таким образом, что каждый процессорный узел содержит одну и только одну компоненту распределенного объекта. Например, объявление в строке 3 вводит целую переменную `i`, распределенную по сети `w`. Конструкция [w] представляет собой спецификатор распределения переменной. В данной программе используются спецификаторы распределения `[*]` и `[host]`, соответствующие всему вычислительному пространству и хост-процессору соответственно, а также `[w:I==1]`, выделяющий узел сети w с координатой I, равной 1. Отметим, что хост-процессор имеет в этой сети координату 0.

Оператор в строке 4 присваивает компоненте переменной `i`, расположенной на хост-процессоре, значение 0. Это локальное присваивание.

Присваивание в строке 5 является расширением присваивания языка C. Оно присваивает компоненте переменной `i`, расположенной на узле сети w с координатой I, равной 1, значение компоненты переменной `i`, расположенной на хост-процессоре. Это распределенное присваивание.

Компилятор транслирует текст исходной программы на mpC в ANSI C-программу с обращениями к функциям системы поддержки времени выполнения. Используется SPMD-модель целевого кода, когда все процессы, составляющие параллельную программу, выполняют одинаковый код.

Система поддержки времени выполнения управляет вычислительным пространством, которое отображается на некоторое число 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, подмножество MPI 1.1) и обеспечивает независимость компилятора от конкретной целевой платформы.

5. Отладчик mpC-программ

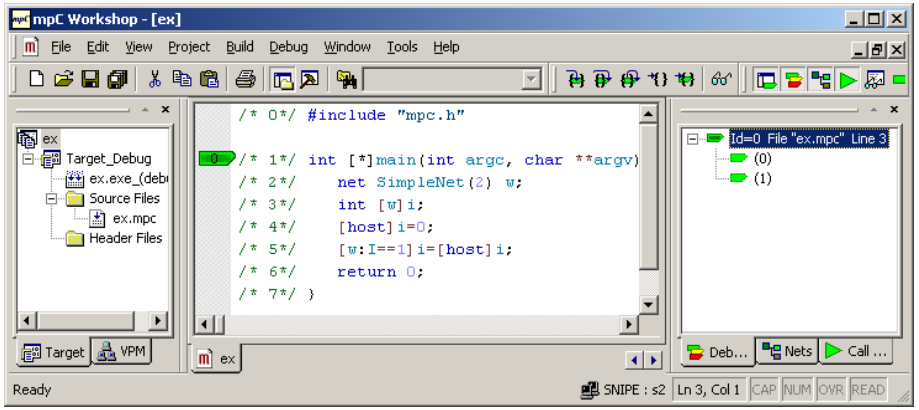
mpC Workshop является интегрированной средой разработки и выполнения mpC программ для Windows, разработанной в Институте системного программирования РАН для компании ATS (www.atssoft.com). Он состоит из набора инструментов, поддерживающих все стадии разработки и выполнения mpC-программ, основным из которых является отладчик mpC-программ.

С каждым виртуальным процессором (процессом параллельной программы) отладчик связывает курсор, который указывает на его текущую выполняемую строку. Курсор имеет цвет, который указывает на его статус. Цвета определяются в соответствии со стилем светофора, то есть, зеленые курсоры могут выполнить команду перемещения, желтые могли бы выполнить команду перемещения, но остановлены пользователем и, наконец, красные курсоры не могут выполнить команду перемещения, так как они не закончили один из предыдущих шагов и ожидают синхронизации с другими виртуальными процессорами для его завершения. Курсоры одного цвета, указывающие на одну и ту же строчку, объединяются в составной курсор.

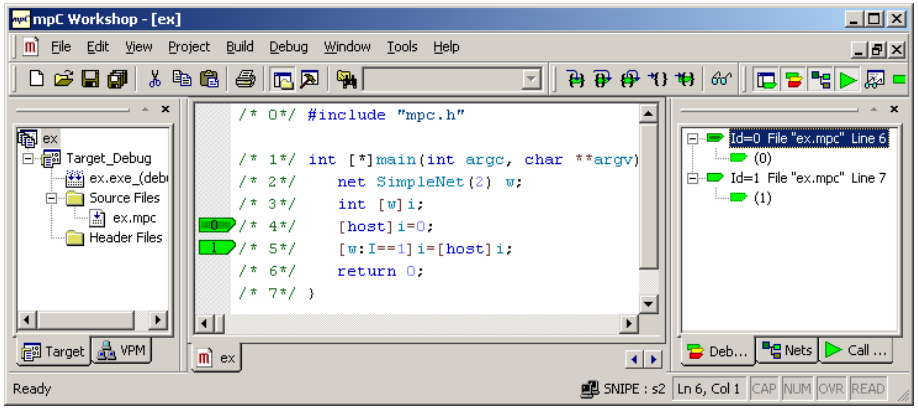
Существует возможность поменять цвет, как у всего составного курсора, так и у некоторых входящих в него курсоров. Можно менять цвет курсора только с желтого на зеленый и наоборот. Красный цвет курсора определяется отладчиком. Далее если это не будет оговорено отдельно, мы не будем делать различия между простым и составным курсором, а также между курсором и виртуальным процессором.

В отладчике mpC принята описанная выше синхронная модель. То есть, отладчик ожидает окончания шага всеми зелеными курсорами. Если отладчик знает, что какой-либо курсор не может закончить текущий шаг по «тривиальной параллельной» причине, он помечает этот курсор красным цветом и не ожидает завершения им последовательного шага на данном шаге параллельной программы. После того, как на каком-то шаге параллельной программы произошло событие, которое позволяет красному курсору закончить незавершенный последовательный шаг, отладчик начинает ожидать его завершения.

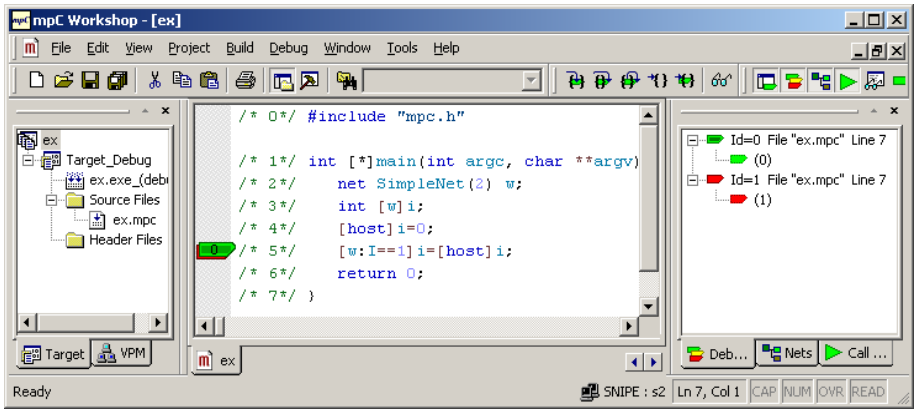
Рассмотрим пошаговое выполнение программы из примера 2. Пусть текущая виртуальная параллельная машина состоит из 2 виртуальных процессоров. В начале сессии отладки существует только один курсор зеленого цвета, показывающий на 1-ю строчку.



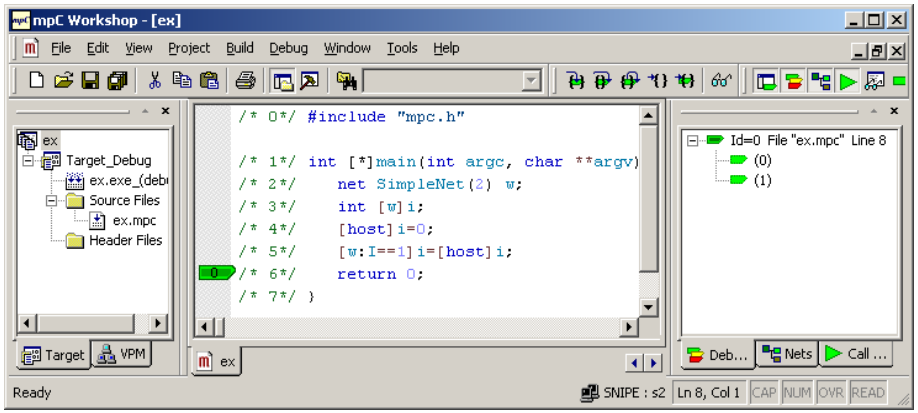
После первого шага появляется два курсора. Первый с номером 0 соответствует хост-процессору, который должен выполнить оператор в строке 4. Второй с номером 1, соответствует узлу сети `w` с координатой 1, который должен выполнить оператор в строке 5.



После второго шага курсор с номером 1 остается в той же позиции, но меняет свой цвет, так как он не может выполнить оператор в строке 5 до тех пор, пока его не начнет выполнять хост-процессор.



После выполнения третьего шага хост-процессором, процесс, которому соответствует курсор с номером 1 также заканчивает выполнение оператора в строке 5 и опять получается только один зеленый курсор.



Отладчик mpC имеет традиционную модель «клиент-сервер». Клиент связан со специальным процессом – менеджером отладки, который получает команды пользователя и управляет выполнением процессов параллельной программы. Когда, при выполнении шага, процесс параллельной программы выполняет действие, требующее синхронизации с другими процессами, информация об этом передается менеджеру отладки. Он понимает семантику соответствующих действий и поддерживает внутри себя модель состояния параллельной mpC программы, которая позволяет ему определять, какие процессы завершат выполнение шага, а какие нет.

Как отмечалось выше, mpC-компилятор транслирует mpC-программу в MPI-программу. Получающаяся в результате MPI-программа является “безопасной” (“safe”) в смысле стандарта MPI [9], то есть, для ее успешного выполнения не требуется буферизация сообщений. Для того чтобы устранить неопределенность, связанную со стандартным режимом, в текущей версии

системы программирования, в отладочном режиме, стандартные команды пересылки заменяются на синхронные. В будущем, когда удастся решить проблему с идентификацией выбранного режима выполнения стандартной команды пересылки, это ограничение будет снято.

6. Заключение

Существует естественное противоречие между общностью параллельного отладчика и сервисом, который он мог бы предоставлять при отладке программ, разработанных с помощью конкретного параллельного языка программирования или коммуникационной платформы. В данной статье мы рассматриваем один из аспектов построения специализированных параллельных отладчиков для конкретной коммуникационной библиотеки (MPI) и конкретного языка параллельного программирования (mpC), а именно, реализацию команд перемещения на примере команды «шаг».

Предлагаемая синхронная схема команды «шаг», с обработкой «тривиальных параллельных» причин незавершения позволяет, освободить пользователя от рутинной работы и, по возможности, приблизить отладку параллельной программы к отладке последовательной программы. Рассмотрена возможность реализации предлагаемой схемы для отладчика MPI-программ и показано, что она может быть реализована при незначительном усложнении существующего интерфейса «MPI – отладчик». Описана реализация предлагаемой схемы в отладчике mpC-программ, который является составной частью интегрированной системы разработки mpC Workshop.

Литература

1. Steven S.Lumetta, David E.Culler, “Mantis User's Guide, Version 1.0”, 1994, http://www.cs.berkeley.edu/projects/parallel/castle/mantis/mantis.tr.html.
2. Steven S. Lumetta, David E. Culler: The Mantis Parallel Debugger, Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools.
3. TotalView Users Guide, Etnus, 2003, http://www.etnus.com/Download/TV.html.
4. Lastovetsky A.: Parallel Computing on Heterogeneous Networks, John Wiley & Sons, New Jersey, 2003, 159-254.
5. A.Lastovetsky, D.Arapov, A.Kalinov, and I.Ledovskih, "A Parallel Language and Its Programming System for Heterogeneous Networks", *Concurrency: Practice and Experience*, 12(13), 2000, pp.1317-1343.
6. Hood, R.: The p2d2 Project: Building a Portable Distributed Debugger. Proceedings of the 2nd Symposium on Parallel and Distributed Tools (SPDT'96). Philadelphia PA, USA, 1996.
7. Cownie, J., Gropp, W.: A standard interface for debugger access to message queue information in MPI. In Dongarra, J., Luque, E., Margalef, T., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 1697 of Lecture Notes in Computer Science., Berlin, Springer (1999) 51—58.
8. http://www-unix.mcs.anl.gov/mpi/mpi-debug/mpi_interface.h.html.
9. The MPI Standard version 1.1 June 12, 1995, http://www.netlib.org/mpi/mpi-1.1-report.ps