

УДК 004.42

КОМПОНЕНТ ПРОВЕРКИ КОРРЕКТНОСТИ ИСПОЛЬЗОВАНИЯ MPI В ПРОГРАММНОМ КОМПЛЕКСЕ S-MPI

А. В. Огородников, Н. А. Побуринная, М. И. Старов
(ООО "Центр компетенций и обучения", г.Саров)

Разрабатываемый инструмент МССТ призван облегчить программистам поиск реальных и потенциальных проблем в параллельном коде, использующем MPI. Данный программный компонент проверки корректности использования MPI входит в состав разрабатываемого комплекса S-MPI — первого подобного коммерческого продукта в России.

На ряде примеров проблемного исходного кода демонстрируется успешное обнаружение проблем с помощью МССТ. Приводятся результаты проверки корректности двух программных пакетов — ЛЭГАК-ДК и ЛОГОС, разработанных в РФЯЦ-ВНИИЭФ.

Замеры производительности МССТ подтверждают его достаточно хорошую конкурентоспособность.

Ключевые слова: МССТ, проверка корректности, MPI, программный комплекс S-MPI.

Введение

Современные *параллельные* программные комплексы, использующие интерфейс MPI (Message Passing Interface) [1], становятся все сложнее; они используют все большее количество вычислительных ядер, а следовательно, процессов MPI. Для улучшения масштабируемости приложений MPI авторы зачастую отказываются от синхронных (блокирующих) обменов MPI, отдавая предпочтение неблокирующим. Также создаются более сложные типы данных для обмена между счетными процессами и другие специализированные ресурсы MPI, такие как *MPI_Comm*, *MPI_Group*, *MPI_Op*, *MPI_Errhandler* и т. д. Чем сложнее приложение, тем за большим количеством буферов и ресурсов MPI приходится следить самим разработчикам, так как компиляторы, как правило, не отслеживают корректность использования MPI. Реализации библиотек MPI также не делают этого в полной мере, так как наличие большого количества проверок противоречит увеличению производительности библиотек.

Вышеперечисленные факты свидетельствуют о том, что вероятность некорректного применения вызовов MPI возрастает (в силу сложности программного комплекса, применяемой в нем схемы коммуникаций, пересылки сложных наборов данных). Наличие ошибок, в свою очередь, может привести к искажению передаваемых данных, *зависанию* программы и т. д.

Как правило, при поиске проблем в программном коде разработчики используют программы-отладчики, позволяющие пошагово исследовать выполнение программы. Но подобное исследование всего параллельного программного кода, в котором идет постоянный межпроцессный обмен, является весьма длительным. Поэтому логично переложить все проверки параллельного кода на вычислительную машину, которая будет выполнять их без задержек и по точно закодированным спецификациям. Это поможет программисту точно определить местонахождение ошибки в коде и сразу заняться ее исправлением.

Представленный компонент проверки корректности использования MPI — МССТ (MPI Correctness Checking Tool) в составе разрабатываемого программного комплекса S-MPI призван значительно

облегчить поиск ошибок в реализации параллельного кода MPI. Текущая версия МССТ содержит более 100 локальных и глобальных (вынесенных в отдельный служебный MPI-процесс) проверок.

Цель разработки

Основной целью разработки МССТ является создание конкурентоспособного инструмента российского производства для проверки корректности использования MPI. Инструмент должен обеспечивать качество и стабильность и характеризоваться:

- отсутствием ложных срабатываний (false positives);
- ясным для разработчика описанием проблем с указанием точного места в программном коде анализируемого приложения MPI;
- полным охватом текущего стандарта MPI-2.

Время получения результатов подобного анализа также достаточно критично для разработчиков в случае, когда необходимо проводить ряд проверок с различной конфигурацией запуска (с разницей в количестве процессов, в используемых коммуникационных средах, алгоритмах обмена коллективных операций MPI и т. д.). Поэтому важно, чтобы МССТ обеспечивал приемлемую производительность в сравнении с конкурентами.

Обзор возможностей

Компонент содержит *оберточные функции* (wrappers) стандарта MPI-2, в которых производятся все необходимые проверки корректности использования MPI и из которых вызывается соответствующая RMPI-функция. Инициализация компонента происходит при вызове функции *MPI_Init* или *MPI_Init_thread*.

При каждом входе в функцию MPI компонентом МССТ проверяется ряд условий, таких как правильность аргументов, соответствие получаемых данных их типу, выполняется сопоставление коллективных операций и т. д. Для проведения так называемых глобальных проверок, которые требуют данных от двух и более процессов, компонент пересылает служебные данные сервисному процессу (автоматически создается при использовании скрипта запуска). При невыполнимости проверяемых условий компонент передает сообщение о найденных проблемах в поток вывода *stdout*.

Инструмент МССТ содержит более 100 локальных и глобальных проверок, в том числе:

- параметров функций MPI; при этом учитывается контекст вызовов для данной функции MPI, а также, возможные ограничения с учетом языковой принадлежности используемой реализации MPI (C, Fortran);
- переносимости исходного кода на другие реализации MPI;
- целостности используемых данных в вызовах MPI;
- своевременного освобождения памяти, задействованной пользовательскими ресурсами MPI;
- возможного наложения памяти, в том числе внутри сложных типов данных MPI;
- совпадения типов пересылаемых данных между процессами;
- коллективных операций, вызванных из вовлеченных в данный коммутатор процессов MPI (сопоставление по ряду признаков);
- возможных потерь двухточечных сообщений.

Кроме того, выполняются проверки возникновения условий взаимоблокировок (deadlocks) с учетом возможных *условий гонки* (race conditions). В отличие от других подобных инструментов диагностируется первопричина случившейся взаимоблокировки с возможностью визуализации блокирующего обмена. При этом используется утилита dot из пакета Graphviz [2] с удобным языком интерпретатора для представления блок-схем в графическом виде.

Инструмент МССТ позволяет отслеживать стек вызовов, который помогает более точно определить проблемное место приложения, и в то же время он не оказывает чрезмерного влияния

на производительность проверяемого приложения MPI, что позволяет ему выгодно отличаться от программ-конкурентов (рисунок). При этом используется API библиотеки Stackwalker из LGPL продукта Dyninst [3].

Чтобы данный компонент включился в работу, необходимо, чтобы он был подгружен при запуске MPI-приложения с использованием механизма *LD_PRELOAD*. Возможен также вариант, когда MPI-приложение статически или динамически связывается (*линкуется*) с данным компонентом.

Для удобства запуска МССТ обеспечен запускающий скрипт, помогающий включить в работу данный компонент. Пример командной строки:

```
$ checkrun -np 4 ./helloworld
```

В ходе выполнения программы на экран или в файл выводятся диагностируемые проблемы. Формат выводимой диагностики:

```
MPI CCT: <тип_диагностики>: <сообщение>.
```

Типом диагностики может быть *ERROR*, *WARNING* или *INFO*.

Примеры диагностик МССТ:

```
MPI CCT: ERROR: MPI_Bcast [rank 0]:
```

```
MPI CCT: ERROR: Two collective calls cause a type mismatch!
```

Анализ найденных ошибок использования MPI поможет программисту устранить их.

Сравнение с программами-конкурентами

В настоящее время наиболее известными являются следующие инструменты проверки корректности использования MPI в процессе выполнения приложения: модуль проверки корректности в

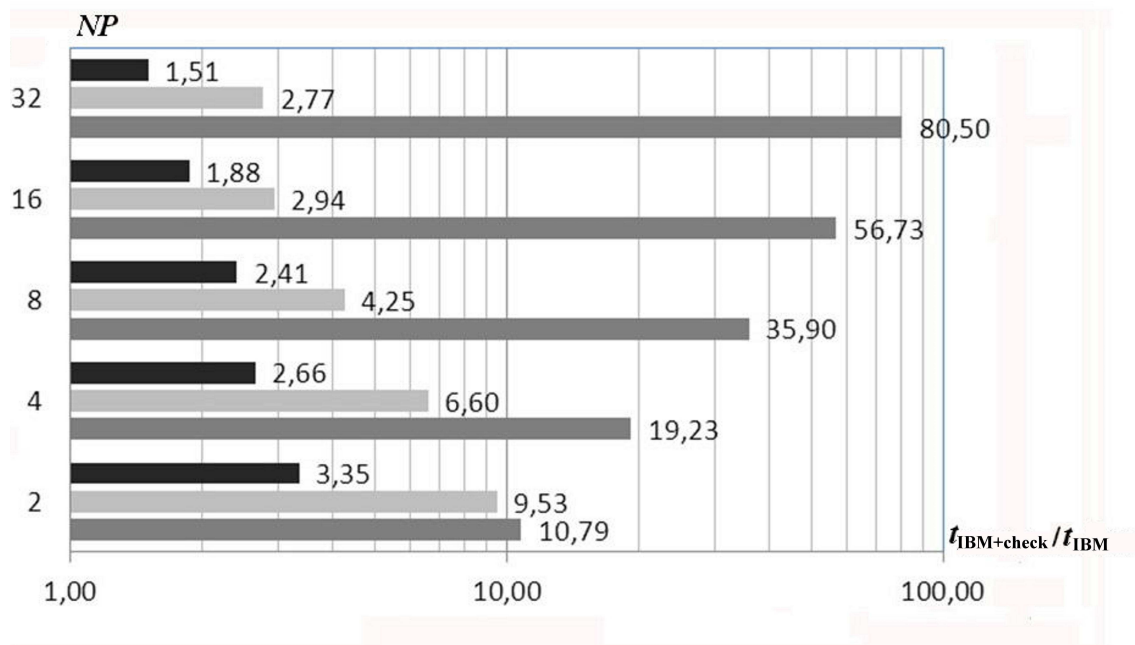


Диаграмма отношения времени выполнения приложения IBM с проверкой корректности использования в нем MPI ко времени выполнения IBM без тестирования для инструментов, отслеживающих стек вызовов: черные столбцы — МССТ; светло-серые — MUST 1.1; темно-серые — Intel TAC 8.0.3 (шкала отношения времени — логарифмическая)

составе Intel Trace Analyzer and Collector (TAC) [4] (см. также главу "Correctness Checking" в [5]), Marmot [6] и MUST [7]. Все эти инструменты используют интерфейс PMPI для самоподключения к выполняемому приложению. Также они имеют общие классы проверок: проверка аргументов функций; слежение за ресурсами MPI, за двухточечными и коллективными операциями; анализ взаимоблокировок. Основные отличия данных инструментов друг от друга приведены в табл. 1, где знак "+" означает наличие указанной характеристики, а "-" указывает на ее отсутствие.

Для оценки производительности MCCT были произведены замеры времени выполнения теста IMB (Intel MPI Benchmarks) [8] на 2, 4, 8, 16 и 32 процессах в сравнении с результатами программ-конкурентов, имеющих возможность отслеживать стек вызовов. Тест IMB во всех случаях запускался с аргументами *-iter 1 -nprmin NP*, где *NP* — количество запускаемых процессов MPI. Использовалось 4 вычислительных сервера, оснащенных процессором Intel Xeon и 48 Гб оперативной памяти. Распределение процессов по узлам X5650 производилось по алгоритму round-robin (распределение по кругу). В качестве коммуникационной среды MPI использовались *разделяемая память* (shared memory) и высокоскоростная коммутируемая последовательная шина (InfiniBand) QDR от Mellanox. Запуск всех инструментов проводился с единственным отличием от настроек по умолчанию: для максимального приближения конфигураций инструментов друг к другу была отключена проверка целостности передачи данных *GLOBAL:*.DATA_TRANSMISSION_CORRUPTED* [7], реализованная только в Intel TAC.

На рисунке показана диаграмма отношения времени выполнения приложения IBM с проверкой корректности использования в нем MPI ко времени выполнения IMB без тестирования. Стоит отметить, что IMB не имеет счетной части и тестирование показывает результат максимальной нагрузки, целиком приходящейся на межпроцессный обмен MPI.

Из приведенной диаграммы видно, что текущая версия MCCT опережает MUST 1.1 (отношение указанных времен меньше) в 2–3 раза, а по сравнению с модулем проверки корректности Intel TAC 8.0.3 она ведет себя быстрее в 3–52 раза, при этом полноценно отслеживая стек вызовов. Такой результат по сравнению с MUST 1.1 обусловлен более аккуратной работой с накапливаемыми данными в стеке вызовов функций при использовании библиотеки Stackwalker [3].

Дополнительное исследование показало, что наблюдаемое огромное отставание модуля проверки корректности Intel TAC объясняется большими накладными расходами проверки перекрытия памяти (*LOCAL:MEMORY:OVERLAP* [7]). Данная проверка также присутствует в MCCT и MUST; она была включена во время тестирования, но такого существенного замедления за собой не повлекла.

Диагностирование ошибок

В данном разделе на примере нескольких тестов показаны возможности MCCT диагностировать ошибки использования MPI, которые, как правило, не диагностируются современными реализациями библиотек MPI. Рассматриваемые примеры были построены и протестированы с помощью тестовой версии библиотеки S-MPI.

Таблица 1

Отличия инструментов проверки корректности использования MPI

Характеристика	MCCT	Intel TAC 8	MUST 1.1	Marmot 2.4
LD_PRELOAD-механизм	+	+	+	—
Отслеживание стека вызовов	+	+	+	—
Проверка целостности передачи данных	планируется	+	—	—
Поддержка MPI-2	частичная	частичная	частичная	полная
Интеграция с трассировщиками	планируется	Intel Trace Collector	—	Vampirtrace
Интеграция с отладчиками	планируется	+	—	+

Взаимоблокировка при выполнении двух операций *MPI_Isend*. В табл. 2 показан двухточечный обмен с использованием *MPI_Isend*. Два процесса входят во взаимоблокировку в ожидании выполнения запроса в *MPI_Wait* перед вызовами *MPI_Recv*. Для сообщений малого размера фактически взаимоблокировка не случается, но инструмент МССТ все равно диагностирует данную проблему — это зависит от реализации библиотеки MPI.

Взаимоблокировка при выполнении операции *MPI_Scatter*. Пример в табл. 3 показывает блокировку приложения вследствие несоответствия коллективной операции: нулевой процесс выполняет *MPI_Finalize*, а первый — *MPI_Scatter*. Для малых размеров рассылаемого сообщения фактического зависания приложения может не случаться (в зависимости от реализации библиотеки MPI).

Наложение памяти. Пример в табл. 4 демонстрирует диагностирование наложения памяти на уже использованное адресное пространство в текущем обмене MPI.

Таблица 2

Взаимоблокировка при выполнении двух операций *MPI_Isend*

Код примера	Результат проверки
<pre>\$ cat deadlock_isend.c #include "mpi.h" int main (int argc, char **argv) { int myrank, remote; char data1[1000000], data2[1000000]; MPI_Request req; MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myrank); if(myrank == 0) remote = 1; else remote = 0; MPI_Isend(&data1, 1000000, MPI_CHAR, remote, 100, MPI_COMM_WORLD, &req); MPI_Wait(&req, MPI_STATUS_IGNORE); MPI_Recv(&data2, 1000000, MPI_CHAR, remote, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE); MPI_Finalize(); return 0; }</pre>	<pre>\$ mpicc -g deadlock_isend.c -o deadlock_isend \$ checkrun -np 2 -nets sm,tcp ./deadlock_isend MPI CCT: ERROR: Global: MPI CCT: ERROR: The application issued a set of MPI calls that can cause a deadlock! MPI CCT: ERROR: A graphical representation of this situation is available in the file named "MCCT_Deadlock.dot". MPI CCT: ERROR: Use the dot tool of the graphviz package to visualize it, e.g. issue "dot -Tjpg MCCT_Deadlock. dot -o deadlock.jpg". MPI CCT: ERROR: The graph shows the nodes that form the root cause of the deadlock, any other active MPI calls have been removed. MPI CCT: ERROR: A legend is available in the dot format in the file named "MCCT_DeadlockLegend.dot". MPI CCT: ERROR: References 1-2 list the involved calls (limited to the first 5 calls, further calls may be involved). MPI CCT: ERROR: The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary). MPI CCT: ERROR: References: MPI CCT: ERROR: 1: MPI_Wait [rank 0] (in main@deadlock_isend.c:20); MPI CCT: ERROR: 2: MPI_Wait [rank 1] (in main@deadlock_isend.c:20); MPI CCT: ERROR: A deadlock has been detected, detailed information is available in the checker output file. MPI CCT: ERROR: You should either investigate details with a debugger or abort, the operation of the checker will stop from now.</pre>

Таблица 3

Взаимоблокировка при выполнении операции *MPI_Scatter*

Код примера	Результат проверки
<pre>\$ cat deadlock_scatter.c ... int main(int argc, char **argv) { MPI_Init(&argc, &argv); int myrank; MPI_Comm_rank(MPI_COMM_WORLD, &myrank); int senddata[1000000], recvdata[1000000]; if(myrank != 0) MPI_Scatter(&senddata, 1000000, MPI_INT, &recvdata, 1000000, MPI_INT, 0, MPI_COMM_WORLD); MPI_Finalize(); return 0; }</pre>	<pre>\$ checkrun -np 2 -nets sm,tcp ./deadlock_scatter MPI CCT: ERROR: Global: MPI CCT: ERROR: The application issued a set of MPI calls that can cause a deadlock! ... MPI CCT: ERROR: References 1-2 list the involved calls (limited to the first 5 calls, further calls may be involved). MPI CCT: ERROR: The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary). MPI CCT: ERROR: References: MPI CCT: ERROR: 1: MPI_Finalize [rank 0] (in main@deadlock_scatter.c:17); MPI CCT: ERROR: 2: MPI_Scatter [rank 1] (in main@deadlock_scatter.c:15); MPI CCT: ERROR: A deadlock has been detected, detailed information is available in the checker output file. MPI CCT: ERROR: You should either investigate details with a debugger or abort, the operation of the checker will stop from now.</pre>

Таблица 4

Наложение памяти

Код примера	Результат проверки
<pre>\$ cat overlap.c ... int main(int argc, char **argv) { int myrank, commsize, data[2]; MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myrank); MPI_Comm_size(MPI_COMM_WORLD, &qcommsize); ... data[0] = myrank + 1; data[1] = myrank + 1; MPI_Gather(data + 1, 1, MPI_INT, data, 1, MPI_INT, 0, MPI_COMM_WORLD); if(myrank == 0) printf("data[0]=%d, data[1]=%d\n", data[0], data[1]); ...</pre>	<pre>\$ mpicc -g overlap.c -o overlap \$ checkrun -np 2 -nets sm,tcp ./overlap MPI CCT: ERROR: MPI_Gather [rank 0]: MPI CCT: ERROR: The memory regions spanned by the recv part overlaps at the 0(th) repetition of datatype at its position (MPI_INT) with regions spanned by the send part of this operation! MPI CCT: ERROR: MPI_Gather [rank 0] called from: MPI CCT: ERROR: #0 main@overlap.c:16 MPI CCT: ERROR: #1 MPI CCT: ERROR: #2 MPI CCT: ERROR: #0 _libc_start_main@/lib64/libc-2.12.so:(0x32a2e1ecdd) MPI CCT: ERROR: #2 _start@/home/aogorodn/work/checker_examples/overlap: (0x4008a9) data[0]=1, data[1]=2 MPI CCT: INFO: Detected 1 errors and 0 warnings. More details is available in the checker output file.</pre>

Несоответствие типов пересылаемых данных. В данном примере (табл. 5) для широко-вещательной рассылки используется коллективная операция *MPI_Bcast*, которая рассылает данные зарегистрированного типа *mpi_Struct1*, представляющего собой структуру $\{MPI_DOUBLE, MPI_INT\}$. Принимающие процессы ожидают данные зарегистрированного типа *mpi_Struct2* — структуру $\{MPI_INT, MPI_DOUBLE\}$.

Потеря двухточечного сообщения. Табл. 6 содержит пример отслеживания потерянных (непереданных) сообщений двухточечного обмена. В данном случае регистрируется потеря сообщения по запросу от вызова *MPI_Send_init*.

В качестве резюме к приведенным выше примерам можно отметить, что все демонстрируемые проблемы успешно диагностируются данной утилитой. При этом предоставляется максимально полная информация для разработчика о местонахождении проблемных мест в исходном коде благодаря реализованному отслеживанию стека вызовов.

Пример использования МССТ на реальном приложении

В качестве предмета проверки были использованы два программных продукта физико-математического моделирования — ЛЭГАК-ДК 4.0 и ЛОГОС 4.0 [9], разработанные в РФЯЦ-ВНИИЭФ.

Успешный запуск МССТ для проверки корректности использования MPI при расчете тестовых заданий *fbo_22_14_newdk.k* и *nasa.yaml* выявил следующие проблемы:

1. Не освобождаются коммутаторы, группы MPI и типы данных MPI перед вызовом *MPI_Finalize*.
2. Функции *MPI_Alltoallv*, *MPI_Isend*, *MPI_Recv* вызываются с нулевыми значениями аргумента *count*.
3. Функции двухточечного обмена вызываются со значением аргумента *tag* больше чем 32 767, что не поддерживается некоторыми реализациями MPI.

Можно отметить, что обнаруженные проблемы не являются критическими в большинстве реализаций библиотеки MPI, но потенциально могут приводить к ошибочному выполнению приложения.

Заключение

В работе анонсирован компонент проверки корректности приложений MPI, входящий в состав разрабатываемого программного комплекса S-MPI, позволяющий сократить время на отладку и оптимизацию сложных параллельных комплексов.

На ряде примеров продемонстрирована способность МССТ успешно диагностировать серьезные ошибки в параллельном коде приложений MPI и выдавать максимально полную информацию по обнаруженной проблеме, включая стек вызовов и расположение в исходном коде.

Данный инструмент прошел успешную проверку на примере двух программных продуктов разработки РФЯЦ-ВНИИЭФ — ЛЭГАК-ДК и ЛОГОС. Инструмент позволил обнаружить ряд проблем, не являющихся критическими для большинства реализаций библиотеки MPI.

Замеры производительности МССТ показали, что продукт достаточно хорошо проявляет себя по сравнению с программами-конкурентами.

В целях развития МССТ планируются:

1. Наладка регулярного тестирования компонента с целью отслеживания и устранения критических ошибок в исходном коде МССТ.
2. Добавление проверки целостности передаваемых данных между MPI-процессами.
3. Реализация возможности настраивать конфигурацию проверок через переменные окружения.
4. Расширение поддержки стандарта MPI-2.
5. Интеграция МССТ с компонентом профилирования и трассировки приложения MPI в составе программного комплекса S-MPI.
6. Поддержка работы в паре с отладчиком, таким как GDB, Alinea DDT, TotalView.

Таблица 5

Несоответствие типов пересылаемых данных

Код примера	Результат проверки
<pre> \$ cat mismatch.c ... typedef struct { double a; int b; } Struct1; typedef struct { int a; double b; } Struct2; int main (int argc, char **argv) { MPI_Datatype mpi_Struct1, mpi_Struct2; int myrank, blocklengths[] = { 1, 1 }; MPI_Aint displs1[] = { 0, 8 }, displs2[] = { 0, 4 }; MPI_Datatype types1[] = { MPI_DOUBLE, MPI_INT }, types2[] = { MPI_INT, MPI_DOUBLE }; ... MPI_Type_struct(2, blocklengths, displs1, types1, &mpi_Struct1); MPI_Type_commit(&mpi_Struct1); MPI_Type_struct(2, blocklengths, displs2, types2, &mpi_Struct2); MPI_Type_commit(&mpi_Struct2); if(myrank == 0) { Struct1 data; data.a = 1; data.b = 2; MPI_Bcast(&data, 1, mpi_Struct1, 0, MPI_COMM_WORLD); } else { Struct2 data; MPI_Bcast(&data, 1, mpi_Struct2, 0, MPI_COMM_WORLD); } ... </pre>	<pre> \$ mpicc -g mismatch.c -o mismatch \$ checkrun -np 2 -nets sm,tcp ./mismatch [0] Broadcasting data: data.a=1.000000, data.b=2 [1] Bcast receive success: data.a=0, data.b=0.000000 MPI CCT: ERROR: MPI_Bcast [rank 0]: MPI CCT: ERROR: Two collective calls cause a type mismatch! MPI CCT: ERROR: This call sends data to the call in reference 1. MPI CCT: ERROR: The mismatch occurs at send type and at (STRUCT)[0][0](MPI_INT) in the receive type. MPI CCT: ERROR: (Information on communicator: MPI_COMM_WORLD) Datatype created at reference 2 is for C, committed at (Information on send transfer of count 1 with type: reference 3, based on the following type(s): { MPI_DOUBLE, MPI_INT}) (Information on receive of count 1 with committed at reference 5, based on the following type(s): { MPI_INT, MPI_DOUBLE}) MPI CCT: ERROR: References: MPI CCT: ERROR: 1: MPI_Bcast [rank 1] (in main@mismatch.c:28); MPI CCT: ERROR: 2: MPI_Type_struct [rank 0] (in main@mismatch.c:16); MPI CCT: ERROR: 3: MPI_Type_commit [rank 0] (in main@mismatch.c:17); MPI CCT: ERROR: 4: MPI_Type_struct [rank 1] (in main@mismatch.c:18); MPI CCT: ERROR: 5: MPI_Type_commit [rank 1] (in main@mismatch.c:19); MPI CCT: ERROR: MPI_Bcast [rank 0] called from: MPI CCT: ERROR: #0 main@mismatch.c:25 MPI CCT: ERROR: #1 __libc_start_main@/lib64/ libc-2.12.so:(0x32a2e1ecdd) MPI CCT: ERROR: #2 _start@/home/.../ checker_examples/mismatch:(0x400949) MPI CCT: INFO: Detected 1 errors and 0 warnings. More details is available in the hecker output file. </pre>

Потеря двухточечного сообщения

Код примера	Результат проверки
<code>\$ cat lost.c</code>	<code>\$ mpicc -g lost.c -o lost</code>
<code>...</code>	<code>\$ checkrun -np 2 -nets sm,tcp ./lost</code>
<code>int main(int argc, char **argv) {</code>	<code>MPI CCT: ERROR: MPI_Start [rank 0]:</code>
<code> int myrank, data, data1 = 10, data2 = 20;</code>	<code>MPI CCT: ERROR: Lost send of rank 0 to rank 1 (both</code>
<code> MPI_Request req;</code>	<code>as ranks in MPI_COMM_WORLD) tag is 1!</code>
<code>... if(myrank == 0) {</code>	<code>MPI CCT: ERROR: (Information on communicator:</code>
<code> MPI_Send_init(&data1, 1, MPI_INT, 1, 1,</code>	<code>MPI_COMM_WORLD)</code>
<code>MPI_COMM_WORLD, &req);</code>	<code>MPI CCT: ERROR: MPI_Start [rank 0] called from:</code>
<code> MPI_Start(&req);</code>	<code>MPI CCT: ERROR: #0 main@lost.c:15</code>
<code> MPI_Request_free(&req);</code>	<code>MPI CCT: ERROR: #1</code>
<code> MPI_Send(&data2, 1, MPI_INT, 1, 2,</code>	<code>__libc_start_main@/lib64/libc-2.12.so:(0x32a2e1ecdd)</code>
<code>MPI_COMM_WORLD);</code>	<code>MPI CCT: ERROR: #2</code>
<code> } else if(myrank == 1)</code>	<code>_start@/home/aogorodn/work/checker_examples/lost:</code>
<code> MPI_Recv(&data, 1, MPI_INT, 0, 2,</code>	<code>(0x4008f9)</code>
<code>MPI_COMM_WORLD, MPI_STATUS_IGNORE);</code>	
<code>...</code>	<code>MPI CCT: INFO: Detected 1 errors and 0 warnings.</code>
	<code>More details is available in the checker output file.</code>

Работа выполнена в рамках контракта (№ 07.524.12.4020) с Министерством образования и науки РФ.

Список литературы

1. *Message Passing Interface Forum*. MPI: A Message Passing Interface // Proc. of "Supercomputing '93". Los Alamitos: IEEE Computer Society Press, 1993. P. 878—883.
2. Graphviz. <http://www.graphviz.org>.
3. Dyninst, Stackwalker. <http://www.dyninst.org/downloads/stackwalkerapi>.
4. Intel Trace Analyzer and Collector. <http://software.intel.com/en-us/articles/intel-trace-analyzer/>.
5. Intel@ Trace Collector. Reference Guide. http://software.intel.com/sites/products/documentation/hpc/itac/itc_reference_guide.pdf.
6. Marmot. <http://www.hlrs.de/organization/av/amt/research/marmot/>.
7. MUST — MPI Runtime Error Detection Tool. http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/must/index_html/document_view?set_language=de.
8. IMB. <http://software.intel.com/en-us/articles/intel-mpi-benchmarks/>.
9. ЛЭГАК-ДК и ЛЮГОС. <http://www.vniief.ru/wps/wcm/connect/vniief/site/researchdirections/Research/theoryandmodel/metodikiiprogrammi>.

Статья поступила в редакцию 29.08.12.