УДК 004.051 **DOI: 10.14529/cmse190406**

ПРОГРАММНАЯ ПОДДЕРЖКА МОДЕЛИ BSF

© 2019 Н.А. Ежова, Л.Б. Соколинский

Южно-Уральский государственный университет
(454080 Челябинск, пр. им. В.И. Ленина, д. 76)
E-mail: EzhovaNA@susu.ru, Leonid.Sokolinsky@susu.ru

Поступила в редакцию: 08.11.2019

В статье описана программная поддержка модели параллельных вычислений BSF (Bulk Synchronous Farm), ориентированной на итерационные алгоритмы с высокой вычислительной сложностью, разрабатываемые для многопроцессорных систем с распределенной памятью экзафлопсного уровня производительности. Программная поддержка включает в себя параллельный BSF-каркас и веб-приложение BSF-Studio. Приведены определение и классификация параллельных программных каркасов. Описан новый BSF-каркас, разработанный согласно BSF-модели: его файловая структура и логика работы. BSF-каркас представляет собой совокупность файлов исходного кода на языке C++, используемых для быстрого создания BSF-программ. Приведено подробное описание проектирования и реализации веб-приложения BSF-Studio, которое представляет собой визуальный конструктор BSF-программ. BSF-Studio обеспечивает поэтапное заполнение проблемно-зависимых частей BSF-каркаса, а также компиляцию и запуск программного кода.

Ключевые слова: BSF, модель параллельных вычислений, параллельный каркас, BSF-Studio, веб-приложение.

ОБРАЗЕЦ ЦИТИРОВАНИЯ

Ежова Н.А., Соколинский Л.Б. Программная поддержка модели BSF // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 4. С. 84–99. DOI: 10.14529/cmse190406.

Введение

Полноценная модель параллельных вычислений [1] должна включать в себя спецификационный компонент определяющий, что есть корректная программа в контексте данной модели. Параллельная программа может выдавать правильный результат, однако быть некорректной по отношению к требованиям модели параллельных вычислений. Такого рода некорректность приводит к тому, что применение стоимостных метрик используемой модели будет выдавать неверные оценки. Для предотвращения таких ситуаций и для ускорения создания программных кодов на практике часто используют параллельные каркасы, разрабатываемые на основе требований соответствующей модели параллельных вычислений.

Параллельный каркас (*parallel skeleton*) в общем виде представляет собой программную конструкцию (или библиотеку функций), которая инкапсулирует некоторый шаблон параллельных вычислений и межпроцессорных коммуникаций [2]. В идеальном случае параллельный каркас представляет собой компилируемый, но не исполняемый код. Чтобы использовать параллельный каркас, программист должен добавить проблемно-зависимые типы и структуры данных, а также реализовать проблемно-зависимые функции. При этом дополнения программного кода могут делаться инкрементально с сохранением свойства компилируемости. Каркас берет на себя функцию объединения проблемно-зависимых операций с программным кодом, обеспечивающим их параллельное выполнение и коммуникации. Таким образом, абстрагирование от аспектов, связанных с параллелизмом, может значительно упростить и систематизировать разработку параллельных программ и помочь в эффективном использовании стоимостных метрик, преобразовании и оптимизации

исходного кода. В силу высокого уровня абстракции параллельные каркасы имеют концептуальную общность с функциями высшего порядка, применяемыми в функциональном программировании, а также с шаблонами объектно-ориентированного программирования, и многие конкретные каркасы используют эти механизмы.

В традиционных инструментах параллельного программирования используется подход, в соответствии с которым программист должен реализовывать низкоуровневое управление взаимодействием между параллельными процессами, используя механизмы, соответствующие используемой модели. Например, ядро MPI основано на обмене данными между процессами типа точка-точка с различными режимами синхронизации. Подобно этому, библиотеки потоковой обработки основаны на примитивах блокировки, условной синхронизации, атомарности и так далее. Этот подход является очень гибким, позволяя организовать взаимодействия любой сложности. Однако на практике многие параллельные алгоритмы основываются на хорошо понятных и хорошо известных шаблонах. Например, алгоритмы обработки изображений часто используют конвейерный параллелизм. Аналогичным образом, алгоритм для подбора параметра требует планирования параллельных процессов, которое не зависит от параметризованной задачи и диапазона значений исследуемого параметра.

Термин *параллельный каркас (algorithmic skeleton)* был введен Мюрреем Коулом (Murray Cole) в работе [3]. Появление этого термина явилось результатом осознания того факта, что многие параллельные приложения используют одни и те же внутренние коммуникационные шаблоны. Каркасный подход предполагает, что подобные шаблоны должны быть обобщены в виде программной конструкции или библиотеки функций, которые отвечают за реализацию скрытого управляющего «каркаса», оставляя программисту реализацию проблемно-зависимых функций, обеспечивающих решение конкретной задачи. Например, *конвейерный каркас* потребует от программиста реализацию операций, выполняемых на каждом шаге конвейера, в то время как каркас будет отвечать за распределение конвейерных операций между различными процессорами, межпроцессорные коммуникации, непрерывность работы конвейера и так далее. Аналогичным образом, в *каркасе для подбора параметра* программист должен будет предоставить код для одиночного прогона параметризованной задачи и требуемый диапазон значений параметра. Каркас будет определять (и, возможно, динамически корректировать) количество используемых рабочих, гранулярность распределенных вычислительных заданий, коммуникационные механизмы и отказоустойчивость. На более высоком уровне абстракции использование *каркаса «разделяй-и-властвуй»* может потребовать от программиста описания способов разделения задачи на независимые подзадачи и последующего объединения результатов их работы, решения вопроса, является ли исходная задача делимой соответствующим образом, и реализации алгоритмов решения неделимых подзадач. Каркас в этом случае будет отвечать за все остальное, начиная с вопроса, стоит ли вообще использовать параллелизм, и кончая такими деталями, как динамическая диспетчеризация, гранулярность и коммуникации.

Подобный высокоуровневый подход меняет процесс разработки программы в нескольких отношениях. Во-первых, он освобождает пользователя от нетривиальной задачи принятия правильных проектных решений на основе многочисленных, взаимосвязанных низкоуровневых особенностей конкретного приложения и конкретной вычислительной плат-

формы. Во-вторых, использование отлаженного и апробированного параллельного каркаса существенно уменьшает вероятность появления ошибок в результирующем коде, что особенно важно для сложных задач, решаемых на вычислительных системах с массовым параллелизмом, где традиционная отладка является слишком сложной. В-третьих, использование параллельного каркаса дает гарантию параллельной эффективности вместо апостериорной оценки производительности, в результате которой, возможно, придется отказаться от трудоемкой параллельной программы по причине ее фатальной неэффективности. В-четвертых, каркасы предоставляют семантически обоснованные методы для сборки и оптимизации программного кода, открывающие новые перспективы в разработке программного обеспечения (в том числе, в повторном использовании программных кодов). И последнее, но не менее важное: повышение уровня абстракции, заключающееся в переходя от частного к общему, дает новое понимание фундаментальных принципов параллельного программирования [2].

В зависимости от функциональности параллельные каркасы могут быть разделены на следующие три типа [4]:

– *каркасы с параллелизмом данных* (*data-parallel*), применяемые для выполнения одинаковых операций над элементами больших однородных структур данных;
– *каркасы с параллелизмом задач* (*task-parallel*), оперирующие задачами и обеспечивающие их синхронизацию в соответствии со связями по данным;
– *резолюционные* (*resolution*) *каркасы*, определяющие общий метод распараллеливания для семейства схожих задач.

Таксономия базовых параллельных каркасов приведена в табл. 1.

– *Map* является наиболее важным базовым каркасом с параллелизмом данных. Его природа тесно связана с функциональными языками программирования. Семантика каркаса *map* заключается в том, что некоторая функция может быть одновременно применена ко всем элементам списка в целях распараллеливания вычислений. Параллелизм данных организуется следующим образом: список разбивается на подсписки равной длины по числу процессорных узлов, и каждый процессорный узел обрабатывает свой подсписок параллельно с другими узлами. Обменов данными между процессорными уздами при этом не происходит. По завершению обработки полученные результаты объединяются в общий результат. Таким образом, каркас *map* соответствует схеме параллельной обработки SIMD (single instruction, multiple data) [5].

Таблица 1

Таксономия базовых параллельных каркасов

Тип каркаса	**Область применения**	**Примеры элементарных каркасов**
Параллелизм данных	Структуры данных	map, fork, reduce
Параллелизм задач	Задачи	farm, pipe

Fork работает подобно *map*. Разница в том, что вместо применения одной и той же функции ко всем элементам списка, к каждому элементу применяется своя функция. Таким образом, каркас *fork* соответствует схеме параллельной обработки MIMD (multiple instruction, multiple data) [5].

Reduce используется для вычисления инфиксных операций над списком путем деления элементов списка на пары и применения к каждой паре указанной инфиксной операции. Полученные результаты образуют список в два раза меньшей длины, к которому снова применяется *reduce*, и так да тех пор, пока не будет получено итоговое интегральное значение. Таким образом, в отличие от *map*, каркас *reduce* требует обменов данными между процессорными узлами для агрегирования частичных результатов.

Farm реализует параллельное выполнение задач по схеме мастер-рабочие. Этот каркас также называют *bag of tasks* (пакет задач). *Farm* обеспечивает параллельное выполнение независимых подзадач на узлах-рабочих и слияние полученных ими результатов в общий результат на узле-мастере, который финализирует решение задачи.

Pipe реализует конвейерный параллелизм. Этот каркас целесообразно использовать, когда одна и та же задача должна решаться для многих наборов входных данных. В этом случае решение задачи разбивается на последовательные стадии, количество которых называется длиной конвейера. Каждая стадия выполняется отдельным процессорным узлом. Параллелизм достигается за счет того, что одновременно с выполнением i-той стадии для набора данных $x^{(j)}$ выполняется стадия $(i-1)$ для набора $x^{(j+1)}$. Каркас *pipe* обеспечивает синхронизацию выполнения стадий и передачу промежуточных результатов от одной стадии другой. Заметим, что в общем случае достаточно иметь каркас *pipe* с фиксированной длиной, так как каркасы могут вкладываться друг в друга.

Статья имеет следующую структуру. В разделе 1 описывается структура параллельного BSF-каркаса на языке C++, используемого для быстрого создания BSF-программ. В разделе 2 описывается визуальный конструктор BSF-программ.

1. Параллельный каркас для модели BSF

Модель параллельных вычислений BSF (Bulk Synchronous Farm) была предложена в работах [6, 7]. Модель BSF является расширением модели BSP и ориентирована на итерационные алгоритмы с высокой вычислительной сложностью, разрабатываемые для многопроцессорных систем с распределенной памятью экзафлопсного уровня производительности. Отличительной особенностью модели BSF от других известных моделей параллельных вычислений является возможность оценки границы масштабируемости алгоритма на ранних этапах его разработки.

В данном разделе описывается параллельный BSF-каркас на языке C++, предназначенный для быстрого создания параллельных BSF-программ для кластерных вычислительных систем. Для организации обменов сообщениями между процессорными узлами BSF-каркас использует программный интерфейс MPI [8–10]. BSF-каркас спроектирован таким образом, что он допускает поэтапное заполнение проблемно-зависимых частей, и при этом компилируется на всех этапах разработки. Исходные тексты BSF-каркаса свободно доступны в сети Интернет по адресу https://github.com/nadezhda-ezhova/BSF-MR.

1.1. Файловая структура каркаса

BSF-каркас состоит из двух групп файлов:

1) файлы с префиксом «BSF» содержат проблемно-независимый код и не подлежат изменениям со стороны пользователя;
2) файлы с префиксом «Problem» предназначены для заполнения пользователем проблемно-зависимых частей программы.

Описания всех файлов исходного кода приведены в табл. 2. Граф зависимостей файлов BSF-каркаса по включению с помощью директивы #include приведен на рис. 1. Серым закрашены файлы, в которых пользовательские изменения не допускаются; узорной штриховкой обозначены файлы с предопределенными заголовками определений, тело которых должен заполнить пользователь; белое заполнение соответствует файлам, полностью заполняемым пользователем. Порядок заполнения BSF-каркаса приведен в файле ReadMe, доступном по адресу https://github.com/nadezhda-ezhova/BSF-MR.

Таблица 2

Файлы исходного кода BSF-каркаса

Файл	**Описание**
Проблемно-независимый код	
BSF-Code.cpp	Реализации головной функции main и всех проблемно-независимых функций
BSF-Data.h	Проблемно-независимые переменные и структуры данных
BSF-Forwards.h	Предописания проблемно-независимых функций
BSF-Include.h	Включение проблемно-независимых библиотек (iostream, mpi.h, omp.h)
BSF-ProblemFunctions.h	Предописания предопределенных функций с проблемно-зависимой реализацией
BSF-Types.h	Определения проблемно-независимых типов
Проблемно-зависимый код	
Problem-bsfCode.cpp	Реализация предопределенных проблемно-зависимых функций
Problem-bsfParameters.h	Предопределенные проблемно-зависимые параметры
Problem-bsfTypes.h	Предопределенные проблемно-зависимые типы
Problem-Data.h	Проблемно-зависимые переменные и структуры данных
Problem-Forwards.h	Предописания проблемно-зависимых функций
Problem-Include.h	Включение проблемно-зависимых библиотек
Problem-Parameters.h	Проблемно-зависимые параметры
Problem-Types.h	Проблемно-зависимые типы

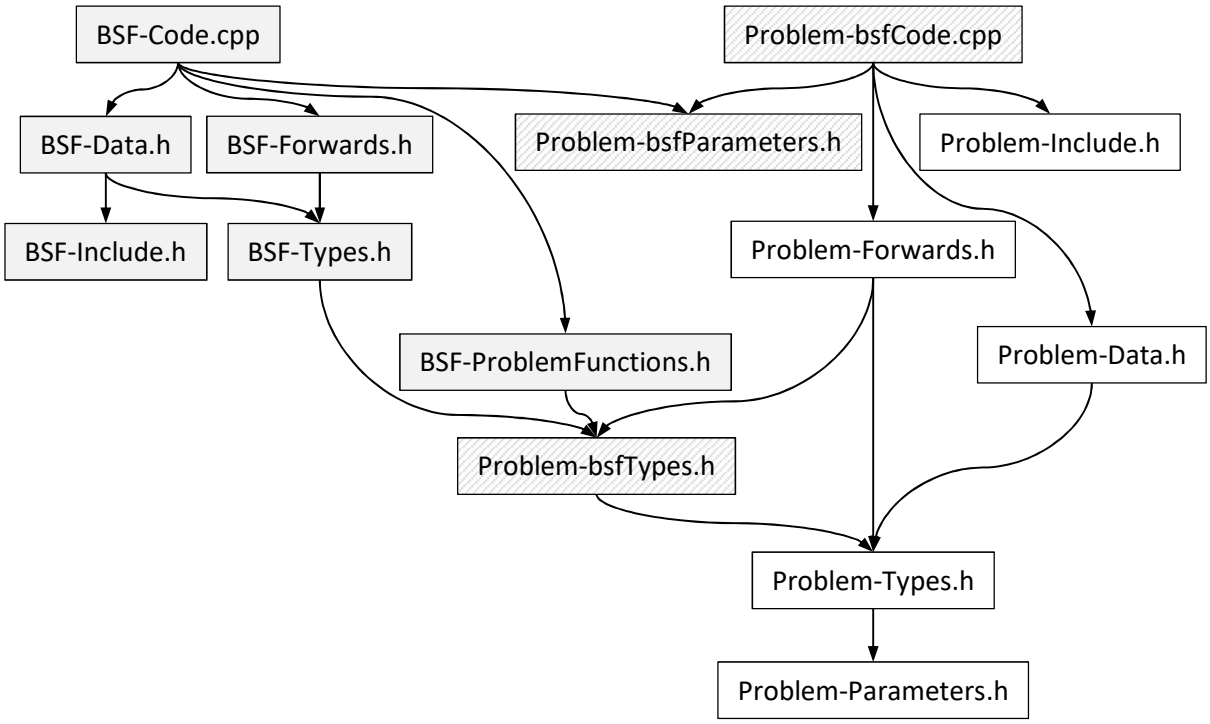

Рис. 1. Граф зависимостей файлов BSF-каркаса по включению #include

Файл ***Problem-bsfParameters.h*** содержит следующие предопределенные константы (#define), используемые в BSF-Code.cpp, значения которых устанавливает пользователь:

- PP_BSF_PRECISION: задает количество значащих цифр (ширину поля вывода) для чисел с плавающей точкой;
- PP_BSF_ITER_OUTPUT: если указанный #define определен, то будет выполняться вывод результатов итераций с определенным шагом, задаваемым константой PP_BSF_TRACE_COUNT;
- PP_BSF_TRACE_COUNT: задает частоту вывода результатов итераций, например, если указано значение 10, то будет выводиться результат каждой десятой итерации (если определен #define PP_BSF_ITER_OUTPUT);
- PP_BSF_OMP: если указанный #define определен, то будет включена #pragma omp parallel for перед циклом, реализующим функцию Map;

PP_BSF_NUM_THREADS: указывает количество потоков управления, в директиве #pragma omp parallel (если константа *PP_BSF_NUM_THREADS* не определена, то запускается количество потоков, определенное по умолчанию).

Файл ***Problem-bsfTypes.h*** включает в себя заголовки описаний (описания с пустым телом) трех предопределенных структурных типов:

- *PT*_bsf_data_T: описывает структуру данных, передаваемых каждому рабочему для выполнения очередной итерации (может содержать текущее приближение и другие параметры);
- PT_bsf_mapElem_T: описывает структуру элемента списка «Map»;
- PT_*bsf_reduceElem_T*: описывает структуру элемента списка «Reduce».

Файл ***BSF-ProblemFunctions.h*** включает в себя предописания (форварды) следующих предопределенных пользовательских функций, которые должны быть реализованы в файле *Problem-bsfCode.cpp*:

- PC_bsf_AssignListSize(int* listSize): присваивает переменной *listSize длину списка «Map» (совпадает с длиной списка «Reduce»);
- PC_bsf_CopyData(PT_bsf_data_T* dataIn, PT_bsf_data_T* dataOut): копирует данные из структуры *dataOut в структуру *dataIn;
- PC_bsf_IterOutput(PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T data, int iterCount, double elapsedTime): выводит результаты итерации, используя в качестве параметров reduceResult — результат выполнения функции Reduce над всем списком; count — количество элементов, участвовавших в Reduce; data — задание, выполненное на данной итерации; iterCount — количество выполненных итераций; elapsedTime — общее время, затраченное на решение задачи.
- PC_bsf_Init(bool* success): выделяет память для проблемно-зависимых структур данных и выполняет их инициализацию; если необходимую память выделить не удалось, переменной *success должно быть присвоено значение false;
- PC_bsf_MapF(PT_bsf_mapElem_T* mapElem, PT_bsf_reduceElem_T* reduceElem, PT_bsf_data_T* data, int* success): вычисляет функцию F, являющуюся параметром оператора Map, используя аргументы *mapElem — элемент списка «Map», над которым выполняется функция F; *reduceElem — элемент списка «Reduce», которому должно быть присвоено значение функции F; *data — задание, содержащее текущее приближение; параметру *success должно быть присвоено значение 0, если полученный элемент списка «Reduce» должен игнорироваться при выполнении операции Reduce.
- PC_bsf_ParametersOutput(int numOfWorkers, PT_bsf_data_T data): выводит начальные параметры задачи, используя аргументы numOfWorkers — количество процессов-рабочих и data — начальное задание, содержащее начальное приближение;
- PC_bsf_ProblemOutput(PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T data, int iterCount, double t): выводит конечные результаты работы программы, используя аргументы *reduceResult — результат выполнения оператора Reduce, count количество элементов «суммированных» при выполнении оператора Reduce, data — последнее задание (последнее приближение), t — общее время работы программы;
- PC_bsf_ProcessResults(bool* exit, PT_bsf_reduceElem_T* reduceResult, int count, PT_bsf_data_T* data): обрабатывает результаты, полученные в результате выполнения очередной итерации, используя аргументы *reduceResult — результат выполнения оператора Reduce, count количество элементов «суммированных» при выполнении оператора Reduce, data — последнее задание (последнее приближение); если вычисления необходимо продолжить, переменной *exit присваивается значение true, в противном случае — false;
- PC_bsf_ReduceF(PT_bsf_reduceElem_T* x, PT_bsf_reduceElem_T* y, PT_bsf_reduceElem_T* z): реализует функцию, являющуюся аргументом оператора Reduce, по формуле $z := x \text{ E } y$.
- PC_bsf_SetInitApproximation(PT_bsf_data_T* data): записывает в *data начальное задание (начальное приближение);
- PC_bsf_SetMapSubList(PT_bsf_mapElem_T* subList, int count, int offset, bool* success): заполняет подсписок списка «Map», используя аргументы *subList — указа-

тель на первый элемент подсписка, count — длина подсписка, offset — сдвиг, относительно начала списка; если у элемента списка «Map» имеются поля типа указатель, то необходимо выделить память под вектор, матрицу или другую структуру; если обнаружена нехватка памяти, переменной **success* необходимо присвоить значение *false*.

Файл ***BSF-Types.h*** содержит определения двух проблемно-независимых структурных типов: *BT_order_T* и *BT_extendedReduceElem_T*. Структурный тип *BT_order_T* определяет структуру задания, пересылаемого мастером рабочим, и включает в себя два поля: *data* типа *PT_bsf_data_T*, определяемого в файле *Problem-bsfTypes.h* (см. стр. 89), и *exit* типа *char*. Если поле *exit* содержит двоичное значение 1, то процессы-рабочие должны завершить свою работу. Структурный тип *BT_extendedReduceElem_T* определяет структуру элемента расширенного списка «*Reduce*» и включает в себя два поля: *elem* типа *PT_bsf_reduceElem_T*, определяемого в файле *Problem-bsfTypes.h* (см. выше), и *counter* типа *int*. До выполнения оператора *Reduce* в поле *counter* может находиться значение 1, либо 0. Значение 0 означает, что данный элемент должен быть проигнорирован при выполнении оператора *Reduce*. Значение 1 означает, что данный элемент должен быть учтен при выполнении оператора *Reduce*. После выполнения оператора *Reduce* над списком (подсписком) в поле *counter* результирующего элемента заносится количество учтенных исходных элементов.

Файл ***BSF-Data.h*** содержит определения проблемно-независимых переменных и структур данных. В число переменных входят следующие:

- *BD_*listSize: длина списка «Map» (совпадает с длиной списка «Reduce»);
- BD_size: количество MPI-процессов;
- BD_rank: номер текущего MPI-процесса;
- BD_masterRank: номер процесса-мастера (равен $BD\_size - 1$);
- BD_numOfWorkers: количество процессов-рабочих (равно $BD\_size - 1$);
- BD_elemsPerWorker: количество элементов списка, приходящихся на одного рабочего (равно $\lfloor BD\_listSize \,/\, BD\_numOfWorkers \rfloor$;
- BD_tailLength: длина остатка списка после деления на число рабочих (равно $BD\_listSize - BD\_elemsPerWorker \cdot BD\_numOfWorkers$);
- BD_exit: индикатор завершения вычислений;
- BD_success: индикатор успешного выполнения инициализации;
- BD_t: время, затраченное на вычисления (не учитываются ввод/вывод данных и инициализация);
- BD_iterCount: количество выполненных итераций.

В число структур данных входят следующие:

- *BD_data*: структура, в которой формируется данные, необходимые рабочим для выполнения очередной итерации;
- BD_mapSubList: указатель на подсписок, обрабатываемый рабочим (используется только процессами-рабочими);
- BD_extendedReduceList: указатель на расширенный список «Reduce»;
- BD_extendedReduceResult_P: указатель на структуру, в которой вычисляется результирующий элемент при выполнении оператора Reduce над расширенным списком «Reduce»;
- BD_order: указатель на структуру, в которой формируется задание для рабочего;

- BD_status: указатель на массив системных MPI-структур «MPI_Status», содержащих параметры сообщений для каждого рабочего;
- BD_request: указатель на массив системных MPI-переменных, содержащий идентификаторы асинхронных обменов по числу процессов-рабочих (используется только процессом-мастером);
- BD_subListSize: указатель на массив целых чисел, содержащий размеры подсписков для всех рабочих;
- BD_offset: указатель на массив целых чисел, содержащий для каждого рабочего сдвиг его подсписка относительно начала общего списка.

Файл ***BSF-Code.cpp*** включает в себя реализацию головной функции main и реализации следующих проблемно-независимых функций:

- BC_MpiRun выполняет инициализацию MPI и соответствующих переменных;
- BC_Init выделяет необходимую память для структур данных, инициализирует переменные и заполняет список «Map» исходными данными;
- BC_Master реализует процесс, выполняемый на узле-мастере;
- BC_Worker реализует процесс, выполняемый на узлах-рабочих;
- BC_MasterMap реализует шаг «Map» на узле-мастере;
- BC_MasterReduce реализует шаг «Reduce» на узле-мастере;
- BC_MpiRun осуществляет инициализацию MPI;
- BC_WorkerMap реализует шаг «Map» на узлах-рабочих;
- BC_WorkerReduce реализует шаг «Reduce» на узлах-рабочих;
- BC_ProcessExtendedReduceList обрабатывает указанную часть списка «Reduce».

Файл ***BSF-Include.h*** подключает с помощью директивы #include необходимые системные библиотеки. Файл ***BSF-Forwards.h*** содержит предописания функций, реализуемых в файле *BSF-Code.cpp.* Файл ***Problem-bsfCode.cpp*** содержит реализации проблемно-зависимых функций, предописания которых находятся в файле *BSF-ProblemFunctions.h* (см. стр. 89). Файл ***Problem-Include.h*** подключает с помощью директивы #include системные библиотеки, необходимые для реализаций проблемно-зависимых функций. Файл ***Problem-Forwards.h*** содержит предописания пользовательских функций, используемых для реализации предопределенных проблемно-зависимых функций. Файл ***Problem-Types.h*** содержит определения пользовательских типов данных. Файл ***BSF-Types Problem-Data.h*** содержит определения пользовательских глобальных переменных и структур данных. Файл ***Problem-Parameters.h*** содержит определения проблемно-зависимых констант.

1.2. Логика работы каркаса

Кратко опишем общую логику работы каркаса. ***Головная функция main*** вызывает функцию *BC_MpiRun*, которая выполняет инициализацию MPI и определяет значения переменных:

- *BD_size*: количество запущенных MPI-процессов (не может быть меньше двух);
- BD_rank: номер собственного MPI-процесса.

Затем вызывается функция *BC_Init*, которая выделяет необходимую память и определяет значения проблемно-независимых глобальных переменных и структур данных, определенных в файле *BSF-Data.h* (см. стр. 91). Если хотя бы в одном MPI-процессе не

удалось выделить необходимую память, процесс-мастер выводит в глобальный поток *cout* диагностическое сообщение «*Error: PC_bsf_Init failed (not enough memory)!*», и все MPI-процессы заканчивают свою работу. Если выделение памяти и инициализация прошли успешно, то процесс с номером *BD_masterRank* выполняет функцию *BC_Master*, а остальные процессы выполняют функцию *BC_Worker*.

Функция BC_Master реализует алгоритм работы мастера. Сначала выводятся параметры задачи с помощью функции *PC_bsf_ParametersOutput*. Затем организуется основной итерационный цикл. В ходе каждой итерации выполняются следующие действия:

1) вызывается функция *BC_MasterMap*, пересылающая всем рабочим в асинхронном режиме (с помощью функции *MPI_Isend*) задание для очередной итерации;
2) вызывается функция *BC_MasterReduce*, получающая от всех рабочих в асинхронном режиме (с помощью функции *MPI_Irecv*) результат выполнения оператора Reduce на подсписках;
3) вызывается предопределенная проблемно-зависимая функция *PC_bsf_ProcessResults*, проверяющая критерий завершения, вычисляющая очередное приближение и формирующая задание для следующей итерации;
4) если определен *#define PP_BSF_ITER_OUTPUT*, выводятся результаты итерации.

Основной итерационный цикл завершается, когда в переменную *BD_exit* функция *PC_bsf_ProcessResults* поместит значение *false*. В этом случае выполняется функция *BC_MasterMap* с параметром *false*, приводящая к завершению процессов-рабочих. После этого происходит вывод результатов с помощью предопределенной проблемно-зависимой функции *PC_bsf_ProblemOutput*, после чего функция *BC_Master* завершает свою работу.

Функция BC_Worker, реализует алгоритм работы рабочего, выполняя в цикле последовательно две функции: *BC_WorkerMap* и *BC_WorkerReduce*. Выход из этого цикла происходит, когда функция *BC_WorkerMap* вернет значение *false*. Функция *BC_WorkerMap* выполняет следующие действия:

1) в синхронном режиме (с помощью функции *MPI_Recv*) получает от мастера задание для выполнения очередной итерации;
2) если поле *exit* в полученном задании содержит значение *true*, происходит выход из функции;
3) применяет предопределенную проблемно-зависимую функцию *PC_bsf_MapF* ко всем элементам своего подсписка «Map», помещая результаты в расширенный список «Reduce».

Функция *BC_WorkerReduce* выполняет следующие действия:

1) выполняет оператор Reduce для своей части расширенного списка «Reduce» с помощью функции *BC_ProcessExtendedReduceList*, которая помещает результат в структуру с указателем *extendedReduceResult_P*;
2) пересылает в синхронном режиме (с помощью функции *MPI_Send*) полученный результат мастеру.

2. Визуальный конструктор BSF-программ

В данном разделе описывается процесс проектирования и реализации программной системы BSF-Studio для быстрого создания корректных BSF-программ на языке C++ с использованием описанного в разделе 1 параллельного каркаса. Основными функциональными возможностями BSF-Studio являются следующие:

– пошаговое заполнение проблемно-зависимых частей BSF-каркаса;
– инкрементная (после каждого шага) компиляция программного кода в облачной среде, поддерживающей технологии параллельного программирования OpenMP [11] и MPI [9], с выводом диагностических сообщений компилятора;
– запуск скомпилированного исполняемого кода в облачной среде, поддерживающей технологии параллельного программирования OpenMP и MPI, с выводом результатов выполнения;
– загрузка отлаженного исходного кода.

Программная система BSF-Studio представляет собой веб-приложение с клиент-серверной архитектурой. Клиентская часть представляет собой одностраничное приложение (Single Page Application) [12], серверная часть — Docker-контейнер [13], в котором запускается приложение, выполняющее по запросу компиляцию и запуск программы, и возвращающее результат. Связь между клиентом и севером осуществляется посредством REST API [14].

2.1. Проектирование и реализация клиентской части

Интерфейс клиента BSF-Studio представляет собой мастер (wizard) для заполнения BSF-каркаса. Процесс работы мастера разбивается на следующие *шаги*:

1) определение константных параметров модели BSF и параметров задачи (размерность, число уравнений и др.);
2) определение типов данных текущего приближения и элементов списков «Map» и «Reduce»;
3) определение пользовательских переменных и структур данных;
4) реализация пользовательских функций.

Указанные шаги реализуются путем последовательного заполнения HTML-форм, содержащих текстовые поля для заполнения соответствующих фрагментов программного кода.

Реализация интерфейса выполнена с помощью JavaScript-библиотеки React с открытым исходным кодом [15]. React позволяет создавать интерактивные пользовательские интерфейсы на основе компонентного подхода: фрагменты веб-приложения (компоненты) инкапсулируются и используются независимо друг от друга. А за счет большого количества библиотек с открытым исходным кодом, предоставляющих готовые React-компоненты, разработка приложений сводится к подбору, подключению и использованию сторонних модулей.

Для работы с данными использовалась библиотека Redux [15]. Redux берет на себя функции записи, хранения, организации доступа к данным из любого React-компонента. Для реализации HTML-форм использовалась библиотека Redux Form [16]. Redux Form предоставляет готовые компоненты и программную обвязку для реализации HTML-форм и хранения их данных с использованием Redux. Для реализации текстовых полей, предназначенных для работы с программным кодом, использовалась библиотека React Ace [17]. Она предоставляет компонент, в котором реализовано оформление текста в зависимости от языка программирования, выбранного из предоставляемого списка, нумерация строк, настраиваемые панели инструментов.

По нажатию на кнопки «Compile» и «Run» программный код компонуется в файлы и отправляется посредством HTTP-запроса на сервер, где происходит объединение пользовательских файлов и файлов BSF-каркаса и выполняется их компиляция и/или запуск.

Результаты компиляции и запуска отображаются в интерфейсе веб-приложения. По нажатию на кнопку «Download» формируются и загружаются файлы BSF-каркаса.

2.2. Проектирование и реализация серверной части

Для разработки серверной части приложения BSF-Studio использована технология Docker [18], которая обеспечивает возможность развертывания приложения на базе контейнерной виртуализированной среды. Это позволило создать виртуальное окружение, максимально приближенное к окружению кластерной вычислительной системы. Данный подход позволяет развернуть серверную часть приложения BSF-Studio в рамках любой операционной системы и аппаратного обеспечения, а также упростить процесс развертывания.

Docker предполагает написание сценария, который содержит последовательные инструкции по настройке окружения. В результате выполнения этого сценария строится виртуальный образ, эмулирующий описанное окружение. В рамках этого Docker-сценария необходимо описать на основе какой операционной системы будет функционировать образ, какие подсистемы и библиотеки должны быть установлены, а также произведена настройка и подготовка образа к функционированию приложения внутри него.

Для решения поставленных задач было взято наиболее компактное ядро Linux Alpine [18], содержащее минимальный необходимый набор программ, который позволяет операционной системе функционировать. В качестве компилятора был выбран GNU C Compiler (GCC) [19]. Помимо этого, были установлены библиотеки параллельного программирования OpenMP и MPICH [20]. Для проверки корректности установки и функционирования всех необходимых элементов при сборке образа производится тестовая компиляция нескольких проектов.

Далее необходимо выбрать технологии и средства реализации непосредственно серверного приложения, которое будет осуществлять сборку предоставленного пользователем исходного кода и возвращать результат компиляции посредством REST API.

Серверное приложение разработано на базе программной платформы Node.js, предоставляющей JavaScript-окружение и веб-фреймворка Express.js, который занимается обработкой входящих запросов, и выполняет функции веб-сервера [21]. Также использовались следующие библиотеки:

- Express-request-id — модуль для Express, который позволяет получать параметры из строки запроса [22].
- ShellJS — библиотека, которая позволяет вызывать команды операционной системы [23].
- Multer — библиотека, которая позволяет фреймворку Express принимать и отправлять файлы [24].

Для автоматизации развертывания контейнеризованных приложений клиентской и серверной частей BSF-Studio создан кластер на основе системы Kubernetes [17].

BSF-Studio-API работает согласно следующему алгоритму. Импортируются необходимые библиотеки и модули: Express, express-request-id, ShellJS, Multer. Далее формируются пути до рабочих директорий для текущего запроса и для приложения. Происходит инициализация экземпляра веб-сервера Express и хранилища, в котором будут находиться полученные в текущем запросе файлы и их именование, и инициализация экземпляра Multer. Модуль Express-Request-Id подключается к веб-серверу Express и происходит его запуск. Далее веб-сервер ожидает HTTP-запрос на компиляцию и запуск программного

кода. Такой запрос должен содержать файлы проблемно-зависимой части кода для BSF-каркаса в качестве параметров. При получении запроса происходит его обработка и загрузка присланных файлов и файлов каркаса в рабочую директорию. Далее выполняется компиляция, результаты отправляются клиенту в формате JSON, рабочая директория удаляется.

Исходные тексты BSF-Studio свободно доступны в сети Интернет:

– серверная часть: https://github.com/ezhova-nadezhda/BSF-Studio-API;
– клиентская часть: https://github.com/ezhova-nadezhda/BSF-Studio.

Заключение

В статье была описана программная поддержка модели BSF. Она включает в себя параллельный BSF-каркас и веб-приложение BSF-Studio. Параллельный BSF-каркас представляет собой совокупность файлов исходного кода на языке C++, используемых для быстрого создания BSF-программ. BSF-каркас содержит реализацию всех проблемно-независимых функций, организующих параллельное выполнение программы с использованием библиотек MPI и OpenMP. Он также содержит определения (заголовки) проблемно-зависимых функций, реализуемых пользователем. Отличительной особенностью BSF-каркаса является то, что он предполагает поэтапное заполнение пользовательской части исходных кодов и при этом компилируется на всех этапах. Веб-приложение BSF-Studio представляет собой визуальный конструктор BSF-программ. BSF-Studio обеспечивает поэтапное заполнение проблемно-зависимых частей программного кода, компиляцию и запуск приложения в облачной среде.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 17-07-00352 а, Правительства РФ в соответствии с Постановлением №211 от 16.03.2013 г. (соглашение № 02.A03.21.0011) и Министерства образования и науки РФ (государственное задание 2.7905.2017/8.9).

Литература

1. Ежова Н.А., Соколинский Л.Б. Обзор моделей параллельных вычислений // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 3. С. 58–91. DOI: 10.14529/cmse190304.
2. Leasure B. et al. Parallel Skeletons // Encyclopedia of Parallel Computing. Springer US, 2011. P. 1417–1422. DOI: 10.1007/978-0-387-09766-4_24.
3. Cole M.I. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, 1991.
4. Gonzalez-Velez H., Leyton M. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers // Softw. Pract. Exp. John Wiley & Sons, Ltd. 2010. Vol. 40, no. 12. P. 1135–1160. DOI: 10.1002/spe.1026.
5. Dongarra J. et al. Sourcebook of Parallel Computing. Morgan Kaufmann, 2003. 842 p.
6. Ежова Н.А., Соколинский Л.Б. BSF: модель параллельных вычислений для многопроцессорных систем с распределенной памятью // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 2. С. 32–49. DOI: 10.14529/cmse180204.

7. Ежова Н.А., Соколинский Л.Б. Исследование масштабируемости итерационных алгоритмов при суперкомпьютерном моделировании физических процессов // Вычислительные методы и программирование. 2018. Т. 19, № 4. С. 416–430. DOI: 10.26089/NumMet.v19r437.
8. Gropp W. et al. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard // Parallel Comput. 1996. Vol. 22, no. 6. P. 789–828. DOI: 10.1016/0167-8191(96)00024-5.
9. Gropp W. MPI3 and Beyond: Why MPI Is Successful and What Challenges It Faces // Recent Advances in the Message Passing Interface. EuroMPI 2012. Lecture Notes in Computer Science, Vol. 7490 / Ed. by J.L. Träff, S. Benkner, J.J. Dongarra. Springer, 2012. P. 1–9. DOI: 10.1007/978-3-642-33518-1_1.
10. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT Press, 1999. 395 p.
11. Pas R. van der, Stotzer E., Terboven C. Using OpenMP — The Next Step: Affinity, Accelerators, Tasking, and SIMD (Scientific and Engineering Computation). 1st ed. MIT Press, 2017. 392 p.
12. Patel Y. White Paper On Single Page Application. Knowarth. 2015. URL: https://www.knowarth.com/wp-content/uploads/2015/02/Single-Page-Application-White-Paper.pdf (дата обращения: 19.09.2019)
13. Docker Open Source Engine Guide. SUSE Linux Enterprise Server 15 SP1. SUSE LLC. 2019. URL: https://documentation.suse.com/sles/15-SP1/pdf/book-sles-docker_color_en.pdf (дата обращения: 19.09.2019)
14. Allamaraju S. RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity. 1st edition. Yahoo Press, 2010. 316 p.
15. Banks A., Porcello E. Learning React Functional Web Development with React and Redux. O'Reilly Media, 2017. 350 p.
16. Rasmussen E. Redux Form: The Best Way to Manage Your Form State in Redux. URL: https://redux-form.com/8.2.2/docs/api/ (дата обращения: 25.02.2018).
17. Hrisho J. React-Ace (GitHub Repository). URL: https://github.com/securingsincity/react-ace (дата обращения: 25.02.2019).
18. Matthias K., Kane S.P. Docker: Up & Running: Shipping Reliable Containers in Production. 2nd Edition. O'Reilly Media, 2018. 352 p.
19. Rothwell T., Youngman J. The GNU C Reference Manual. Free Software Foundation, Inc, 2007. P. 86. URL: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf (дата обращения: 20.02.2018).
20. MPICH Documentation and Guides. URL: https://www.mpich.org/documentation/guides/ (дата обращения: 20.02.2018).
21. Brown E. Web Development with Node and Express: Leveraging the JavaScript Stack. 1st ed. O'Reilly Media, 2014. 332 p.
22. Strukchinsky V. Express-Request-Id (GitHub Repository). URL: https://github.com/floatdrop/express-request-id (дата обращения: 17.03.2018).
23. Fischer N., Freitag B. ShellJS - Unix Shell Commands for Node.js (GitHub Repository). URL: https://github.com/shelljs/shelljs (дата обращения: 13.03.2018).
24. Unnebäck L. Multer (GitHub Repository). URL: https://github.com/expressjs/multer (дата обращения: 17.03.2018).

Ежова Надежда Александровна, аспирант, кафедра системного программирования, Южно-Уральский государственный университет (национальный исследовательский университет) (Челябинск, Российская Федерация)

Соколинский Леонид Борисович, д.ф.-м.н., профессор, проректор по информатизации, Южно-Уральский государственный университет (национальный исследовательский университет) (Челябинск, Российская Федерация)

DOI: 10.14529/cmse190406

SOFTWARE SUPPORT OF THE PARALLEL COMPUTATION MODEL BSF

© 2019 N.A. Ezhova, L.B. Sokolinsky

South Ural State University (pr. Lenina 76, Chelyabinsk, 454080 Russia)

E-mail: EzhovaNA@susu.ru, Leonid.Sokolinsky@susu.ru

Received: 08.11.2019

The paper is devoted to the software support of the parallel computation model BSF (Bulk Synchronous Farm) intended to iterative algorithms with high computational complexity, which developed for multiprocessor systems with distributed memory of exaflops performance level. Software support includes parallel BSF-skeleton and BSF-Studio web-application. The definition and classification of parallel skeletons are given. The logic and file structure of new BSF-skeleton are described. The BSF-skeleton is a collection of C++ source code files used for BSF-programs development. A detailed description of the design and implementation of the BSF-Studio web-application, which is a visual constructor of BSF-programs, is given. BSF-Studio provides compilation and execution of program code.

Keywords: parallel computation model, BSF, parallel skeleton, BSF-Studio, web-application.

FOR CITATION

Ezhova N.A., Sokolinsky L.B. Survey of Parallel Computation Models. *Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering.* 2019. vol. 8, no. 4. pp. 84–99. (in Russian) DOI: 10.14529/cmse190406.

References

1. Ezhova N.A., Sokolinsky L.B. Survey of Parallel Computation Models. Bulletin of South Ural State University. Series: Computational Mathematics and Software Engineering. 2019. vol. 8, no. 3. pp. 58–91. (in Russian) DOI: 10.14529/cmse190304.
2. Leasure B. et al. Parallel Skeletons. Encyclopedia of Parallel Computing. Springer US, 2011. pp. 1417–1422. DOI: 10.1007/978-0-387-09766-4_24.
3. Cole M.I. Algorithmic Skeletons : Structured Management of Parallel Computation. MIT Press, 1991.
4. Gonzalez-Velez H., Leyton M. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. Softw. Pract. Exp. John Wiley & Sons, Ltd. 2010. vol. 40, no. 12. pp. 1135–1160. DOI: 10.1002/spe.1026.
5. Dongarra J. et al. Sourcebook of Parallel Computing. Morgan Kaufmann, 2003. 842 p.
6. Ezhova N.A., Sokolinsky L.B. BSF: Parallel Computation Model for Multiprocessor Systems with Distributed Memory. Bulletin of South Ural State University. Series: Computational Mathematics and Software Engineering. 2018. vol. 7, no. 2. pp. 32–49. (in Russian) DOI: 10.14529/cmse180204.

7. Ezhova N.A., Sokolinsky L.B. Scalability Evaluation of Iterative Algorithms Used for Supercomputer Simulation of Physical Processes. Proceedings — 2018 Global Smart Industry Conference, GloSIC 2018. Article number 8570107. IEEE, 2018. 10 p. DOI: 10.1109/GloSIC.2018.8570131.
8. Gropp W. et al. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Comput. 1996. vol. 22, no. 6. pp. 789–828. DOI: 10.1016/0167-8191(96)00024-5.
9. Gropp W. MPI3 and Beyond: Why MPI Is Successful and What Challenges It Faces. Recent Advances in the Message Passing Interface. EuroMPI 2012. Lecture Notes in Computer Science, vol. 7490. / Ed. by J.L. Träff, S. Benkner, J.J. Dongarra. Springer, 2012. pp. 1–9. DOI: 10.1007/978-3-642-33518-1_1.
10. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT Press, 1999. 395 p.
11. Pas R. van der, Stotzer E., Terboven C. Using OpenMP — The Next Step: Affinity, Accelerators, Tasking, and SIMD (Scientific and Engineering Computation). 1st ed. MIT Press, 2017. 392 p.
12. Patel Y. White Paper On Single Page Application. Knowarth. 2015. Available at: https://www.knowarth.com/wp-content/uploads/2015/02/Single-Page-Application-White-Paper.pdf (accessed: 19.09.2019)
13. Docker Open Source Engine Guide. SUSE Linux Enterprise Server 15 SP1. SUSE LLC. 2019. Available at: https://documentation.suse.com/sles/15-SP1/pdf/book-sles-docker_color_en.pdf (accessed: 19.09.2019)
14. Allamaraju S. RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity. 1st ed. Yahoo Press, 2010. 316 p.
15. Banks A., Porcello E. Learning React Functional Web Development with React and Redux. O'Reilly Media, 2017. 350 p.
16. Rasmussen E. Redux Form: The Best Way to Manage Your Form State in Redux. Available at: https://redux-form.com/8.2.2/docs/api/ (accessed: 25.02.2018).
17. Hrisho J. React-Ace (GitHub Repository). Available at: https://github.com/securingsincity/react-ace (accessed: 25.02.2019).
18. Matthias K., Kane S.P. Docker: Up & Running: Shipping Reliable Containers in Production. 2nd ed. O'Reilly Media, 2018. 352 p.
19. Rothwell T., Youngman J. The GNU C Reference Manual. Free Software Foundation, Inc, 2007. 86 p. Available at: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf (accessed: 20.02.2018).
20. MPICH Documentation and Guides. Available at: https://www.mpich.org/documentation/guides/ (accessed: 20.02.2018).
21. Brown E. Web Development with Node and Express: Leveraging the JavaScript Stack. 1st ed. O'Reilly Media, 2014. 332 p.
22. Strukchinsky V. Express-Request-Id (GitHub Repository). Available at: https://github.com/floatdrop/express-request-id (accessed: 17.03.2018).
23. Fischer N., Freitag B. ShellJS — Unix Shell Commands for Node.js (GitHub Repository). Available at: https://github.com/shelljs/shelljs (accessed: 13.03.2018).
24. Unnebäck L. Multer (GitHub Repository). Available at: https://github.com/expressjs/multer (accessed: 17.03.2018).