DOI: 10.15514/ISPRAS-2019-31(2)-8

Полуавтоматический подход к параллельному решению задач с использованием модели Multi-BSP

М.О. Аланиз, ORCID: 0000-0001-9984-2248 <marcelo.alaniz@fing.edu.uy>
С.Е. Несмачнов Кановас, ORCID: 0000-0002-8146-4012 <sergion@fing.edu.uy>
Республиканский университет,
Уругвай, г. Монтевидео, ул. Эрреры-и-Рейсига, д. 565

Аннотация. Модель Multi-Bulk Synchronous Parallel (Multi-BSP) – это модель параллельного программирования для многоядерных машин, которая расширяет классическую модель Bulk Synchronous Parallel. Multi-BSP направлена на поддержку разработки алгоритмов и оценки времени их работы. Эта модель в значительной степени опирается на правильное вычисление параметров, которые характеризуют оборудование. Конечно, использование оборудования также зависит и от особенностей задач и алгоритмов, применяемых для их решения. В этой статье представлен полуавтоматический подход к решению задач с применением параллельных алгоритмов на основе модели Multi-BSP. Во-первых, характеристики конкретного многоядерного компьютера определяются путем применения автоматической процедуры. После этого аппаратная архитектура, обнаруженная на предыдущем этапе, применяется для разработки переносимого параллельного алгоритма. Наконец, выполняется точная настройка параметров для повышения общей эффективности. Мы предлагаем бенчмарк для измерения параметров, которые характеризуют расходы на коммуникации и синхронизацию в конкретном оборудовании. Наш подход обнаруживает иерархическую структуру многоядерной архитектуры и вычисляет параметры для каждого уровня. Вторым вкладом нашего исследования является предложение системы поддержки Multi-BSP. Она позволяет разрабатывать алгоритмы, применяя рекурсивную методологию к иерархическому дереву, уже построенному с помощью бенчмарка, уделяя особое внимание трем элементарным функциям и основываясь на стратегии «разделяй и властвуй». Валидация предлагаемого метода производилась путем изучения алгоритма, реализованного в прототипе механизма Multi-BSP, тестирования различных конфигураций параметров, которые лучше всего подходят для каждой задачи, и использования трех различных высокопроизводительных многоядерных компьютеров.

Ключевые слова: высокопроизводительные исчисления; бенчмарк; многоядерное программирование; модель BSP

Для цитирования: Аланиз М.О., Несмачнов Кановас С.Е. Полуавтоматический подход к параллельному решению задач с использованием модели Multi-BSP. Труды ИСП РАН, том 31, вып. 2, 2019 г., стр. 97-120. DOI: 10.15514/ISPRAS-2019-31(2)-8

A semi-automatic approach for parallel problem solving using the Multi-BSP model

M.O. Alaniz, ORCID: 0000-0001-9984-2248 <marcelo.alaniz@fing.edu.uy>
S.E. Nesmachnow Cánovas, ORCID: 0000-0002-8146-4012 <sergion@fing.edu.uy>
Universidad de la República,
Herrera y Reissig 565, Montevideo, Uruguay

Abstract. The Multi-Bulk Synchronous Parallel (Multi-BSP) model is a recently proposed parallel programming model for multicore machines that extends the classic Bulk Synchronous Parallel model. Multi-BSP aims to be a useful model to design algorithms and estimate their running time. This model heavily relies on the right computation of parameters that characterize the hardware. Of course, the hardware utilization also depends on the specific features of the problems and the algorithms applied to solve them. This article introduces a semi-automatic approach for solving problems applying parallel algorithms using the Multi-BSP model. First, the specific multicore computer to use is characterized by applying an automatic procedure. After that, the hardware architecture discovered in the previous step is considered in order to design a portable parallel algorithm. Finally, a fine tuning of parameters is performed to improve the overall efficiency. We propose a specific benchmark for measuring the parameters that characterize the communication and synchronization costs in a particular hardware. Our approach discovers the hierarchical structure of the multicore architecture and compute both parameters for each level that can share data and make synchronizations between computing units. A second contribution of our research is a proposal for a Multi-BSP engine. It allows designing algorithms by applying a recursive methodology over the hierarchical tree already built by the benchmark, focusing on three atomic functions based in a divide-and-conquer strategy. The validation of the proposed method is reported, by studying an algorithm implemented in a prototype of the Multi-BSP engine, testing different parameter configurations that best fit to each problem and using three different high-performance multicore computers.

Keywords: High Performance Computing; Benchmark; Multicore Programming; BSP Model

For citation: Alaniz M.O., Nesmachnow Cánovas S.E. A semi-automatic approach for parallel problem solving using the Multi-BSP model. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 2, 2019. pp. 97-120 (in Russian). DOI: 10.15514/ISPRAS-2019-31(2)-8

1. Введение

Модель BSP была представлена в конце 1980-ых гг. как модель программирования для распределенных вычислительных систем в предположении, что у каждого вычислительного узла имеется только одно ядро [9]. Хотя модель успешно использовалась в 1990-ых гг., постепенно ее популярность уменьшилась по причине возникновения в течении последних десяти лет новых многоядерных архитектур. Поскольку оценка возможностей компьютеров приобрела новое значение, модель BSP была расширена до Multi-BSP [10]. Multi-BSP расширяет BSP в двух направлениях:

- это иерархическая модель с произвольным числом компонентов, учитывающая физическую структуру многочисленных уровней памяти и кэша как одном чипе, так и в многокристальных архитектурах;
- в модели Multi-BSP, в отличие от исходной модели BSP, в качестве дополнительного параметра учитывается размер памяти на каждом уровне.

Учитывая характерные черты, присущие современным параллельным архитектурам, Multi-BSP обеспечивает более полную модель, позволяющую конструировать эффективные параллельные алгоритмы. Исследование, которому посвящена эта статья, состоит в разработке комплексного подхода к проектированию и реализации параллельных приложений на основе модели Multi-BSP с использованием самых современных инструментов и учетом требований не только производительности, но и переносимости алгоритмов. Комплексная разработка предполагает наличие описанной аппаратной

архитектуры, шаблона для разработки алгоритма, функции оценки стоимости алгоритма, а также конкретной методологии для реализации алгоритма на параллельной аппаратуре.

Основными результатами описываемого исследования являются:

a) предложение конкретной методологии для обнаружения иерархических структур многоядерной архитектуры и определение параметров, характеризующих расходы на коммуникации и синхронизацию в данной параллельной аппаратуре;
b) разработка системы поддержки Multi-BSP (Multi-BSP Engine), позволяющей разрабатывать алгоритмы при помощи применения рекурсивного шаблона «разделяй и властвуй» к иерархическому дереву, ранее построенному на основе применения бенчмарка;
c) валидация предложенного подхода, включающая разработку алгоритма с применением системы поддержки Multi-BSP, оценку различных конфигурационных параметров, которые лучше всего подходят для решения данной задачи, и реализацию алгоритма для трех разных высокопроизводительных многоядерных компьютеров, включая сопроцессор Xeon Phi, который не использовался в предыдущих аналогичных исследованиях.

Часть исследований, представленных в данной статье, была выполнена в рамках проекта «Scheduling evaluation in heterogeneous computing systems with hwloc» (SEHLOC). Основная цель проекта (SEHLOC) заключалась в разработке подсистем поддержки времени выполнения, которые позволяют комбинировать характеристики программного кода приложений и информацию о топологии вычислительных платформ для обеспечения планирования выполнения задач, позволяющего получить выигрыш от соответствия программных и аппаратных средств и обеспечить способ эффективного выполнения реалистичных приложений.

Эта статья расширяет нашу предыдущую работу [1], в которой были представлены первые идеи о применении Multi-BSP. В данной статье представлен более полный подход и представлена система Multi-BSP, поддерживающая весь процесс разработки параллельных алгоритмов.

Статья устроена следующим образом. В разд. 2 представлены модели BSP и Multi-BSP, а также автоматическое инструментальное средство, позволяющее обнаруживать особенности архитектуры и содействующее обеспечению переносимости приложений. В разд. 3 описаны родственные работы, относящиеся к разработке и использованию бенчмарков BSP, а также разработка и реализация бенчмарка MBSPDiscover. В разд. 4 описываются преимущества разработки алгоритмов с использованием Multi-BSP и разработанной системы поддержки, причем основное внимание уделяется рекурсивному построению шаблонов параллельных приложений для обеспечения переносимости. Функция стоимости разработана на основе модели затрат Multi-BSP. Система поддержки проверена на примере разработки простого алгоритма и реализации этого алгоритмы для трех параллельных машинах, характеристики которых были получены с использованием предложенного бенчмарка (разд. 5). Наконец, в разд. 6 представлены выводы и сформулированы основные направления будущих исследований.

2. Модели BSP и Multi-BSP

Чтобы установить контекст этой статьи, в этом разделе мы описываем модели BSP и Multi-BSP. Представлено краткое описание традиционной модели BSP и того, как модель доросла до включения концепции многоядерности, что привело к приданию особого значения иерархии компонентов. Далее описываются аналитические методы прогнозирования, что требуется для понимания основ обеих моделей. После этого описывается стоимостная функция Multi-BSP. В конце этого раздела мы рассуждаем о необходимости автоматического

процесса обнаружения особенностей архитектуры для обеспечения переносимости, а также описываем применение для этих целей конкретного пакета программ.

2.1 Оригинальная модель BSP

Модель BSP основана на понятии абстрактного параллельного компьютера, который полностью моделируется набором параметров: *количеством доступных процессоров (p)*, *скоростью процессора (s)*, *стоимостью коммуникаций (g)* и *стоимостью синхронизации (l)*. На основе этих можно точно вычислить время выполнения любого BSP-алгоритма.

В модели BSP вычисления организуются в виде последовательности глобальных супершагов, каждый из которых состоит из трех фаз:

a) каждый участвующий процессор выполняет локальные вычисления, во время которых его процессы могут использовать только значения, хранящиеся в локальной памяти данного процессора;
b) процессы обмениваются данными, чтобы обеспечить возможность доступа к удаленным данным;
c) каждый участвующий процесс должен достичь следующего барьера синхронизации.

После этого можно начать следующий супершаг.

Фактически, эта модель программирования соответствует парадигме «одна программа – много данных» (Single Program Multiple Data, SPMD): несколько копий программ на C/C ++, работают на p процессорах; обмен данными между копиями и их синхронизация производятся с использованием специальных библиотек, таких как BSPlib [5] или PUB [3]. Помимо определения абстрактной машины и наложения структуры на параллельные программы, модель BSP предоставляет функцию стоимости, опирающуюся на параметры архитектуры.

Общее время выполнения программы BSP может быть вычислено как накопленная сумма стоимости ее супершагов, где стоимость каждого супершага является суммой трех значений:

d) w – максимальный объем вычислений, выполняемых каждым процессором;
e) $h \times g$, где h – максимальное число сообщений, отправленных/полученных каждым процессором, причем каждое сообщение занимает g единиц времени и
f) l – временн*ая* стоимость барьера, синхронизирующего процессоры.

Влияние компьютерной архитектуры учитывается параметрами g и l. Эти значения, наряду со скоростью процессора s, могут быть определены эмпирически для каждого параллельного компьютера путем выполнения бенчмарков во время развертывания.

2.2 Модель Multi-BSP

Современные суперкомпьютеры состоят из высокопараллельных многоядерных узлов. Для обеспечения эффективности этих узлов потребовалось усовершенствование подсистемы основной памяти путем добавления нескольких иерархических уровней кэшей, а также обеспечения удаленного доступа к основной памяти, что приводит к архитектуре с неоднородным доступом к памяти (Non-Uniform Memory Access, NUMA).

В 2010 году Лесли Вэлиант (Leslie G. Valiant) обновил модель BSP, чтобы учесть эту ситуацию, что привело к появлению модели Multi-BSP [10]. Те же абстракции и архитектура шлюза, которые использовались в исходной BSP, в Multi-BSP были адаптированы для многоядерных машин, и экземпляр модели стали описывать в виде древовидной структуры вложенных компонентов, где листья – это процессоры, а каждый внутренний узел – компьютер BSP с локальной основной памятью или некоторым устройством хранения данных.

Формально машина Multi-BSP определяется списком уровней. Каждый уровень описывается четырьмя параметрами p_i, g_i, L_i, m_i.

- p_i – это число компонентов уровня $(i-1)$ в компоненте уровня i. Для $i = 1$ компонентами первого уровня являются p_1 реальных процессоров, которые можно считать компонентами уровня 0. Продолжительность одного вычислительного шага такого процессора над словом основной памяти на уровне 1 принимается в качестве базовой единицы времени.
- g_i – это параметр стоимости коммуникаций, определяемый как частное от деления числа операций, которые процессор может выполнить за одну секунду на количество слов, которые могут быть переданы за одну секунду между памятью компонента уровня i и его родительским компонентом на уровне $(i+1)$. Слово определяется как фрагмент данных, над которыми выполняются операции процессора. В модели предполагается, что основная память на уровне 1 привязана к процессорам, следовательно, скорость передачи данных (соответствующая значению g_0) также имеет значение 1.
- Li – это стоимость барьерной синхронизации для супершага на уровне i. Это определение предполагает требование барьерной синхронизации для подкомпонентов каждого компонента, но не предполагает синхронизацию подкомпонентов вышестоящих ветвей в иерархии компонентов.
- m_i – это число слов в основной памяти внутри компонента на уровне i, которая не находится ни в одном из компонентов уровня $(i-1)$.

Рис. 1 демонстрирует компонент уровня i в модели Multi-BSP. Супершаг уровня i представляет собой набор инструкций, выполняемых в компоненте уровня i, что позволяет каждому из p_i компонентов на уровне $(i-1)$ действовать независимо, включая все супершаги уровня $(i-1)$. Как только все p_i компоненты закончат свои вычисления, они могут обменяться информацией с памятью m_i компонента уровня i. Стоимость коммуникаций этой операции определяется $g_i - 1$. Полная стоимость будет равна $m \times g_i - 1$, где m – это максимальное число слов, передаваемых между памятью компонента уровня i и любым его подкомпонентом уровня $(i-1)$. После прохождения синхронизационного барьера всеми p_i компонентами может начаться следующий супершаг.

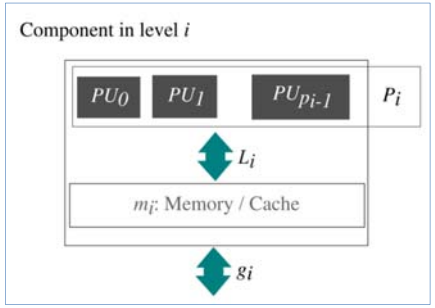

Рис. 1: Схематическое изображение компонента на уровне i в модели Multi-BSP
Fig. 1: Schematic view of a component on level i of the Multi-BSP model

На рис. 2 показана диаграмма компьютера, архитектура которого может быть определена тремя компонентами Multi-BSP (level0, level1 и level2): (1, 0, 0, 0), (4, g_1, L_1, 5118KB) и (8, g_2, L_2, 64 GB). Можно игнорировать level0, поскольку он содержит всего один процессор, и, таким образом, в нем отсутствуют внутренние коммуникации или синхронизация. Соответственно, в этой машине имеются два компонента, которые соответствуют двум уровням иерархии архитектуры.

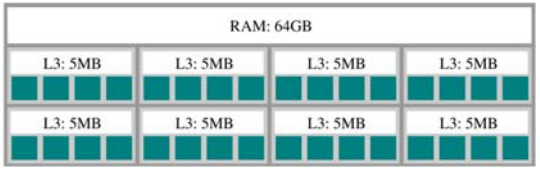

Рис. 2: Модель Multi-BSP: (4, g1, L1, 5118KB), (8, g2, L2, 64 GB)
Fig. 2: Multi-BSP model: (4, g1, L1, 5118KB),(8, g2, L2, 64 GB)

2.3 Прогнозирование затрат для модели Multi-BSP

Так же, как и у других абстрактных вычислительных моделей, одной из главных задач модели Multi-BSP является предоставление четкого представления о времени выполнения компьютерной программы. В этом подразделе представлена математическая формулировка модели стоимости исполнения, основанная на полном определении операционной семантики Альберта-Яна Изельмана (Albert-Jan Yzelman) [11]. Позже в подразделе 4.3 мы представляем упрощенную формулировку и подробно описываем систему поддержки Multi-BSP, рассматриваемую нами, как основной вклад данной работы.

Прогноз затрат на конкретном компьютере выражается в терминах объема вычислений, объема перемещаемых данных и задержки в соответствии с формулой (1), где L соответствует числу уровней в дереве Multi-BSP, N_k – число супершагов на k-м уровне, $h_{k,i}$ – максимум всех h-отношений в i-м супершаге на уровне k, а $w_{k,i}$ – максимум объема всех работ в i-м супершаге на уровне k.

$$T = \sum_{k=0}^{L-1} \left(\sum_{i=0}^{N_k - 1} w_{k,i} + h_{k,i} \times g_k + l_k \right) \qquad (1)$$

Формула (1) соответствует сумме стоимостей супершагов каждого k-го компонента Multi-BSP. Стоимость отдельного супершага состоит из двух независимых слагаемых: стоимость вычислений и стоимость коммуникаций. Стоимость коммуникаций включают в себя стоимость синхронизации (l_k, всегда одно значение на один супершаг), а слагаемое $h_{k,i} \times g_k$, зависящее от числа операций put/get между потоками, формально определяется на основе понятия h-отношения (h-отношением (*h-relation*) называется максимальное число сообщений, получаемых или посылаемых процессором). Выполнение супершага внутри каждого компонента происходит последовательно, а внутри супершага каждый поток работает параллельно. Таким образом, величины $h_{k,i}$ и $w_{k,i}$ соответствуют максимумам всех h-отношений на супершаге i на уровне k и максимуму объемов всех работ на супершаге i на уровне k соответственно.

Для обеспечения переносимости при комплексной разработке с использованием Multi-BSP требуется использовать процедуру для обнаружения свойств используемой компьютерной архитектуры. В бенчмарке Multi-BSP используется переносимый инструмент HardWare LOCality (hwloc) [4], обнаруживающий характеристики используемой аппаратуры. hwloc позволяет получать информацию о компьютере во время выполнения. Мы используем hwloc версии 1.7.2, которая обеспечивает абстракцию (не зависящую от используемой операционной системы, архитектуры и т.д.) иерархической топологии современных архитектур, включая процессоры, узлы памяти NUMA, сокеты, общие кэши, ядра и расположение устройства ввода/вывода.

3. Механизм обнаружения и тестирования для Multi-BSP

Многоядерные архитектуры широко используются для разработки и выполнения высокопроизводительных приложений [6]. Количество ядер, как и уровней кэша в

многоядерной архитектуре за последние годы стабильно увеличивалось. Поэтому существует реальная потребность выявления и оценки различных параметров, которые характеризуют структуру процессорных ядер и основной памяти, не только для того, чтобы понимать и сравнивать различные архитектуры, но и для их разумного использования для более качественной разработки высокопроизводительных приложений. Это обосновывается тем, что возможность повышения производительности при использовании многоядерного процессора очень зависит от программных алгоритмов, их реализации и использования возможностей аппаратного обеспечения.

В модели Multi-BSP производительность параллельного алгоритма зависит от параметров, которые характеризуют многоядерную машину: расходы на коммуникации и синхронизацию, количество ядер и объем кэша. Составление аналитических уравнений для расчета этих параметров – сложная задача. По этой причине жизнеспособным методом оценки производительности и определения характеристик архитектуры является эталонное тестирование (*benchmarking*).

В этом разделе представлен обзор родственных работ по бенчмаркам моделей BSP и Multi-BSP, а также описана разработка и реализация инструмента обнаружения и эталонного тестирования (Multi-BSP-Disc-Bench), предназначенного для оценки параметров g и L, которые характеризуют машину Multi-BSP.

3.1 Родственные работы

Программа BSPbench из пакета BSPEupack [3] является основным инструментом для эталонного тестирования модели BSP. Этот бенчмарк измеряет полное h-отношение, когда каждый процессор отправляет и получает ровно h слов данных. Методология пытается выявить самое медленное коммуникационное взаимодействие, циклически перемещая одиночные слова в другие процессоры. Это показывает, действительно ли системное программное обеспечение комбинирует данные для одного и того же пункта назначения и может ли оно эффективно обрабатывать коммуникации «каждый с каждым». В этих случаях результирующее значение параметр g, полученное программой эталонного тестирования BSPbench, называются пессимистическим.

Пакет программ Oxford BSP [5] включает другую программу эталонного тестирования для BSP – bspprobe. Этот бенчмарк измеряет оптимистические значения g, используя не отдельные слова, а пакеты большего размера. Еще одним вариантом эталонного тестирования BSP является использование инструмента mpibench из пакета MPIedupack [5].

Эталонному тестированию вычислительной модели multi-BSP посвящена недавняя публикация Савади (Abdorreza Savadi) и Делдари (Hossein Deldari) [7], которые использовали подход, очень близкий к тому, который применяли мы. За основу берется эталонное тестирование BSP, однако экземпляр модели определяется по-другому. В отличии от методологии эталонного тестирования, использованной в нашей работе, авторы [7] принимают во внимание мельчайшие детали архитектуры, например, как поддерживается когерентность кэшей для распространения значений в иерархической структуре памяти. В их подходе анализ результатов основывается на сравнивании действительных значений g и L, полученных тестированием, и теоретических значений этих параметров, которые рассчитываются как оптимистические нижние границы, т.е. авторы предполагают, что все нужные данные всегда помещаются в кэше и все ядра работают на максимальной скорости. В этом наш подход отличается, поскольку мы не делаем никаких предположений о базовой аппаратной платформе, а скорее скрываем ее характеристики в результатах тестов. Мы считаем, что эта стратегия лучше подходит для современных архитектур, которые слишком сложны для точного моделирования из-за наличия расширенных, скрытых и/или плохо документированных возможностей.

С практической точки зрения, основной особенностью инструмента обнаружения и эталонного тестирования, который мы описываем в этой статье, является оценка реальных операций Multi-BSP, реализованных на основе библиотеки MulticoreBSP for C [12]. Кроме того, наши результаты проверяются с использованием набора реальных программ Multi-BSP, где реальное время выполнения на нескольких высокопроизводительных платформах сравнивается с прогнозируемым временем с использованием теоретической функции стоимости Multi-BSP.

3.2 Разработка инструментального средства Multicore-BSP-Disc-Bench

При разработке своего инструмента Multi-BSP-Disc-Bench мы использовали общие идеи бенчмарка BSPbench [3] для стандартной модели BSP и расширили его возможности с учетом отличий модели Multi-BSP. Основное отличие бенчмарка для BSP от бенчмарка для Multi-BSP заключается в необходимости получения пар значений параметров g и L для каждого уровня компонентов в модели Multi-BSP. Кроме того, в случае Multi-BSP обработка выполняется внутри многоядерных узлов, а не внешними узлами, соединенными сетью.

Важно подчеркнуть, что качество инструмента сравнения не должно зависеть от конкретной архитектуры. Это дополнительное требование решается путем автоматического обнаружения взаимосвязей различных ядер на каждом уровне кэша. Еще одна важная цель бенчмарка для Multi-BSP состоит в том, чтобы обнаруживать особенности архитектуры во время выполнения. Для этого мы используем инструмент hwloc.

Компоненты предлагаемого бенчмарка описываются в следующих подразделах.

3.2.1 Архитектура и модули программного обеспечения

На рис. 3 представлены архитектура и состав модулей инструментального средства Multi-BSP-Disc-Bench.

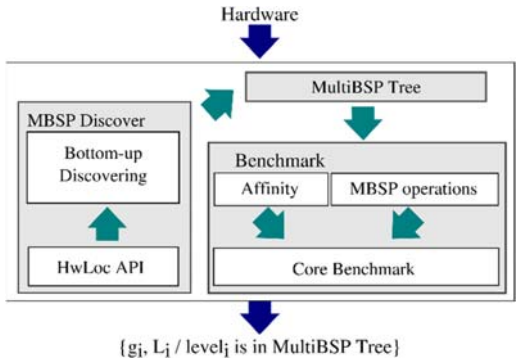

Рис. 3: Схематическое изображение архитектуры программного обеспечения Multi-BSP-Disc-Bench и процесса обнаружения и эталонного тестирования

Fig. 3: Schematic view of the software architecture of Multi-BSP-Disc-Bench, and the discovering and benchmarking process

Функциональные возможности модулей Multi-BSP-Disc-Bench состоят в следующем.

- Модуль обнаружения (*Multi-BSP Discover*). Этот модуль собирает данные об архитектуре и свойствах аппаратуры, используя hwloc, и загружает данные в дерево ресурсов (структура памяти, определенная внутри блока API hwloc).
- Интерфейс (*Multi-BSP Tree*). После построения структуры, содержащей информацию о ресурсах, набор функций интерфейсного модуля проходит по дереву, используя восходящий процесс для построения нового дерева с именем Multi-BSP Tree. Это новое дерево содержит всю информацию, необходимую для поддержки модели Multi-BSP.

- Модуль эталонного тестирования (*Multi-BSP Bench*). Этот модуль извлекает из дерева Multi-BSP информацию об индексах и объеме кэш-памяти ядер для каждого уровня. После этого он измеряет стоимость коммуникаций и стоимость синхронизации через подмодуль Multi-BSP и использует подмодуль нахождения соответствия для закрепления каждого уровня на правильном ядре. Наконец, этот модуль вычисляет результирующие параметры *g* и *L*.

Эти шаги процесса тестирования применяются в соответствии с псевдокодом, показанном в алгоритме 1.

```
1: Multi-BSP-Tree ← Multi-BSP-Discover ()
2: for each level in Multi-BSP-Tree do
3:   [g, L] ← coreBenchmark(level)
4: end for
```

Алгоритм 1. Псевдокод Multi-BSP Discover
Algorithm 1. Multi-BSP Discover pseudocode

Multi-BSP Tree выступает в качестве интерфейса между Multi-BSP Discover и модулем эталонного тестирования. В качестве примера на рис. 4 показана структура, соответствующая конкретной аппаратной архитектуре с 32 ядрами, сгенерированная Multi-BSP Discover.

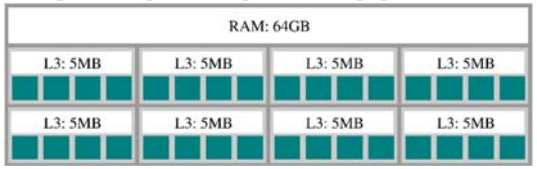

Рис. 4: Структура Multi-BSP Tree, сгенерированная Multi-BSP-Disc-Bench
Fig. 4: Multi-BSP Tree structure generated by Multi-BSP-Disc-Bench

3.2.2 Модуль coreBenchmark

Модуль coreBenchmark разработан для расчетов параметров gi и Li согласно псевдокоду, приведенному в Алгоритме 2.

```
01: setPinning(level.cores indexes)
02: begin(level.cores)
03: rate ← computingRate(level)
04: sync()
05: for h = 0 to HMAX do
06:   initCommunicationPattern(h)
07:   sync()
08:   t0 ← time()
09:   for i = 0 to NITERS do
10:     communication()
11:     sync()
12:   end for
13:   t ← time() − t0
14:   if master then
15:     times.append(t × rate/NITERS)
16:   end if
17: end for
18: level.g, level.L ← leastSquares(times)
19: return (level.g, level.L)
```

Алгоритм 2. Функция coreBenchmark
Algorithm 2. coreBenchmark function

Функция *coreBenchmark* принимает в качестве параметров информацию соответствующего уровня модели Multi-BSP, а данные, используемые для установления соответствия потоков

(т.е. данные об индексах и объеме памяти кэша) хранятся в структуре Multi-BSP Tree. Сначала (строка 1 алгоритма 2) функция *setPinning* модуля установления соответствия (*affinity module*) используется для связи нитей, порожденных функцией begin (строка 2), с ядрами, соответствующими данному уровню. Функция порождает одну нить на каждое ядро данного уровня и рассчитывает скорость вычислений компонента Multi-BSP, используя функцию *computingRate* (строка 3). У каждого уровня имеется набор ядер с общей памятью; впоследствии при эталонном тестировании учитываются только эти ядра.

Функция *computingRate* (строка 3) измеряет время, требуемое для выполнения 2×*n*×DAXPY операций. Подпрограмма DAXPY выполняет векторную операцию $y = \alpha \times x + y$, складывая несколько векторов с двойной точностью с другим вектором двойной точности. DAXPY – это стандартная операция для оценки эффективности вычислительной платформы при выполнении операций с плавающей запятой, интенсивно использующих память, из набора Basic Linear Algebra Subprograms – Level 1 (BLAS1, http://www.netlib.org/blas). После этого выполняется синхронизация для текущего уровня (строка 4), чтобы гарантировать, что для всех потоков имеется правильное значение скорости вычислений.

Мы используем функцию *coreBenchmark* для измерения полной *h*-коммуникации. Это абстракция, которую мы определяем, как расширение абстракции *h*-отношения из стандартной модели BSP, но в этом случае понятие применяется к случаю наличия общей памятью внутри одного узла. *h*-коммуникация – это такая коммуникация, при которой каждое ядро записывает/считывает ровно *h* слов данных. Мы рассматриваем наихудший случай, измеряя самую медленную возможную коммуникацию путем циклического считывания отдельных слов данных из памяти других процессоров. Таким образом, значения g_i и l_i, вычисляемые с использованием нашего бенчмарка, являются пессимистическими значениями, и реальные значения всегда будут лучше. Переменная *h* представляет наибольшее количество слов, прочитанных или записанных в общей памяти данного уровня. *HMAX* – это максимальное значение для всех параметров (*h*), используемых в шаблонах связи для каждого уровня. Значения *HMAX* могут различаться для разных уровней иерархии; мы предлагаем находить подходящие значения с помощью эмпирического анализа.

Время коммуникаций (вычисляемое на основе шаблона *h*-коммуникации) инициализируется процедурой *initCommPattern* (строка 6). Этот процесс повторяется *NITERS* раз (строки 9–12), потому что каждая операция является слишком быстрой, чтобы ее можно было измерить с достаточной точностью. После этого главный поток на каждом уровне сохраняет время, потраченное на каждую *h*-коммуникацию (строка 15).

Наконец, параметры *g* и *L* вычисляются с использованием традиционного метода аппроксимации наименьших квадратов для подгонки данных к линейной модели (линия 18) в соответствии с результатами и аппроксимациями, найденными в родственных работах [1, 5].

Таким образом, метод обеспечивает точные приближения для g_i и L_i.

3.3 Эмпирический анализ h-коммуникаций

Методология, применяемая для измерения h-коммуникаций и затем определения значений параметров g и L, основана на измерении реализации операций Multi-BSP. Мы называем операциями Multi-BSP функции/процедуры, которые требуются для реализации алгоритма, разработанного с использованием вычислительной модели Multi-BSP. В нашей разработке операционный модуль Multi-BSP содержит реализацию этих функций, включая операции, предоставляемые библиотекой MulticoreBSP for C [12]. Эта библиотека устанавливает методологию программирования в соответствии с вычислительной моделью Multi-BSP.

Важно учитывать архитектуру инструмента Multi BSP-Disc-Bench, показанную на рис. 3: если алгоритмы Multi-BSP программируются с использованием других библиотек, можно реконфигурировать инструмент, изменяя операционный модуль Multi-BSP, и повторно

выполнить процедуру обнаружения и эталонного тестирования с новой конфигурацией. Более подробная информация о методологии эмпирической оценки *h*-коммуникации представлена в нашей статье [1].

4. Предлагаемая система поддержки разработки и выполнения алгоритмов мульти-BSP

В этом разделе представлено наше предложение системы поддержки разработки и выполнения алгоритмов мульти-BSP. Предлагаемая система будет поддерживать разделение данных, управление потоками и выполнение, инкапсулируя всю базовую логику, которая будет скрыта для программиста. На самом деле, в предлагаемой системе рекурсивно применяется стратегию «разделяй и властвуй».

4.1 Основные идеи

Предлагаемая система задумана как базовый слой, скрывающий детали реализации, необходимые для работы с алгоритмами Multi-BSP. Основная цель системы состоит в том, чтобы предоставить простой и осмысленный способ разработки и реализации кода Multi-BSP, обращая внимание только на стратегии решения задач, а не фокусируясь на конкретных деталях модели Multi-BSP, таких как управление потоками, разделение данных и распределенное выполнение.

В предлагаемой системе используется процесс обнаружения для определения базовой архитектуры многоядерного компьютера и строится стратегия «разделяй и властвуй» для решения задачи. Стратегия «разделяй и властвуй» применяется рекурсивно на разных уровнях аппаратной архитектуры, которые представляются в виде дерева. Стратегия направлена на решение трех основных проблем: разделение данных, наращивание потоков и свертывание потоков для вычисления окончательных результатов.

Предлагаемая система сможет предоставить программистам несколько преимуществ, три из которых наиболее актуальны:

- у программиста будет иметься улучшенная спецификация его алгоритмов;
- один программный слой будет управлять общими вопросами, свойственными каждой реализации Multi-BSP и
- подход обеспечит переносимость алгоритмов Multi-BSP.

Полезная особенность любой реализации Multi-BSP заключается в том, что программисту требуется гарантия того, что применяемое разделение данных приведет к эффективному использованию кэш-памяти, то есть обеспечит большое число успешных обращений к кэш-памяти и минимизирует промахи. Это обеспечивается естественным образом, если в алгоритме применяется стратегии разделения данных на основе доступного оборудования, а потоки или процессы выполняются в процессорных блоках, ближайших к этой памяти единицах. Как правило, размер раздела данных никогда не должен превышать размер соответствующего размера кэша.

Другая проблема алгоритмов Multi-BSP заключается в необходимости их разработки в тесной связи с архитектурой аппаратуры. В этой ситуации нельзя гарантировать переносимость специально разработанного алгоритма, и, возможно, он будет правильно работать только на машинах того же типа (с теми же размерами кэшей каждого уровня, с тем же распределением процессорных блоков и т.д.). Чтобы решить эту проблему, в предлагаемой системе реализуется общий метод для сокрытия всех специфических деталей аппаратного обеспечения, и программисту нужно будет обеспечить только общие функции, которые могут использоваться в различных архитектурах, не преодолевая трудности, связанные с особенностями конкретной аппаратуры, и обеспечивая таким образом

переносимость. В следующем подразделе описываются особенности предлагаемой системы поддержки Multi-BSP.

4.2 Структура системы поддержки Multi-BSP

Как отмечалось выше, система поддержки Multi-BSP – это обобщенная рекурсивная процедура, которая обходит дерево, представляющее конкретную обобщенную архитектуру аппаратуры. Путь, которого придерживается система, определяет стратегию разделения данных, наращивания и свертывания потоков. В нашей реализации применен алгоритм прямого обхода, но это всего лишь одно из возможных простых решений, позволяющих получить полезные результаты. Алгоритм обхода можно кастомизировать, изменяя используемые пути прохода по дереву.

В системе используется тот же процесс автоматического обнаружения особенностей аппаратуры, который применяется в Multi-BSP-Disc-Bench для генерации Multi-BSPTree. После получения этой информации система обрабатывает дерево рекурсивным образом, причем каждый уровень рекурсии отображается на соответствующий уровень вычислительной модели Multi-BSP.

Псевдокод, показывающий общую схему функционирования системы поддержки Multi-BSP представлен в алгоритме 3. У системы имеются два входных параметра: i) текущий узел дерева, представляющий некоторый компонент Multi-BSP, ii) данные для работы. Вся информация, нужная для определения соответствия потоков, уже доступна в компоненте Multi-BSP. В структуре данных имеются указатели на подкомпоненты (*t.sons*), и у каждого подкомпонента имеются правильные индексы для работы.

У каждого потока имеется уникальный идентификация $p \in [0 \ldots n-1]$, где n – число потоков, полученное при помощи вызове функции *bsp_nprocs*. Идентификатор потока p определяется путем вызова функции *bsp pid* (строки 3 и 4 в алгоритме 3).

После этого, к данных применяется функция разделения с использованием в качестве параметров значений *p* и *n* (строка 5). До обработки дерева требуется барьерная синхронизация (функция *bsp_synch()*), чтобы гарантировать доступность соответствующего частного фрагмента данных в каждом из потоков (строка 5). Затем рекурсия начинает обрабатывать узлы дерева.

```
01: bsp_set_pin(t.sons){ Affinity using sons components }
02: bsp_begin()  { Spawn threads }
03: n ← bsp_nprocs() { Amount of threads in the current level }
04: p ← bsp_pid() { thread id / component number at level n }
05: dpi ← partition(|d|, p, n) { Data for thread p and level i }
06: bsp_synch()  { Sync to guarantee all threads have their partitions }
07: if n > 1 then { is not a leaf component / it has sons }
08:   foreach tson, i in t.sons do
09:     vr[i] ← mbspEngine(dpi, tson) { recursion down over sons}
10:   end for
11:   bsp synch() { Waiting to receive the result of every son }
12:   if master then
13:     r ← reduce(vr) { The master of the level executes reduce}
14:   end if
15: else
16:   r ← work(dij) { The leaf thread executes work function }
17: end if
18: bsp_synch() { Wait for master to have r or sons running work}
19: return r
```

Алгоритм 3. Система поддержки Multi-BSP
Algorithm 3. Multi-BSP engine

Обработке данных в тех компонентах, которые не являются листовыми узлами, соответствует код в строках 7–16. Выполняется рекурсивный вызов процедуры *mbspEngine* (строки 8–10) с использованием в качестве параметров текущего раздела данных и узла дерева для каждого подкомпонента. Каждый вызов возвращает результат, который сохраняется в векторе *vr*. Чтобы гарантировать, что все результаты хранятся в *vr*, необходима синхронизация (строка 11). Затем вызывающий поток (то есть главный поток) редуцирует значения, применяя к вектору *vr* функцию *reduce* (строки 12–14). Наконец, действия, выполняемые при достижении рекурсией листовых узлов Multi-BSPTree, показаны в строках 15–17, где выполняется функция *work* для вычисления частичных результатов.

На рис. 5 показано типичная работа системы при решении разделяемой задачи, где применяются функции, определенные пользователем: *partition*, *work* и *reduce*. Работа происходит на компьютере, архитектура которого содержит два уровня, каждый из которых имеет два подкомпонента. Каждый серый квадрат представляет экземпляр компонента Multi-BSP, а каждая стрелки представляют связи между компонентами.

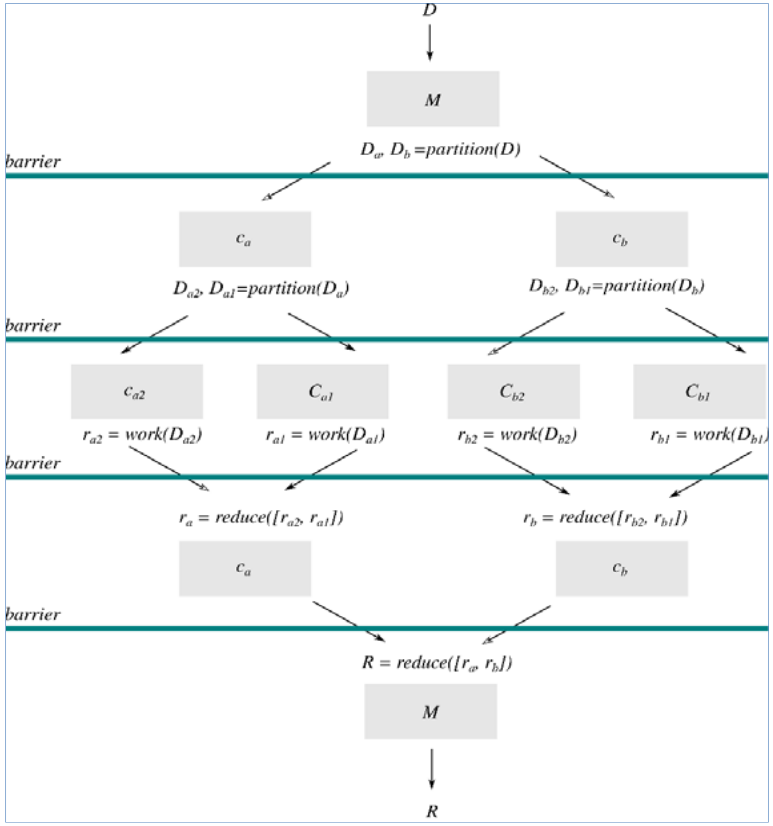

Рис. 5: Схематическое изображение работы MBSPEngine
Fig. 5: Schematic view of an MBSPEngine execution

4.3 Оценочная функция для предлагаемой системы

В системе имеются три последовательных этапа на каждом шаге *T*, выполняемых над деревом архитектуры: рекурсивный обход дерева сверху вниз (*CD*), вызов функции *work* на уровне листьев (*CW*) и возврат из рекурсии снизу вверх (*CU*). Каждый шаг *T* представляет собой работу, необходимую для обработки фрагмента данных *D*.

Стоимость выполнения системы представляет собой сумму значений стоимостных функций для каждого из трех последовательных шагов: $CT = CD + CW + CU$. Как показано ниже, значение стоимостной функции для каждого шага и компонента вычисляется на основе формулы (1).

Рекурсия сверху вниз. Декомпозиция данных фрагмента *D* выполняется с применением функции partition только один раз (то есть один супершаг) на уровень. Тогда соответствующее значение N_k в формуле (1) равно 1, и поскольку функция *partition* является последовательной и безопасной для потоков (т.е. не требует параллельных вычислений), для нее можно использовать стандартную нотацию «О большое», в результате чего получается формула (2).

$$C_D = \sum_{k=0}^{L-1} (O(partition_k) + h_k \times g_k + l_k) \qquad (2)$$

У компонента, работающего на этапе разделения, имеются *p* подкомпонентов, с каждым из которых он нуждается в коммуникации. Поэтому, значение h_k для каждого разделения максимально, что приводит к формуле (3).

$$C_D = \sum_{k=0}^{L-1} \left(O(partition_k) + \frac{D_k}{p_k} \times g_k + l_k \right) \qquad (3)$$

Work. Функция *work* еще проще, поскольку она выполняется только один раз на одном конкретном уровне: листьях дерева, соответствующих процессорным блокам каждого компонента. Тогда N_k и *L* равны 1, а стоимость определяется формулой (4).

$$C_w = O(work_{leaf}) + l_0 \qquad (4)$$

Рекурсия снизу вверх. Стоимость восходящей рекурсии складывается из стоимости функции *reduce* на каждом уровне. Важно, что каждое выполнение функции *reduce* зависит от количества непосредственных потомков этого уровня. После вызова функции *reduce* каждый непосредственный потомок вернет одно значение, что подразумевает наличие одой коммуникации снизу вверх для каждого подкомпонента. Если у компонента на уровне *k* имеется p_k подкомпонентов, выполняющих *reduce*, то количество сообщений (h_k) будет равно p_k, по одному на каждый подкомпонент. Тогда стоимость задается формулой (5).

$$C_D = \sum_{k=0}^{L-1} (O(reduction_k) + p_k \times g_k + l_k) \qquad (5)$$

Результирующая формула (6) дает время выполнения алгоритма, разработанного для системы поддержки Multi-BSP.

$$C_T = O(work_{leaf}) + l_0 + \sum_{k=0}^{L-1} \left(O(partition_k) + O(reduction_k) + p_k \times g_k + \frac{D_k}{p_k} \times g_k + 2l_k \right) \qquad (6)$$

Следующий подраздел представляет пример алгоритма, разработанного при помощи предлагаемой системы путем задания конкретных спецификаций функций partition, work и reduce

4.4 Пример применения MBSPEngine

В этом подразделе описывается простой пример применения предлагаемой системы для решения простой декомпозируемой задачи. Этот пример обеспечивает ориентир для разработки и реализации более сложных алгоритмов путем определения всего трех основных функций, включаемых в систему: *partition*, *work* и *reduce*.

Функция *partition* разбивает исходные данные по одному фрагменту на каждый подкомпонент. Обработка выполняется рекурсивно над деревом MBSP, и фрагмент данных компонента является источником для следующего вызова функции *partition* внутри его подкомпонентов. Когда рекурсия достигает фазы остановки, в компоненте на уровне 0 на процессорах выполняется функция *work*. Когда функция *work* возвращает результат, мастер-процесс объединяет результаты подкомпонентов путем вызова функции *reduce*. Результат функции *reduce* отправляется в родительский компонент. Процесс выполнения разделений, работы на процессорах и редукции результатов представлен на рис. 5.

Важно отметить, что не каждый параллельный алгоритм будет соответствовать общей схеме, поддерживаемой в нашей системе. Рекурсивная стратегия «разделяй и властвуй» обычно не наилучшим образом подходит для задач, данные которые не могут быть разделены произвольным образом. Чтобы можно было разрабатывать переносимые алгоритмы MultiBSP с применением нашей системы, нужно, чтобы допускалась возможность рекурсивного разделения данных по компонентам раскрываемой аппаратной архитектуры.

Кроме того, программист заранее не знает архитектуру компьютеров, для которых он разрабатывает конкретную программу. Программист может разработать конкретный алгоритм, тесно связанный с данным компьютером, но в этом случае весьма вероятно, что алгоритм не сможет эффективно использовать функции других аппаратных платформ (например, когда доступно большее количество вычислительных ресурсов/ядер). В некоторых случаях алгоритм вообще не сможет работать на другой архитектуре. Использование предложенной системы позволяет программисту разрабатывать и реализовывать переносимые алгоритмы. Такие алгоритмы обнаружат архитектуру компьютера, на котором они выполняются, и воспользуются возможностями доступных ресурсов и топологии. В соответствии с обнаруженной информацией задачи разделения данных будут выполняться на вводном этапе, затем на доступных ресурсах будет выполняться функция work, а результаты будут редуцироваться на этапе свертывания рекурсии. Таким образом, для любого алгоритма, который вписывается в общую схему, поддерживаемую системой (то есть, если его данные могут быть разделены произвольны произвольным образом), она предоставляет полезный способ прозрачного и эффективного использования преимуществ базовой аппаратной архитектуры.

Чтобы лучше проиллюстрировать преимущества предложенного движка, в этом подразделе в качестве примера мы представляем алгоритм вычисления скалярного произведения двух векторов. Очевидно, что он подходит для рекурсивной стратегии «разделяй и властвуй» (т.е. следуя подходу параллелизма по данным) и, следовательно, и для нашей системы. Алгоритм прост, но он очень полезен, чтобы показать, как можно работать с предлагаемый системой.

Начнем с определения трех функций, необходимых для двигателя. Функция *Partition* (Алгоритм 4) разделяет исходные данные на основе номера компонента и числа компонентов на данном уровне. Для представленного примера функция обеспечивает разделение, пригодное для алгоритма вычисления скалярного произведения: фрагмент общих данных, разделенный между заданным числом компонентов. Конкретный номер каждого номера компонента используется для определения фрагмента, который будет использоваться компонентом.

```
01 Partition(interval, componentNumber, n) {
02   sliceSize = interval.length / n
02   return interval.slice [
```



```
03       sliceSize * componentNumber,
04       sliceSize * (componentNumber+1)
05   ]
06 }
```

Алгоритм 4. Функция разделения для примера скалярного произведения
Algorithm 4. Partition function for dot product instance

Функция *Work* (алгоритм 5) получает фрагмент, или интервал исходных данных. В данном случае эта функция выполняется только для листовых компонентов. Это самые элементарные компоненты, то есть процессоры со своей ближайшей памятью. Как показано на рис. 5, каждый поток, работающий на этом уровне, напрямую соответствует процессору. Как показано ниже, возвращаемое значение будет использоваться функцией *Reduce*.

```
01 Work (slice) {
02     for value in slice {
03        result += value*value
04     }
05     return result
06 }
```

Алгоритм 5. Функция work для примера скалярного произведения
Algorithm 5. Work function for dot product instance

Наконец, функция Reduce (алгоритм 6) получает массив значений. Входные значения получаются либо в результате выполнения функции Work, либо в результате другого выполнения функции Reduce в непосредственно подчиненных подкомпонентах.

```
01 Reduce( arrayValues ) {
02    for v in arrayValues {
03       result += arrayValues[i]
04    }
05 }
```

Алгоритм 6. Функция reduce для примера скалярного произведения
Algorithm 6. Reduce function for dot product instance

5. Экспериментальный анализ

В этом разделе приводятся значения параметров *g* и *L*, полученные для разных архитектур с использованием предложенного бенчмарка. Эти значения будут использованы позже при валидации алгоритма, разработанного с использованием нашей системы, на основе сравнения реального времени выполнения и расчетного времени, сообщенного моделью.

5.1 Архитектуры, использованные в экспериментальном анализе

Для описанных экспериментов особенно важны иерархические уровни рассматриваемых архитектур. Основные цели экспериментального анализа состоят в том, чтобы проверить, соответствуют ли вычисленные значения параметров Multi-BSP теоретическим значениям.

Для экспериментального анализа были выбрано три реальных многоядерных инфраструктуры с достаточно большим числом ядер и интересными уровнями кэша.

- Первым образцом является dell32, архитектура которого показана на рис. 6 (диаграмма получена путем применения hwloc). В dell32 имеются четыре процессора AMD Opteron 6128 Magny-Cours всего с 32 ядрами, 64 Гб основной памяти и два уровня иерархии.

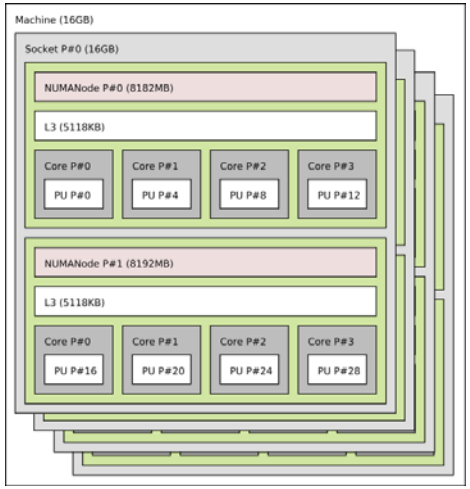

Рис. 6. Вывод hwloc, описывающий топологию многоядерного компьютера dell32
Fig. 6. hwloc output describing the topology of the dell32 multicore computer

- Образец #2 – jolly, архитектура которого показана на рис. 7 (диаграмма получена путем применения hwloc). В jolly имеются четыре процессора AMD Opteron 6272 Interlagos процессоры всего с 64 ядрами, 128 Гб основной памяти и три уровня иерархии.

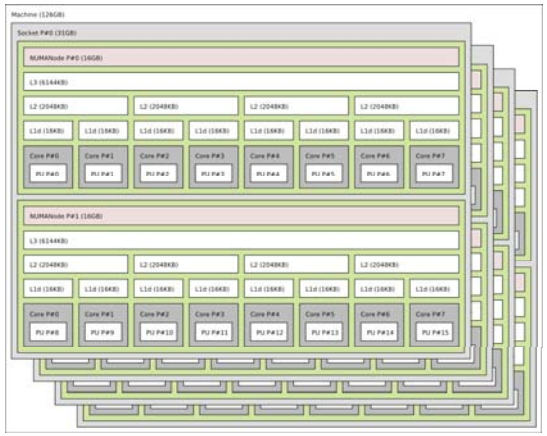

Рис. 7. Вывод hwloc, описывающий топологию многоядерного компьютера jolly
Fig. 7. hwloc output describing the topology of the jolly multicore computer

- Образец #3: узел XeonPhi в высокопроизводительной вычислительной платформе Cluster FING Республиканского университета [6]. У Xeon Phi 60 ядер, 8 Гб основной памяти, кэш L2 – 512 Кб и кэш L1 – 32 Кб. У каждого ядра имеются четыре процессорных блока для гиперпотоковой обработки, в результате чего общее число физических потоков составляет 240. Архитектура узла XeonPhi представлена на рис. 8

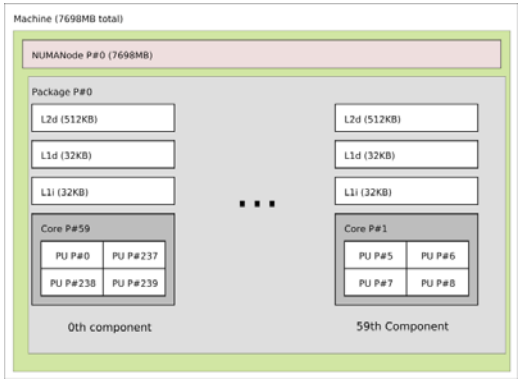

Рис. 8. Вывод hwloc, описывающий топологию сопроцессора XeonPhi
Fig. 8. hwloc output describing the topology of the XeonPhi co-processor

Для каждой целевой архитектуры на первом этапе необходимо определить соответствующие экземпляры в модели Multi-BSP. Для лучшего понимания состава Multi-BSP дальнейшее описание продвигается шаг за шагом в процессе построения спецификации.

Процедура создания спецификации для компьютера dell32 продвигается с нижнего уровня (ядра) к верхним уровням и создает кортежи компонентов, совместно использующих пространство памяти. Первый кортеж состоит из одного ядра на уровне 0. Это ядро не разделяет какую-либо память с каким-либо другим компонентом, поэтому объем его совместно используемой памяти равен 0, и оба параметра g и L равны нулю по определению: $\text{tuple}_0 = <p_0 = 1, g_0 = 0, L_0 = 0, m_0 = 0>$. Что касается следующего уровня, четыре базовых компонента уровня 0 совместно используют кэш-память L3 размером 5 МБ, образуя новый компонент Multi-BSP уровня 1. Этот новый компонент формально описывается кортежем $\text{tuple}_1 = <p_1 = 4, g_1, L_1, m_1 = 5$ МБ>. Наконец, все восемь компонентов уровня 1 совместно используют оперативную память объемом 64 ГБ, образуя следующий и последний уровень 2 в спецификации Multi-BSP. Этот формально описывается как $\text{tuple}_2 = <p_2 = 8, g_2, L_2, m_2 = 64$ ГБ>.

Заключительный шаг состоит в соединение всех кортежей в последовательность для получения полной спецификации машины Multi- BSP с отбрасыванием нулевого уровня, так как значения g_0 и L_0 известны по определению. Тогда архитектура образца #1 описывается соотношением (16).

$$M_1 = [< p_1 = 4, g_1, L_1, m_1 = 5\ Mb >, < p_2 = 8, g_2, L_2, m_2 = 64\ Gb >] \quad (16)$$

С использованием той же процедуры создается спецификация Multi-BSP для экземпляра #2. Опять же, уровень 0 описывается как $\text{tuple}_0 = <p_0 = 1, g_0 = 0, L_0 = 0, m_0 = 0>$. Уровень 0 имеет одинаковую спецификацию на всех машинах за исключением ядер, в которых используется технология гиперпоточности (в этом случае требуется дополнительный уровень для спецификации физических потоков).

Далее, имеются два компонента, разделяющие кэш L2 размером в 2 Мб. Первый уровень описывается кортежем $\text{tuple}_1 = <p_1 = 2, g_1, L_1, m_1 = 2$ МБ>. Компоненты первого уровня совместно используют четыре кэш-памяти L3 размером в 6 MB, образуя второй уровень, который описывается кортежем $\text{tuple}_2 = <p_2 = 4, g_2, L_2, m_2 = 6$ MB>. На последнем уровне группируются восемь компонентов второго уровня. Они разделяют основную память объемом в 128 GB. Третий уровень описывается кортежем $\text{tuple}_3 = <p_3 = 8, g_3, L_3, m_3 = 128$ GB>.

Подобно спецификации для dell32, окончательной спецификацией экземпляра MultiBSP для jolly является соотношение (17).

$$M_1 = [< p_1 = 2, g_1, L_1, m_1 = 2\ Mb >, < p_2 = 4, g_2, L_2, m_2 = 6\ Mb >, < p_3 = 8, g_3, L_3, m_3 = 128\ Gb >] \quad (17)$$

Третьей архитектурой является сопроцессор XeonPhi. С применением той же процедуры на нижнем уровне идентифицируются процессорные блоки, и они включаются в уровень 0. Как и в других архитектурах, процессорные блоки вообще не разделяют память, и определением этого уровня является $\text{tuple}_0 = <p_0 = 1, g_0 = 0, L_0 = 0, m_0 = 0>$. Затем, четыре компонента нулевого уровня разделяют кэш-память L2размером в 512 Kb. Так что первый уровень описывается как $\text{tuple}_1 = <p_1 = 4, g_1, L_1, m_1 = 512\ \text{Kb}>$. Наконец, как показывает рис. 8, последнему уровню соответствует $\text{tuple}_2 = <p_2 = 60, g_2, L_2, m_2 = 7698\ \text{Mb}>$.

Вышеупомянутые экземпляры модели Multi-BSP применяются в этой статье для прогнозирования времени работы алгоритма Multi-BSP, выполняемого на каждом компьютере. Параметры g_i и L_i в каждом кортеже должны быть предварительно рассчитаны с использованием процедуры эталонного тестирования, описанной в предыдущем разделе. В следующем подразделе приводятся значения g и L, полученные для всех архитектур на каждом уровне.

5.2. Результаты производительности исследованных архитектур

Эксперименты с производительностью ориентированы на определение времени выполнения простого алгоритма, разработанного и реализованного с использованием предложенной системы поддержки Multi-BSP. Таким образом, в этом подразделе приводится время выполнения h-коммуникаций на каждом уровне с увеличением числа h в соответствии с правилами функции *coreBenchmark*.

Полученные результаты представлены на рис. 9-11. Рис. 9 показывает h-коммуникации на каждом уровне для dell32 (уровни один и два). Рис. 10 демонстрирует аналогичные результаты для jolly (первый, второй и третий уровни). Рис. 11 показывает h-коммуникации на каждом уровне для Xeon Phi.

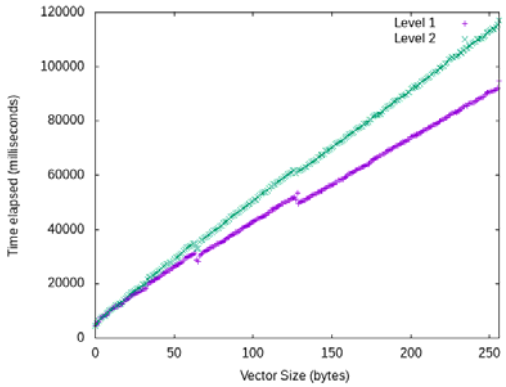

Рис. 9. Время h-коммуникаций в dell32
Fig. 9. Time for h-communications in dell32

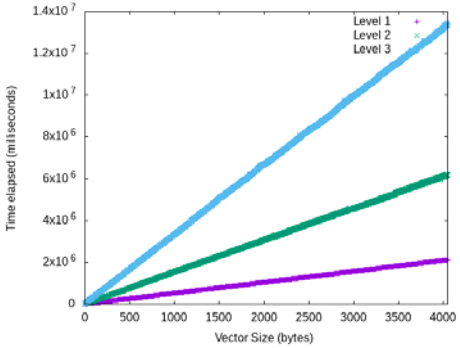

Рис. 10. Время h-коммуникаций в jolly
Fig. 10. Time for h-communications in jolly

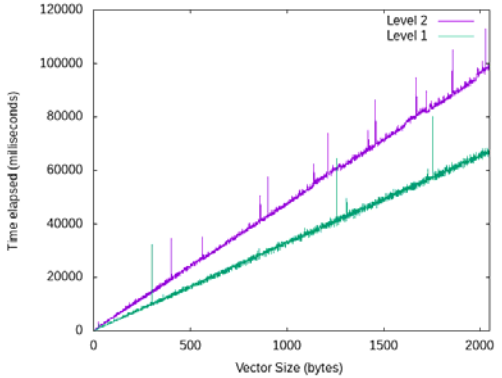

Рис. 11. Время h-коммуникаций в Xeon Phi
Fig. 11. Time for h-communications in Xeon Phi

В dell32 все коммуникации уровня 1 выполняются через общую память (кэш L3), поэтому они в два раза быстрее, чем на уровне 2, в котором используется основная память. В jolly коммуникации на уровне 1 выполняются через кеш L2, поэтому они в три раза быстрее, чем на уровне 2, где для этого используется кэш L3. В свою очередь, коммуникации на уровне 2 в 1,5 раза быстрее, чем на уровне 3, которые выполняются путем доступа к основной памяти.

Наконец, для определения значений g_i и L_i в h-коммуникациях для каждого уровня применяется метод наименьших квадратов. Окончательные значения для dell32, jolly и Xeon Phi представлены в табл. 1.

Табл. 1. Результирующие значения параметров g и L по уровням для dell32, jolly и Xeon Phi
Table 1. Results of g and L parameters per level for dell32, jolly and Xeon Phi

	dell 32		***jolly***			***Xeon phi***	
Level	***2***	***1***	***3***	***2***	***1***	***2***	***1***
g	977.5	334.9	1315.9	549.9	105.3	2470.1	1947.9
L	15550.2	7792.9	16184.4	7157.9	498.2	1380955.3	1322578.2

5.3 Анализ примера со скалярным произведением

Для проверки результатов, описанных в предыдущем подразделе, был проведен эксперимент с использованием алгоритма вычисления скалярного произведения, реализованного на основе системы поддержки MultiBSP. Процесс проверки включает в себя следующие этапы (применяются для разных размеров вектора):

1) оценить объем коммуникаций и синхронизации на каждом уровне, используя аппаратные счетчики;
2) вычислить значения параметров gi и Li, используя разработанный бенчмарк;
3) вычислить время исполнения алгоритма, используя теоретическую оценочную модель для мульти- BSP в виде, представленном в [10];
4) выполнить алгоритм вычисления скалярного произведения;
5) сравнить время выполнения алгоритма скалярного произведения с теоретическим предсказанным временем.

На рис. 12-14 показано сравнение теоретически оцененных затрат с реальными затратами на коммуникации и синхронизацию при выполнении алгоритма скалярного произведения, реализованного на основе системы поддержки MultiBSP. На рисунках показано время выполнения алгоритма скалярного произведения с использованием предложенного механизма MBSP в сравнении с временем, рассчитанным на основе теоретической модели, с учетом возрастающего размера входных данных.

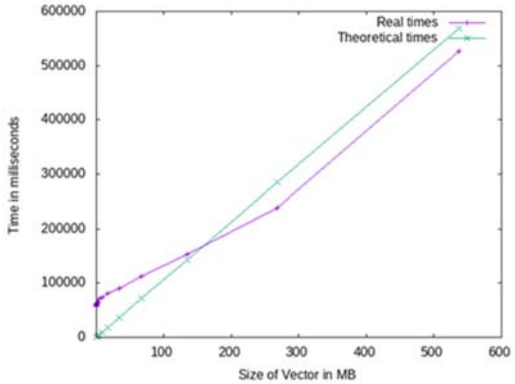

Рис. 12. Сравнение теоретически оцененных и реальных затрат для компьютера dell32
Fig. 12. Theoretical vs real cost for the dell32 computer

Результаты на рис. 12-14 показывают точность предсказанного времени по сравнению с реальным временем. Относительные погрешности составляют от 0% до 7%. В dell32 средняя ошибка составляет 6%, а максимальная ошибка – 9%. В jolly средняя ошибка составляет 7%, а максимальная ошибка прогнозируемого времени – 17%. Наилучшие результаты были получены для Xeon Phi, для которого средняя ошибка составляет всего 2%, а максимальная ошибка – 5%. Эти результаты показывают, что теоретическая оценка времени выполнения алгоритма скалярного произведения является достаточно точной по отношению к реальному времени для всех исследованных архитектур.

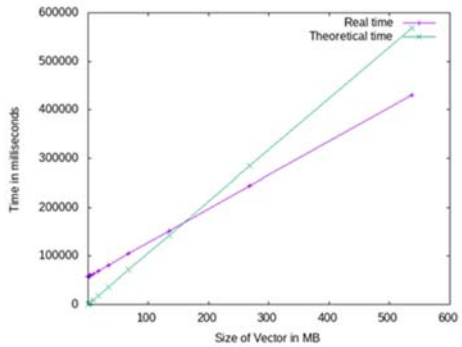

Рис. 13. Сравнение теоретически оцененных и реальных затрат для компьютера jolly
Fig. 12. Theoretical vs real cost for the dell32 computer

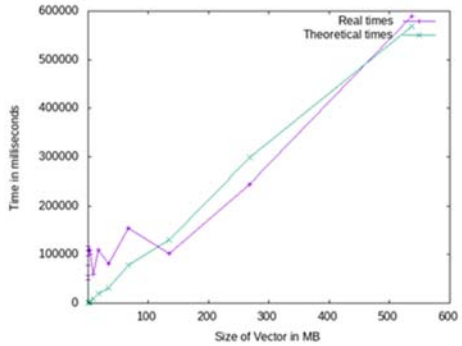

Рис. 14. Сравнение теоретически оцененных и реальных затрат для сопроцессора Xeon Phi
Fig. 14. Theoretical vs real cost for the Xeon Phi co-processor

6. Заключение и планы на будущее

В этой статье предложен упрощенный подход к разработке и реализации алгоритмов с использованием модели MultiBSP.

Предложенный подход включает методологию для автоматического обнаружения аппаратных особенностей данного компьютера и систему поддержки для разработки и реализации параллельных алгоритмов на основе обобщенной процедуры спецификации. Следуя этому подходу, при программировании не нужно сосредотачиваться на конкретных деталях реализации и эталонного тестирования MultiBSP, которое включены в предлагаемую систему. Программист стимулируется к разработке алгоритмов MultiBSP с использованием общей спецификации, основанной на рекурсивной стратегии «разделяй и властвуй», действующей над компонентами архитектуры (то есть над ядрами, кэшем и оперативной памятью).

Также представлена реализация бенчмарка MultiBSP для выявления характеристик используемых архитектур. Этот бенчмарк применяется в нашей системе, в который также используется процесс обнаружения для выполнения алгоритмов MultiBSP, позволяющий скрыть от программиста все детали разделения данных и закрепления потоков.

Валидация выполненных реализаций была произведена на трех современных высокопроизводительных архитектурах. Был построен и исследован частный случай

использования разработанной системы для решения декомпозируемой задачи – алгоритм вычисления скалярного произведения векторов.

Этот примерный алгоритм использовался для сравнения теоретического времени выполнения, оцененного с использованием модели стоимости MultiBSP, и реального времени выполнения реализации алгоритма скалярного произведения на основе разработанной системы. Результаты теоретической оценки оказались достаточно точными: средняя относительная погрешность составила от 2% до 7%. Наилучшие результаты были получены для Xeon Phi, для которого средняя ошибка составила всего 2%, а максимальная ошибка – 5%.

Предложенная методология обеспечивает основу для разработки пригодного для практического использования фреймворка, включающего набор инструментов для разработки, реализации и выполнения алгоритмов MultiBSP, а также точного прогнозирования времени их выполнения.

Основные направления будущих исследований связаны с расширением анализа предлагаемой методологии, например, путем изучения возможностей системы поддержки MultiBSP с использованием новых, более сложных алгоритмов. Систему можно расширить для решения недекомпозируемых задач, используя преимущества его модульной организации и учитывая знания о задачах, поставляемые пользователями (т.е. система может автоматически определять соответствие потоков, обеспечивать параллельное выполнение и локальность данных для функции разделения, задаваемой пользователем). Процесс обнаружения параметров аппаратного обеспечения может быть расширен дополнительными уровнями, включая обнаружение характеристик сети с использованием специальных программных библиотек и определение соответствия и локальности данных с учетом скорости и пропускной способности сети.

Список литературы / References

[1] Alaniz M., Nesmachnow S., Goglin B., Iturriaga S., Gil Costa V., Printista M. MBSPDiscover: An automatic benchmark for MultiBSP performance analysis. Communications in Computer and Information Science; vol. 485, 2014, pp, 158–172.

[2] Bisseling R. Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, 2004, 334 p.

[3] Bonorden O., Juurlink B., von Otte I., Rieping I. The Paderborn University BSP (PUB) Library. Parallel Computing, vol. 29, no. 2, 2003, pp. 187–207.

[4] Broquedis F., Clet-Ortega J., Moreaud S., Furmento N., Goglin B., Mercier G., Thibault S., Namyst R. Hwloc: A generic framework for managing hardware affinities in HPC applications. In Proc. of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 180–186.

[5] Hill J., McColl B., Stefanescu D., Goudreau M., Lang K., Rao S., Suel T., Tsantilas T., Bisseling R. BSPlib: The BSP programming library. Parallel Computing, vol. 24, no. 14, 1998, pp. 1947–1980.

[6] Nesmachnow S. Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. Revista de la Asociación de Ingenieros del Uruguay, vol. 61, no. 1, 2010, pp. 12-15 (In Spanish).

[7] Savadi A., Deldari H. Measurement latency parameters of the MultiBSP model: A multicore benchmarking approach. Journal of Supercomputing, vol. 67, no. 2, 2014, pp. 565–584.

[8] Soudris D., Jantsch A. (eds.). Scalable Multi-core Architectures: Design Methodologies and Tools. Springer Publishing Company, 2011, 223 p.

[9] Valiant L. A bridging model for parallel computation. Communications of the ACM, vol. 33, no. 8, 1990, pp. 103–111.

[10] Valiant, L. A bridging model for multi-core computing. Journal of Computing and System Sciences, vol. 77, no. 1, 2011, pp. 154–166.

[11] Yzelman, A. Fast sparse matrix-vector multiplication by partitioning and reordering. Ph.D. thesis, Utrecht University, Utrecht, the Netherlands, 2011, 136 p.

[12] Yzelman A., Bisseling R., Roose D., and Meerbergen K. MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming. International Journal of Parallel Programming, vol. 42, no. 4, 2014, pp. 619-642.

Информация об авторах / Information about authors

Марсело Аланис работает исследователем Вычислительном центре Института компьютерных наук Инженерного факультета Республиканского университета.

Marcelo Alaniz is a researcher at the Numerical Center (CeCal) at Computer Science Institute, Engineering Faculty.

Серджо Энрике НЕСМАЧНОВ КАНОВАС обладает степенью PhD в области компьютерных наук, полученной в Республиканском университете, Уругвай. В настоящее время он занимает должность профессора в Вычислительном центре Института компьютерных наук Инженерного факультета Республиканского университета. Основные научные интересы: параллельные и распределенные вычисления, научные вычисления, эволюционные алгоритмы и метаэвристика, а также численные методы.

Sergio Enrique NESMACHNOW CÁNOVAS has a Ph.D. degree in Computer Science from Universidad de la República, Uruguay. He currently holds an Aggregate Professor position in the Numerical Center (CeCal) at Computer Science Institute, Engineering Faculty. His main research interests are parallel and distributed computing, scientific computing, evolutionary algorithms and metaheuristics, and numerical methods.