УДК 519.685

СТЕНД ДЛЯ ОТЛАДКИ И ТЕСТИРОВАНИЯ КАЧЕСТВА РАБОТЫ ЛОКАЛЬНЫХ СИСТЕМНЫХ РАСПРЕДЕЛЕННЫХ АЛГОРИТМОВ ДИНАМИЧЕСКОЙ БАЛАНСИРОВКИ НАГРУЗКИ[1]

В.А. Перепелкин, И.И. Сумбатянц

При параллельной реализации итерационных численных методов на регулярных сетках возникает необходимость в статической или динамической балансировке вычислительной нагрузки. Для исследования того или иного алгоритма балансировки нагрузки важно проводить его разносторонее тестирование на множестве различных задач рассматриваемого класса с различными конфигурациями вычислителя и различными наборами входных данных задач. В статье представлен тестовый стенд, предназначенный для автоматизации проведения такого тестирования. Он позволяет описать прикладную задачу и подключить реализацию алгоритма статической или динамической балансировки вычислительной нагрузки для последующего тестирования на мультикомпьютере. На выходе стенд предоставляет информацию о том, как происходило исполнение итерационного сеточного метода с точки зрения баланса вычислительной нагрузки. Приведен пример использования стенда для исследования диффузионного алгоритма динамической балансировки нагрузки на процессоры мультикомпьютера.

Ключевые слова: динамическая балансировка нагрузки, большие численные модели, автоматизация тестирования.

Введение

При реализации больших численных моделей на суперкомпьютерах, в частности, итерационных методов на регулярных сетках, встает проблема обеспечения равномерной загрузки во времени вычислительных узлов суперкомпьютера. Для решения этой проблемы используют алгоритмы статической и/или динамической балансировки вычислительной нагрузки. Эффективность таких алгоритмов (в смысле способности обеспечивать равномерность загрузки процессоров) часто сложно оценить теоретически. В значительной степени это обусловлено тем, что эффективность балансировки может существенно зависеть от конфигурации вычислителя и входных данных прикладной программы. Поэтому важную роль в исследовании алгоритмов балансировки нагрузки играет тестирование их эффективности. Такое тестирование должно охватывать широкий спектр ситуаций, чтобы получить представление о работе алгоритма балансировки в целом. В частности, при тестировании должны варьироваться такие параметры, как классы прикладных задач, входные данные задачи, размер задачи, количество вычислительных узлов, фоновая нагрузка на сеть, параметры самого алгоритма балансировки нагрузки (если имеются), и т.п.

Для автоматизации проведения таких тестов целесообразно создание отладочного тестового стенда (ОТС). ОТС — это программа, принимающая на вход реализацию некоторого алгоритма балансировки нагрузки и выполняющая серию тестов с различными

---

[1] Статья рекомендована к публикации программным комитетом Международной научной конференции "Параллельные вычислительные технологии – 2015"

параметрами. В процессе тестирования производится реальное (не имитационное) исполнение задачи на мультикомпьютере. Результаты тестов позволяют судить о том, каковы качественные и количественные характеристики эффективности исследуемого алгоритма балансировки нагрузки на процессоры. Вследствие того, что в различных ситуациях требования к алгоритмам балансировки вычислительной нагрузки различаются, ОТС может быть разработан только для некоторой ограниченной предметной области. В настоящей работе предлагается ОТС, разработанный для исследования алгоритмов балансировки нагрузки на процессоры для итерационных методов на регулярных сетках, включая метод частиц-в-ячейках [3]. Так как в настоящей работе речь идет о численных моделях, реализуемых на суперкомпьютере, то ОТС должен быть ориентирован на локальные масштабируемые алгоритмы как прикладной задачи, так и балансировки нагрузки на процессоры.

В разделе 1 представлена методика оценки алгоритмов балансировки нагрузкию В разделе 2 описывается предлагаемый тестовый стенд, в разделе 3 представлена его реализация, а в разделе 4 приводится пример использования стенда на практике.

1. Методика оценки алгоритмов балансировки нагрузки

Для оценки и анализа работы балансировщика ОТС собирает данные о процессе исполнения численного метода. Такие данные должны быть достаточно полными и полезными для анализа. Так как стенд нацелен на практическое использование для определенного класса задач, то был исследован ряд статей, в которых были представлены алгоритмы статической и динамической балансировки нагрузки для характерного примера из целевого класса задач: итерационных сеточных методов и моделирования физических процессов методом частиц-в-ячейках.

Так было определено, что для анализа и тестирования алгоритмов балансировки нагрузки авторы использовали следующие данные:

- Общее время исполнения и время моделирования [2, 1, 4]
- Развертка максимума и минимума количества частиц на всех вычислительных узлах по времени моделирования [1]
- Максимальное количество частиц отображенных на один вычислительный узел [2]
- Развертка распределения вычислительной нагрузки по времени моделирования [6]
- Время исполнения итерации на каждом узле [6]

Помимо этого существенными представляются такие показатели как:

- Общее количество балансировок
- Развертка нагрузки на коммуникационную сеть по времени

2. Структура ОТС

В соответствии с поставленной задачей, ОТС должен обеспечивать тестирование заданного алгоритма балансировки нагрузки. Соответственно, возникает потребность в унификации интерфейса модуля балансировки нагрузки (балансировщика). Кроме того, поведение прикладных программ с точки зрения баланса нагрузки на процессоры может быть очень многообразным. Как следствие, целесообразно не вкладывать все возможные ситуации в ОТС, а предоставить возможность закладывать в него различные прикладные программы. Для этого в настоящей работе была предложена модель вычислений, в которой может быть представлен прикладной алгоритм из заданного класса.

2.1. Модель балансировщика

Балансировщик — это реализация локального распределенного алгоритма балансировки нагрузки на вычислительные узлы мультикомпьютера. Балансировщик должен устранять дисбаланс нагрузки и не допускать ситуаций, когда исполнение не может быть продолжено из-за отсутствия свободных ресурсов на узле. В силу своей распределенности балансировщик может опираться на каждом вычислительном узле на информацию о загруженности текущего узла и о загруженности узлов, находящихся в некоторой окрестности от него (в смысле сетевой инфраструктуры). Кроме того, балансировщик может иметь параметры (конфигурацию), определяемые статически и влияющие на его поведение (например, порог дисбаланса для диффузионных алгоритмов балансировки нагрузки).

Для ОТС балансировщик — это модуль, который содержит реализацию алгоритма балансировки и некоторый предикат, который позволяет определить, необходимо ли производить балансировку. В определенные моменты времени ОТС проверяет необходимость в балансировке, и после положительного результата инициирует процесс балансировки в некоторой окрестности текущего вычислительного узла. Значение предиката проверяется на каждом вычислительном узле и в случае необходимости балансировка асинхронно инициируется на одном из узлов. При этом возможно одновременное выполнение нескольких балансировок в разных частях мультикомпьютера.

2.2. Модель вычислений прикладной программы

Прикладной алгоритм представляется в виде графа $\langle F, N, op, R\rangle$, где F — множество вершин графа (далее — фрагментов), N — множество ребер, соединяющих вершины из множества F (отношение соседства), op — оператор, который применяется к каждому элементу F, R — оператор редукции, который применяется ко всем элементам F. Процесс вычислений происходит в дискретном времени $T = \{t_1 \dots t_n\}$, и в каждый момент времени t_i элемент $f \in F$ имеет состояние (значение) f_{t_i}. Каждое следующее состояние фрагмента $f_{t_{i+1}}$ зависит от его текущего состояния f_{t_i}, от текущих состояний соседних фрагментов $\{fn_{t_i} : (fn \in F) \& ((fn, f) \in F)\}$ и от редукционных данных времени t_i. Тогда в общем виде процесс вычислений выглядит следующим образом:

$$\forall f \in F \left(f_{t_i} = op\left(f_{t_{i-1}}, \{fn_{t_{i-1}} : (fn \in F) \& \big((fn, f) \in N\big)\}, R(t_{i-1}) \right) \right)$$

Для примера рассмотрим метод частиц-в-ячейках\cite{at, pic}. В этом методе имеется пространство моделирования, в котором движутся частицы, взаимодействуя с полем (полями). Поля дискретизуются на статичной регулярной сетке. Применение метода пространственной декомпозиции приводит к разделению пространства моделирования на домены, каждый из которых содержит часть сеточных значений и значения частиц, принадлежащих этому домену. Распределение частиц по доменам изменяется во времени. В этом примере фрагментом будет домен с его сеточными значениями и значениями частиц, оператор скрывает в себе вычисление новых координат частиц и новых сеточных значений, а редукционный оператор применяется для определения таких величин, как суммарная энергия системы.

2.3. Реализация вычислений на мультикомпьютере

Тестовый стенд реализует описанный прикладной вычислительный процесс на мультикомпьютере. Каждый узел мультикомпьютера может обмениваться сообщениями с ограниченным множеством других узлов. Конкретная топология соединений задается стендом. Далее будем называть такое множество *локальной окрестностью узла*, а узлы из этого множества — *соседними узлами*.

В ходе исполнения ОТС может предоставить фрагменту данные соседних фрагментов, передать данные от одного фрагмента другому и произвести редукцию данных между всеми фрагментами.

Начальное отображение фрагментов на узлы мультикомпьютера можно производить несколькими способами:

– Статически определять место создания фрагментов до начала исполнения
– Создавать фрагменты на нескольких выделенных узлах после начала исполнения, для построения начального отображения с помощью балансировщика

По требованию балансировщика система может перемещать фрагменты на соседние узлы мультикомпьютера в рамках локальной окрестности и создавать распределенную реализацию фрагмента, если на узле нет достаточного количества ресурсов для его хранения или обработки (в случае, если эта возможность поддерживается со стороны прикладной задачи).

3. Реализация ОТС

Предложенный ОТС был реализован в виде программного прототипа. Рассмотрим его. ОТС содержит интерфейс для подключения балансировщика, интерфейс для подключения модуля, реализующего прикладной алгоритм и систему, обеспечивающую исполнение прикладного алгоритма. Таким образом, на вход ОТС принимает реализацию задачи и реализацию балансировщика.

3.1. Реализация задачи

В соответствии с моделью вычислений, прикладная задача — это множество фрагментов, отношение соседства на множестве фрагментов и операторы шага итерации и редукции. ОТС предоставляет интерфейсы для описания множества фрагментов, которые инкапсулируют в себе эти операторы и отношение соседства. ОТС и интерфейсы были реализованы на языке C++. Это язык был выбран по двум причинам: существует несколько реализаций MPI для C++, модель вычислений достаточно хорошо укладывается в объектно-ориентированную парадигму программирования. Таким образом, со стороны пользователя, прикладная задача — это реализация соответствующего интерфейса ОТС на языке C++. Таких реализаций для описания задачи может быть несколько, с разными поведением и функциями.

Пользователь ОТС может определять любые данные в реализации фрагмента, которые будут определять его состояние. Интерфейсы содержат управляемый пользователем внутренний счетчик, который определяет модельный момент времени (шаг модельного времени или номер итерации). Для упрощения описания процесса исполнения помимо счетчика модельного времени был введен счетчик прогресса исполенния на данном шаге итерации (подшаг итерации). Информация об отношении соседства на множестве фрагментов хранится распределенно: каждый объект из множества фрагментов

содержит информацию о его локальной окрестности (локальная окрестность задается пользователем в момент создания экземпляра фрагмента). Реализации операторов шага итерации и редукции — это реализации методов интерфейсов фрагмента. На рис. 1 представлена существенная часть исходного кода интерфейса для описания множества фрагментов.

```
class Fragment {
private:
    ID _vid; ///< ID of fragment
    ts::NodeID _vnodeID = 0; ///< Logic fragment location
    std::map<ID, ts::NodeID> _vneighboursLocation; ///< Fragment's
        /// neighbours locations

public:
    // General
    Fragment(ID id);
    virtual ~Fragment();
    ID id();
    void setNodeID(ts::NodeID);

    // Neighbours and their locations
    bool isNeighbour(const ID& id);
    void updateNeighbour(ID id, ts::NodeID node);
    void addNeighbour(ID id, ts::NodeID node);

    // Fragment steps defined by user
    virtual ReduceData* reduce() = 0;
    virtual ReduceData* reduce(ReduceData* data) = 0;
    virtual void reduceStep(ReduceData* data) = 0;
    virtual void runStep(std::vector<Fragment*> neighbours) = 0;

    // Flag setters
    void setEnd();
    void setUpdate();
    void setReduce();
    void setNeighbours(uint64_t iteration, uint64_t progress);

    // State setters
    void nextIteration();

    // State getters
    uint64_t iteration();
    uint64_t progress();
};
```

Рис. 1. Исходный кода интерфейса для описания множества фрагментов.

Реализация задачи в терминах стенда — это множество объектов (фрагментов) из реализованного множества фрагментов, которые передаются ОТС в качестве входа. Начальное распределение фрагментов по узлам мультикомпьютера может быть задано статически: в процессе порождения фрагментов на мультикомпьютера, либо для этой цели может быть использован балансировщик: фрагменты порождаются на нескольких выделенных узлах, и балансировщик во время порождения начинает распределять фрагменты по соседним узлам мультикомпьютера.

ОТС скрывает в себе такие операции как: применение оператора к фрагменту, поиск соседних фрагментов для применения оператора, редукция данных, миграция фрагмента.

3.2. Исполнение задачи

ОТС содержит множество фрагментов, итерационные шаги которых необходимо исполнять. Исполнение производится по подшагам. Управление ходом исполнения производится с помощью установки флагов: при необходимости в редукции данных; необходимость в соседних фрагментах для применения оператора; необходимость в обновлении состояния фрагмента для глобального использования; для указания того, что на следующем подшаге будет следующая итерация. Флаги согласованно устанавливаются фрагментами в реализованном операторе.

ОТС передает фрагменты на исполнение в соответствии с выставленными флагами. Если фрагменту для данного подшага не нужны значения (состояния) соседних фрагментов, то он сразу передается на исполнение. В противном случае сначала производится поиск соответствующих фрагментов на данном узле, если не все фрагменты найдены, то данный фрагмент пропускается, пока на узел не придут значения всех соседних фрагментов. Во многих задачах для применения оператора к фрагменту нет необходимости знать полное состояние соседних фрагментов (например, в методе частиц-в-ячейках из сеточных значений соседей необходимы смежные грани сетки), поэтому для оптимизации скорости работы ОТС была введена такая абстракция как \textit{частичная реализация фрагмента}, которая определяется пользователем. И именно частичная реализация фрагмента передается на узел с фрагментом, которому для применения оператора необходимы соседние фрагменты.

3.3. Балансировка

Балансировщик реализуется как внешная динамически подключаемая к ОТС библиотека. Соответственно, балансировщик может быть реализован на любом языке программирования, который позволяет собрать динамическую библиотеку. Реализация алгоритма балансировки работает в терминах нагрузки: на вход принимает количество вычислительной нагрузки на локальный узел и на соседние узлы, а на выходе предоставляет информацию о том, как нагрузка должна быть распределена. После этого ОТС сам решает какие именно фрагменты необходимо передать. Величина нагрузки определяется ОТС в единицах, зависящих от задачи (и определяемых пользователем при описании прикладной задачи).

ОТС периодически обращается к балансировщику для определения необходимости инициации балансировки на узле.

Миграция фрагмента с узла i на узел j производится следующим образом: оповещается узел j о начале миграции; оповещаются все узлы, на которых есть соседние фрагменты, о том, что фрагмент будет перемещен на узел j; на узел j передается фрагмент и все частичные реализации фрагментов, которые были переданы ранее на узел i.

3.4. Ограничения реализации

Реализованный программный прототип ОТС имеет следующие ограничения:

– Исполнение допустимо только на логической топологии «кольцо»
– Нет поддержки распределенных реализаций фрагментов

4. Пример использования

В качестве прикладной задачи для практического испытания ОТС была выбрана задача расчета изображения методом трассировки лучей. Эта задача была выбрана в качестве примера в связи с тем, что при ее параллельной реализации методом пространственной декомпозиции балансировка нагрузки на процессоры является сложной задачей, как правило требующей динамического решения.

Трассировка лучей — технология построения изображения трёхмерных моделей, при которых отслеживается обратная траектория распространения луча \cite{rt}. Характерной особенностью этой задачи является различная вычислительная сложность обработки разных пикселей, которая определяется тем, сколько раз преломится/отразится луч, выпущенный через него. Это приводит к необходимости динамической балансировки нагрузки на процессоры.

Декомпозиция задачи была выполнена по пространству на блоки, каждый из которых вычисляется независимо. После обработки блоков полученные части изображения масштабируются и собираются в отдельном фрагменте.

С точки зрения модели вычислений каждый фрагмент содержит описание сцены, координаты камеры и экрана, границы экрана, до которых необходимо производить расчет. Процесс исполнения состоит из 3 шагов: обсчет пикселей части изображения, уменьшение полученной части изображения, передача полученного изображения специальному фрагменту. В качестве оценки вычислительной нагрузки каждого фрагмента использовалось количество пикселей, которые необходимо обработать. На каждом вычислителе порождается одинаковое количество фрагментов, после чего инициируется исполнение.

4.1. Балансировка нагрузки

Для задачи трассировки лучей был реализован локальный распределенный алгоритм балансировки нагрузки, который можно описать следующей схемой:

1. *load* = нагрузка на текущем узле
2. Цикл по всем узлам *i* из 1-окрестности текущего узла:
    a. Если нагрузка на *i* (*nload*) меньше нагрузки *load* на 20% и более
        i. $diff = (load - nload) / 2$
        ii. Передать узлу *i* количество нагрузки *diff*
        iii. $load = load - diff$

Таким образом, балансировка нагрузки инициируется каждый раз, когда нагрузка на узле изменяется на 1/5 часть, с момента начала исполнения или с момента последней балансировки.

4.2. Результаты

Предложенный алгоритм балансировки нагрузки был реализован и протестирован на ОТС. С помощью трассировки лучей отрисовывалось изображение размером 5000×5000 пикселей, которое было декомпозировано на 100 фрагментов и отображено на 4 и 10 вычислительных узлов. Фрагменты были отображены на вычислительные узлы таким образом, чтобы возникал дисбаланс вычислительной нагрузки. Для удобства визуализации нагрузка была нормирована. Для сравнения были также получены резуль-

таты работы задачи без динамической балансировки. Для начала приведем результаты тестирования.

4.2.1. Время исполнения

По результатам измерения времени исполнения программы, приведенным в таблице, можно заметить, производительность увеличивается с увеличением количества узлов. Но балансировщик не дает значительного прироста производительности в данном случае. Для того, чтобы понять, почему это происходит, можно обратиться к другим данным, полученным в процессе исполнения.

Таблица.

Времена выполнения программы

Количество узлов	Без балансировки	С балансировкой
4	243 сек.	224 сек.
10	119 сек.	104 сек.

4.2.2. Развертка нагрузки на вычислительные узлы по времени

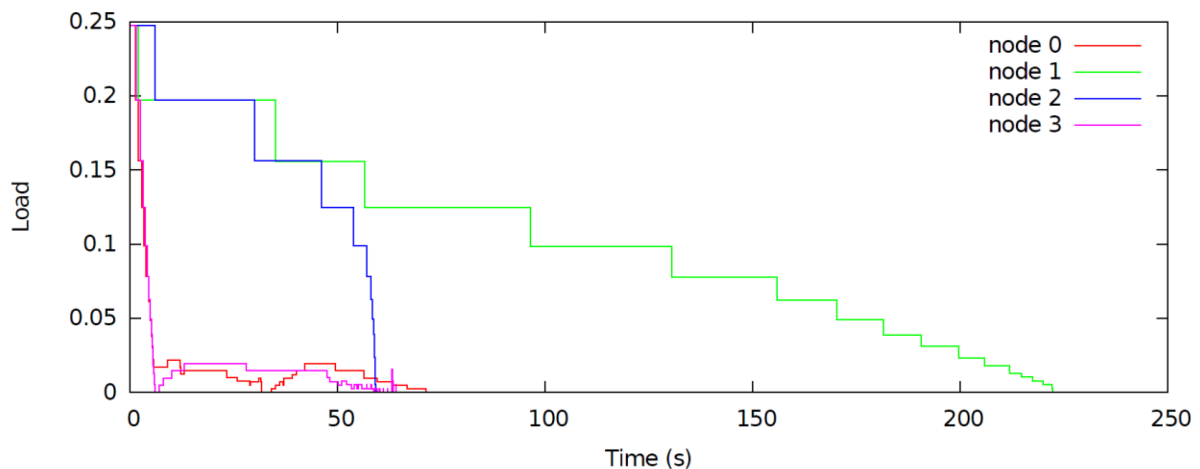

Рис. 2. Развертка нагрузки на вычислительные узлы по времени (4 узла)

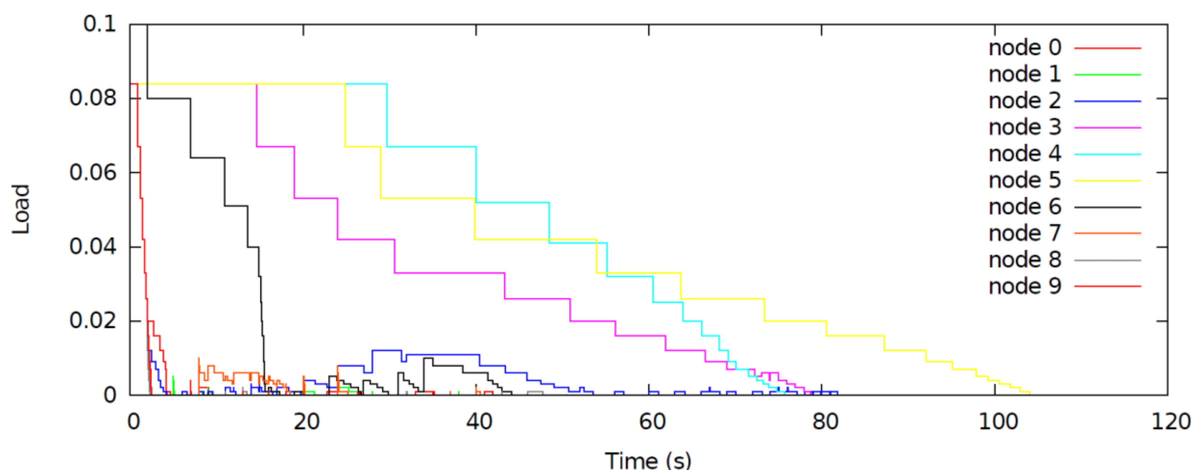

Рис. 3. Развертка нагрузки на вычислительные узлы по времени (10 узлов)

Как можно заметить из графиков (рис. 2 и 3), балансировщик действительно работает: нагрузка на некоторые узлы периодически увеличивается. Но основная проблема

— это порог дисбаланса: балансировщик срабатывает при изменении нагрузки на 1/5 часть от текущей. Ближе к концу вычислений нагрузка не изменяется на такую величину, но, тем не менее, на узле остается большое количество пикселей, на которые приходится больше всего отражений. Из-за этого балансировщик работает неэффективно. Таким образом, другой проблемой исследуемого алгоритма балансировки является то, что оценка степени загруженности узла по числу пикселей является грубой, если средний вычислительный вес пикселей на разных узлах существенно отличается.

Данный пример наглядно демонстрирует процесс использования ОТС для исследования свойств алгоритма балансировки нагрузки на заданной задаче.

Заключение

Предложен отладочный тестовый стенд, предназначенный для испытания различных характеристик алгоритмов балансировки нагрузки на процессоры. Стенд принимает на вход описание прикладной задачи на базе предложенной модели вычислений, а также реализацию алгоритма балансировки вычислительной нагрузки на базе предложенного интерфейса. Приведен пример исследования алгоритма динамической балансировки вычислительной нагрузки на процессоры в задаче построения изображения методом трассировки лучей.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта 14-01-31328 мол_а.

Литература

1. Ferraro, R.D. Dynamic load balancing for a 2D concurrent plasma PIC code / Robert D. Ferraro, Paulett C. Liewer, Viktor K. Decyk //Journal of computational physics. — 1993. — Vol. 109, N. 2. — P. 329–341.
2. Kraeva, M.A. Assembly technology for parallel realization of numerical models on MIMD-multicomputers. / M.A. Kraeva, V.E. Malyshkin // Future Generation Computer Systems. — 2001. — P. 755–765.
3. Kraeva, M.A. Implementation of PIC method on MIMD multicomputers with assembly technology / M.A. Kraeva, V.E. Malyshkin // High-Performance Computing and Networking. — 1997. — P. 541–549.
4. Nakashima, H. OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations / Hiroshi Nakashima, Yohei Miyake, Hideyuki Usui, Yoshiharu Omura //Proceedings of the 23rd international conference on Supercomputing. — 2009. — P. 90–99.
5. Ploeg, A.J. Interactive Ray Tracing / A.J. van der Ploeg. // 2011. — P. 1–4.
6. Wolfheimer, F. A parallel 3D particle-in-cell code with dynamic load balancing / Felix Wolfheimer, Erion Gjonaj, Thomas Weiland //Journal of computational physics. — 1993. — Vol. 109, N. 2. — P.329–341.

Перепелкин Владислав Александрович, м.н.с. лаборатории Синтеза параллельных программ Института вычислительной математики и математической геофизики СО РАН (г. Новосибирск, Россия), perepelkin@ssd.sscc.ru.

Сумбатянц Илья Ильич, магистрант Новосибирского государственного университета (г. Новосибирск, Россия), ilya.sumb@gmail.com.

TEST BENCH FOR DISTRIBUTED DYNAMIC LOAD BALANCING ALGORITHMS WITH LOCAL COMMUNICATIONS

V.A. Perepelkin, Institute of Computational Mathematics and Mathematical Geophysics, Siberian Branch of Russian Academy of Sciences (Novosibirsk, Russia),
I.I Sumbatyants, National Research University of Novosibirsk (Novosibirsk, Russia)

Parallel implementation of iterative methods on regular meshes often requires static or dynamic load balancing. To study a load balancing algorithm it is important to perform versatile testing on a variety of application problems of given class, on different hardware configuration and input data sets. In the paper a software test bench is introduced. The purpose of the bench is to automate such testing. It allows to describe an application problem and to utilize user load balancing algorithm to perform tests on a multicomputer. The result of such testing is an information on the load algorithm's performance.

Keywords: dynamic load balancing, large-scale numerical modeling, performance testing automation.

References

1. Ferraro R.D., Paulett C.L., Viktor K.D.. Dynamic load balancing for a 2D concurrent plasma PIC code //Journal of computational physics, 1993. Vol. 109, N. 2. P. 329–341.
2. Kraeva M.A., Malyshkin V.E. Assembly technology for parallel realization of numerical models on MIMD-multicomputers // Future Generation Computer Systems, 2001. P. 755–765.
3. Kraeva M.A., Malyshkin V.E. Implementation of PIC method on MIMD multicomputers with assembly technology // High-Performance Computing and Networking, 1997. P. 541–549.
4. Nakashima H., Miyake Y., Usui H., Omura Y. OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations //Proceedings of the 23rd international conference on Supercomputing, 2009. P. 90–99.
5. Ploeg A.J. Interactive Ray Tracing, 2011. P. 1–4.
6. Wolfheimer F. Gjonaj E., Weiland T. A parallel 3D particle-in-cell code with dynamic load balancing //Journal of computational physics, 1993. Vol. 109, N. 2. P.329–341.