Стенд для отладки и тестирования качества работы локальных системных распределенных алгоритмов динамической балансировки нагрузки

В.А. Перепелкин[1] И.И. Сумбатянц[2]

Институт вычислительной математики и математической геофизики СО РАН[1],
Новосибирский Государственный Университет[2]

При параллельной реализации итерационных численных методов на сетках возникает необходимость в статической или динамической балансировке вычислительной нагрузки. Для исследования того или иного алгоритма балансировки нагрузки важно проводить разностороннее тестирование на множестве различных задач, рассматриваемого класса, с различными конфигурациями. Для автоматизации проведения такого тестирования в статье представлен тестовый стенд, который позволяет описать прикладную задачу и подключить реализацию алгоритма статической или динамической балансировки для последующего тестирования, а на выходе предоставляет информацию о том, как происходило исполнение итерационного сеточного метода.

1. Введение

При реализации больших численных моделей на суперкомпьютерах, в частности, итерационных сеточных методов, встает проблема обеспечения равномерной загрузки во времени вычислительных узлов. Для решения этой проблемы используют алгоритмы статической и/или динамической балансировки нагрузки на процессоры. Эффективность таких алгоритмов (в смысле способности обеспечивать баланс загрузки процессоров) часто сложно оценить теоретически. В том числе вследствие того, что эффективность может существенно зависеть от конфигурации вычислителя и входных данных прикладной программы. Поэтому важную роль играет тестирование эффективностии алгоритмов балансировки нагрузки на процессоры. Такое тестирование должно охватывать широкий спектр ситуаций, чтобы получить представление о работе алгоритма балансировки в целом. В частности, 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 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 сеточных методов, включая метод частиц-в-ячейках [2]. 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.

2. 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:

1. Общее время исполнения и время моделирования [1, 4, 5]
2. Развертка максимума и минимума количества частиц на всех вычислительных узлах по времени моделирования [4]
3. Максимальное количество частиц отображенных на один вычислительный узел [1]
4. Развертка распределения вычислительной нагрузки по времени моделирования [6]
5. Время исполнения итерации на каждом узле [6]

Помимо этого существенными представляются такие показатели как:

1. Общее количество балансировок
2. Развертка нагрузки на коммуникационную сеть по времени

3. 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.

3.1. Модель балансировщика

Балансировщик – это реализация локального распределенного алгоритма балансировки. Балансировщик должен устранять дисбаланс нагрузки и не допускать ситуаций, когда исполнение не может быть продолжено из-за отсутствия свободных ресурсов на узле. Балансировщик может опираться на информацию о нагрузке в локальной окресности и на входные параметры, изменяя которые, можно менять его поведение.

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.

3.2. Модель вычислений прикладной программы

Прикладной алгоритм представляется в виде графа $< F, N, op, R >$, где F – множество вершин графа (далее – фрагментов), N – множество ребер на множестве 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 $\{fn_{t_i} : (fn \in F) \wedge ((fn, f) \in N)\}$ и от редукционных данных времени t_i. Тогда в общем виде процесс вычислений выглядит следующим образом:

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

Для примера рассмотрим метод частиц-в-ячейках [1, 2]. 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.

3.3. 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:

1. Статически определять место создания фрагментов до начала исполнения
2. 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.

4. 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 приниимает реализацию задачи и реализацию балансировщика.

4.1. 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 множества фрагментов, которые будут определять состояние фрагмента. 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 фреймворка. Ниже 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 location

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

    // Neighbours and their location
    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();
};
```

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.

4.2. 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. ОТС передает фрагменты на исполнение в соответствии с выставленными флагами. Если фрагменту для данного подшага не нужны соседние фрагменты, то он сразу передается на исполнение. Если же есть необходимость в соседних фрагментах, то сначала производится поиск соответствующих фрагментов на данном узле, если не все фрагменты найдены, то данный фрагмент пропускается, пока на узел не придут все соседние фрагменты. Во многих задачах нет необходимости во всем фрагменте (например, в методе частиц-в-ячейках необходимы теневые грани ячеек), поэтому для оптимизации была введена такая абстракция как *частичная реализация фрагмента*, которая определяется пользователем. И именно частичная реализация фрагмента передается на узел с фрагментом, которому для применения оператора необходимы соседние фрагменты.

4.3. 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 сама решает какие именно фрагменты необходимо передать. Величина нагрузки определяется ОТС в единицах, зависящих от задачи (и определяемых пользователем при описании прикладной задачи).

Система тестирования 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.

4.4. Ограничения реализации

1. Исполнение токлько на логической топологии «кольцо»
2. Нет возможности создавать распределенную реализацию фрагмента

5. 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 [3]. 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 вычислется независимо. После чего 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 проиводить расчет. Процесс исполнения состоит из 3 шагов: обсчет точек части сцены, уменьшение полученной части изображения, передача полученного изображения специальному фрагменту. В качестве оценки ыычислительной нагрузки каждого фрагмента использовалось количество точек, которые необходимо обработать. На каждом вычислителе порождается одинаковое количество фрагментов, после чего инициируется исполнение.

5.1. 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 $load$
      i. diff = $(load - nload)/2$
      ii. Передать узлу i количество нагрузки diff
      iii. $load = load -$ diff

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

5.2. Результаты

Предложенный алгоритм балансировки был реализован и протестирован на ОТС. С помощью трассировки лучей отрисовывалось изображение размером 5000x5000 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 результаты работы задачи без динамической балансировки. Для начала приведем результаты тестирования.

5.2.1. Время исполнения

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

Как 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.

5.2.2. Развертка нагрузки на вычислительные узлы по времени

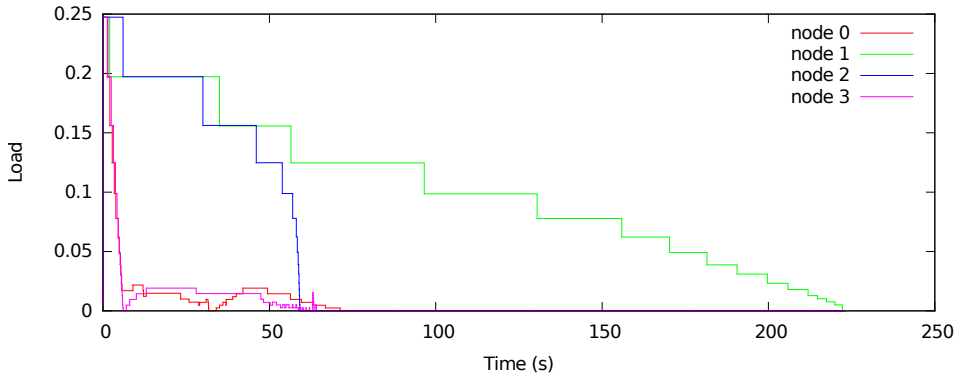

Рис. 1. Развертка нагрузки на вычислительные узлы по времени (4 узла)

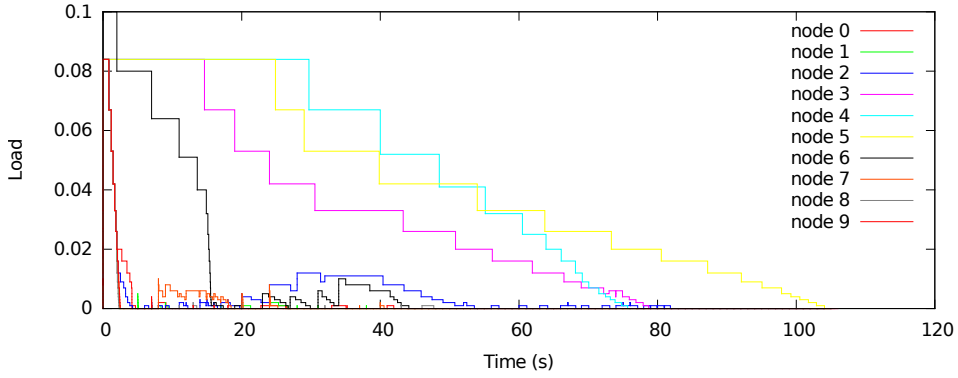

Рис. 2. Развертка нагрузки на вычислительные узлы по времени (10 узлов)

Как можно заметить из графиков, 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.

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

6. 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.

Литература

1. M.A. Kraeva, V.E. Malyshkin. Assembly technology for parallel realization of numerical models on MIMD-multicomputers // Future Generation Computer Systems, 2001, P. 755–765.

2. M.A. Kraeva, V.E. Malyshkin. Implementation of PIC method on MIMD multicomputers with assembly technology, //High-Performance Computing and Networking, 1997, P. 541–549

3. A.J. van der Ploeg. Interactive Ray Tracing, //2011 P. 1–4

4. Ferraro, Robert D and Liewer, Paulett C and Decyk, Viktor K. Dynamic load balancing for a 2D concurrent plasma PIC code, //Journal of computational physics. 1993. Vol. 109, N. 2. P. 329–341.

5. Nakashima, Hiroshi and Miyake, Yohei and Usui, Hideyuki and Omura, Yoshiharu. 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.

6. Wolfheimer, Felix and Gjonaj, Erion and Weiland, Thomas. A parallel 3D particle-in-cell code with dynamic load balancing. , //Journal of computational physics. 1993. Vol. 109, N. 2. P. 329–341.