Параллельное моделирование Монте-Карло на системах с распределённой памятью

Рубанов Л. И.

***Аннотация* — Рассматривается задача параллельного моделирования многочисленных итераций метода Монте-Карло на суперкомпьютерах с распределённой памятью. Требуется обеспечить эффективное использование вычислительной мощности кластера при различной трудоемкости итераций и/или производительности процессоров. Проанализированы известные методы распараллеливания циклов обработки и получены эмпирические оценки их эффективности для двух часто встречающихся видов неоднородности итераций.**

***Ключевые слова* — метод Монте-Карло, неоднородный цикл, параллельная обработка, распределённая память.**

I. Введение

Данная статья является дальнейшим развитием темы, ранее поднятой в работе [1].

Многие большие системы, представляющие интерес для практики, не могут быть исследованы аналитически из-за своей сложности. В таких ситуациях обычно прибегают к компьютерному моделированию на основе математических и/или алгоритмических моделей процессов, протекающих в интересующей системе, и их взаимосвязей. Для этого часто используются методы стохастической динамики, например, искусственный отжиг (аннилинг), метод статистических испытаний (Монте-Карло) и др. В результате моделирования устанавливаются количественные характеристики системы и выявляются взаимосвязи между ними. Сложность системы закономерно приводит к сложности модели, а требования достоверности и точности результатов дополнительно повышают вычислительную сложность моделирования. Как следствие этого, модели все чаще опираются на использование технологий параллельных вычислений [2, 3].

Сказанное в полной мере справедливо для внутриклеточных систем и фундаментальных процессов молекулярного уровня – транскрипции, трансляции, регуляции, эволюции, которые имеют большое значение для биологии и медицины. Автор непосредственно участвует в разработке и реализации ряда моделей таких систем и процессов [4–8] и проводит компьютерное моделирование с помощью этих моделей.

Для указанных моделей характерно, что хотя каждая отдельная траектория моделирования и может быть вычислена современным процессором достаточно быстро (обычно на два-три порядка быстрее реального времени протекания внутриклеточного процесса), достоверные результаты можно получить лишь после совместного анализа большого числа траекторий модели, порядка 10^4–10^8. Поэтому для моделирования необходимы параллельные вычислительные системы с сотнями и тысячами процессоров.

Мультипроцессорные системы с общей памятью и более чем 64 процессорными ядрами редки, что не позволяет воспользоваться удобной для параллельного программирования системой OpenMP [3, 9]. Поэтому будем ориентироваться главным образом на значительно более распространённые суперкомпьютеры с распределённой памятью (кластеры). Для таких систем параллельные программы чаще всего разрабатываются с использованием библиотек стандартного интерфейса передачи сообщений MPI [2, 3, 10].

Ключевой аспект параллельного моделирования методом Монте-Карло – сравнительная трудоёмкость итераций. Если она одинакова, т.е. цикл *однородный*, итерации можно выполнять синхронно, как это делается в систолических архитектурах. Нас же интересует случай существенно неоднородных циклов работы модели, когда длительность итерации (например, время вычисления одной траектории моделирования) меняется в широких пределах. Аналогичная ситуация складывается и при однородных вычислениях на кластере, состоящем из узлов с различной производительностью. Как будет показано, недоучет этого фактора может привести к катастрофическому снижению производительности моделирования.

II. Постановка задачи

Рассматриваемая схема моделирования в общем виде представляет собой многократно повторяемый цикл (или совокупность вложенных циклов), тело которого есть суперпозиция двух функций, $Merge(Work(p_n))$:

```
for (n = 0; n < N; n++) {
  w = Work(p[n]);
  Merge(w);
}
```

Здесь *Work* – функция без побочного эффекта, в которой сосредоточена основная вычислительная сложность итерации цикла (назовём её «функция обработки»), а *Merge* – вычислительно простая функция с побочным эффектом, приводящим к формированию результата всего цикла («функция слияния»). Функция

Статья получена 9 января 2014. Работа выполнена при поддержке гранта Министерства образования и науки РФ, соглашение № 8481.

Рубанов Л. И. – ведущий научный сотрудник Института проблем передачи информации им. А.А. Харкевича Российской академии наук, Москва (e-mail: rubanov@iitp.ru).

обработки вычисляет одиночную траекторию модели, а функция слияния осуществляет совместную обработку множества траекторий. Примерами такой обработки могут быть вычисление оценок значений интересующих характеристик системы или вероятностей конечных состояний моделируемого процесса, нахождение экстремальных значений в ходе траектории, построение гистограмм и т.п.

Функция *Work* может быть скалярной или векторной и в общем случае зависит от необязательного параметра (или вектора параметров) p_n, где n – индекс цикла (номер итерации). Для простоты записи мы ограничимся представленным выше скалярным случаем, полагая, что параметр и возвращаемое значение функции обработки имеют плавающий тип с двойной точностью.

Параметры функции обработки не носят характера входных данных, а служат лишь для указания нужной порции данных или модификации режима их обработки, задания граничных условий и т.п. (в этом смысле они не обязательны). Предполагается, что все необходимые входные данные для вычисления функции *Work* доступны внутри неё, например, содержатся в статическом массиве или генерируются с помощью псевдослучайного датчика. Таким образом, в данной постановке не предполагается хранить результаты каждой итерации цикла (т.е. траектории модели), а требуется получить лишь сводные данные (статистики) по большой выборке траекторий.

Рассматриваются только такие алгоритмы, в которых итерации цикла могут выполняться независимо друг от друга и, следовательно, в произвольном порядке. Задача состоит в ускорении обработки за счёт параллельного выполнения нескольких итераций цикла на отдельных процессорах, т.е. *распараллеливания цикла* (РЦ).

Такая постановка отличается от рассматриваемой в проекте CENTAUR [11], где в теле цикла отсутствует побочный эффект, а функция обработки состоит в покоординатном преобразовании исходного вектора данных с размерностью, равной числу повторений цикла N. При этом неявно предполагается, что вычислительная сложность такого преобразования не зависит от n, а поиск наиболее эффективного решения сводится к выбору стратегии распределения элементов данных по узлам гибридного кластера. Мы же в данной работе ищем способы эффективного распараллеливания именно неоднородных циклов, число повторений которых слабо связано с размерностью исходных данных. В том числе, допуская вложенные циклы, что соответствует запуску модели при различных сочетаниях значений параметров.

В качестве примера мы рассмотрим следующие два вида неоднородных итераций со средним временем τ:

1) длительность итерации есть случайная величина, распределенная равномерно на интервале $(0, 2\tau)$;
2) длительность итерации определяется временем между двумя последовательными событиями пуассоновского процесса с интенсивностью τ.

Кратко обозначим первый вид вариантом U, второй – P.

Производительность параллельной обработки обычно характеризуют двумя показателями: общим временем обработки (или ускорением, относительно времени последовательной обработки) и эффективностью [2]. Первая характеристика показывает, насколько быстрее заданный объем обработки выполняется с помощью параллельной программы по сравнению с программой без распараллеливания, выполняемой на одном процессорном ядре. Показатель эффективности равен отношению $\eta = \left[t_0 / (mt)\right] \times 100\%$, где t_0 – время последовательной обработки одним процессором, m – число процессоров, а t – время параллельной обработки этими процессорами. Такая величина наглядно характеризует среднюю загруженность процессоров, независимо от их числа. Напротив, общее время обработки с увеличением числа процессоров в идеале убывает по гиперболическому закону, и когда число процессоров изменяется на 3–4 порядка, это затрудняет сравнение, поскольку при измерении малых интервалов времени больше сказываются случайные факторы.

Естественно считать, что чем выше загруженность процессоров (больше эффективность), тем эффективнее используется кластер. Здесь стоит уточнить, что наше рассмотрение касается алгоритмов, не использующих внешнюю память, так что процессор находится в состоянии ожидания, когда данная ветвь алгоритма ждет сообщения от другой параллельной ветви либо сигнала синхронизации от управляющего контроллера кластера.

III. Разделение области изменения индекса цикла

A. *Базовый метод*

Весь интервал изменения индекса цикла $[0, N)$ разбивается на m одинаковых порций по числу доступных процессоров. После этого k-му процессору поручается выполнение итераций с номерами $[ks, (k+1)s)$, где $s = \lfloor (N + m - 1) / m \rfloor$ – размер порции ($\lfloor * \rfloor$ обозначает целую часть). Поскольку в общем случае N не кратно m, последняя порция может содержать меньше s значений. Для реализации, однако, более удобен другой способ разбиения, когда порции выделяются не слитно, а с чередованием, так что k-й процессор выполняет итерации с номерами из k-го класса вычетов по модулю m:

```
#define NODATA -1.0
int size;   /* Total number of CPUs */
int rank;   /* This CPU number */
MPI_Init(argc, argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
double *ww =                                    (1)
  (double*)malloc(size * sizeof(double));
for (n = rank; n < N+size-1; n += size) {
  w = n < N ? Work(p[n]) : NODATA;
  MPI_Gather(&w, 1, MPI_DOUBLE,
             ww, 1, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
  if (rank == 0)
    Merge1(ww);
}
MPI_Finalize();
```

В этом алгоритме вместо *m* и *k* используются традиционные имена переменных *size* и *rank*, значения которым присваиваются перед циклом (в дальнейшем мы будем опускать такие подготовительные и заключительные операции). Функция слияния *Merge1* внутри цикла здесь имеет векторный аргумент, который заполняется результатами от всех параллельных ветвей с помощью коллективной операции MPI_Gather. Слияние осуществляется в корневой ветви с номером 0. При обработке последней порции «лишние» процессоры передают вместо результата некоторое выделенное значение NODATA, которое функция слияния должна интерпретировать надлежащим образом.

Реализация (1) необходима только для относительно сложных алгоритмов слияния (например, формирования гистограммы значений). Для тривиальных алгоритмов слияния (сумма, произведение, максимум, минимум и т.п.) вместо выделения массива и операции MPI_Gather используется одна из «готовых» коллективных операций MPI_Reduce (либо доопределяемая программистом). Например, для нахождения среднего значения результата обработки алгоритм (1) редуцируется так:

```
double sum = 0, r, w;
for (n = rank; n < N+size-1; n += size) {
  w = n < N ? Work(p[n]) : 0.0;
  MPI_Reduce(&w, &r, 1, MPI_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);          (2)
  if (rank == 0)
    sum += r;
}
sum /= N;
```

Описанный метод РЦ широко применяется для циклов с однородными итерациями, но малопригоден для неоднородного случая по следующей причине. Стандарт MPI [10] разрешает разработчику библиотеки реализовывать коллективные операции (в том числе MPI_Gather, MPI_Reduce) с синхронизацией всех в них участвующих параллельных ветвей. Это значит, что коллективный обмен сообщениями закончится не ранее, чем отработает функция *Work* в самой трудоёмкой из *m* параллельно выполняемых итераций цикла; остальные ветви будут ждать самую медленную, что значительно снижает эффективность параллельной обработки.

B. Вывод коллективных операций из тела цикла

Избежать синхронизации на каждой итерации цикла можно, разделив функцию слияния на две компоненты, первая из которых осуществляет слияние внутри цикла, выполняемого каждой ветвью (эта часть идентична первоначальной функции *Merge*), а вторая объединяет полученный в каждой ветви результат слияния после окончания цикла; она функционально идентична *Merge1* в (1). Такой алгоритм, в котором коллективная операция MPI_Gather вынесена из тела цикла, имеет вид (3). Как показано в разделе V, данный метод распараллеливания является намного более эффективным. Одновременно отпадает необходимость особым образом обрабатывать последнюю, неполную порцию итераций.

```
double *ww, r;
ww = (double*)malloc(size * sizeof(double));
for (n = rank; n < N; n += size) {
  w = Work(p[n]);
  r = Merge(w);                           (3)
}
MPI_Gather(&r, 1, MPI_DOUBLE, ww, 1,
  MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank == 0)
  Merge1(ww);
```

Аналогичное по смыслу преобразование алгоритма (2) дает следующий простой алгоритм с редукцией:

```
double sum, w = 0;
for (n = rank; n < N; n += size)
  w += Work(p[n]);
MPI_Reduce(&w, &sum, 1, MPI_DOUBLE,       (4)
  MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
  sum /= N;
```

Как легко видеть, описанный приём исключает взаимодействие параллельных ветвей, позволяя каждой асинхронно выполнять причитающуюся долю итераций цикла. Однако по окончании цикла должна выполниться заключительная коллективная операция MPI_Gather или MPI_Reduce, и в этот момент все ветви будут ждать ту, в которой цикл выполнялся дольше всего.

C. Изменение порядка итераций

Непроизводительные потери времени в конце работы алгоритмов (3), (4) можно снизить путем выравнивания суммарной трудоемкости итераций, выполняемых каждым процессором. Такая возможность возникает довольно часто, если функция обработки реализует детерминированный алгоритм, время работы которого можно вычислить или оценить в зависимости от значения параметра или размерности исходных данных. Оценка трудоемкости необязательно должна быть точной; достаточно, чтобы она задавала частичный порядок на множестве всех итераций.

С использованием такого частичного порядка можно перед началом параллельного цикла переупорядочить итерации *по убыванию* трудоемкости. Тогда каждый процессор будет сначала выполнять самые трудоемкие из назначенных ему итераций, а в конце – наименее трудоемкие. Хотя это не гарантирует полной сбалансированности ветвей, можно ожидать некоторого увеличения эффективности, особенно по сравнению с тем случаем, когда итерации выполняются заведомо в порядке *возрастания* трудоёмкости. Эксперименты показывают, что данный метод также повышает эффективность и алгоритмов (1), (2). При этом никакие изменения в алгоритмы (1)–(4) вносить не требуется – необходимое переупорядочение достигается с помощью внешних программ сортировки, применяемых к наборам исходных данных или значений параметров.

Поясним этот метод на примере из задачи кластеризации белков [12], где было необходимо

распараллелить вычисление оценок попарного сходства для двух больших наборов белков. Для вычисления каждой оценки строится выравнивание двух белковых последовательностей алгоритмом динамического программирования с трудоёмкостью $O(pq)$, где p, q – длины последовательностей. Такой параллельный вложенный цикл можно было реализовать по обычной схеме перечисления пар с формированием своего класса вычетов для каждого процессора:

```
int n = 0;
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    if (n++ % size == rank)
      Merge(Work(i,j));
```

Однако в этом случае нагрузка на параллельные ветви оказывается плохо сбалансированной, даже если заранее отсортировать оба набора последовательностей по убыванию длины. Более равномерного баланса можно достичь следующими действиями: 1) если два набора сильно отличаются числом последовательностей, в качестве первого (которому соответствует индекс i) используем набор с бóльшим числом; 2) если число последовательностей в обоих наборах одного порядка, в качестве первого используем набор с большей вариацией длины последовательностей; 3) первый набор упорядочим по убыванию длины последовательностей. Затем алгоритм реализуется по следующей схеме:

```
for (int i = rank; i < M; i += size)
  for (int j = 0; j < N; j++)
    Merge(Work(i,j));
```

Хотя такая схема не гарантирует частичного порядка для трудоёмкости индивидуальных итераций в каждой ветви, а только суммарно для блоков, соответствующих внутреннему циклу, эффективность РЦ в среднем повышается.

D. Z-образная схема упорядочения итераций

Предположим для простоты, что число итераций цикла N кратно числу процессоров и итерации упорядочены строго по убыванию трудоёмкости, как описано в предыдущем разделе. Рассмотрим в качестве примера самый простой из описанных выше алгоритмов (4).

Нетрудно заметить, что длительность i-й по порядку итерации в k-й ветви всегда больше длительности i-й по порядку итерации в k+1-й ветви для любых i, k. Следовательно, суммарное время работы ветви 0 строго больше суммарного времени ветви 1, которое, в свою очередь, строго больше суммарного времени ветви 2, и т.д. В результате в конце цикла все параллельные ветви будут ждать конца обработки в нулевой ветви, независимо от конкретных длительностей отдельных итераций.

Указанный недостаток можно ослабить, изменяя способ распределения упорядоченных по убыванию трудоёмкости итераций между процессорами. А именно, будем поочерёдно менять направление раздачи итераций на противоположное: пусть первые m итераций выполняют, соответственно, процессоры с номерами 0, 1, 2,…, m–1; следующие m итераций – процессоры с номерами m–1, m–2,…, 0; следующие m опять раздаются в порядке возрастания номеров, и т.д.:

```
double sum, w = 0;
int step = 2*rank + 1;
int range = 2*size;
for (n = rank; n < N; n += step) {
  w += Work(p[n]);
  step = range – step;                    (5)
}
MPI_Reduce(&w, &sum, 1, MPI_DOUBLE,
  MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
  sum /= N;
```

Аналогично строятся реализации Z-упорядочения и для алгоритмов (1)–(3). При таком способе распределения итераций отсутствует гарантированное преобладание трудоёмкости какой-либо ветви, т.е. нагрузка лучше сбалансирована между процессорами.

IV. Функциональное разделение ветвей

Для рассматриваемого нами случая, когда требуется распараллеливать неоднородную по трудоёмкости обработку, в литературе (например, [3]) рекомендуются методы, основанные на функциональной специализации параллельных ветвей. Такие методы реализуют архитектуру master/slave, в которой все процессоры разбиваются на две группы – рабочие и управляющие. Процессоры первой группы (slave) непосредственно осуществляют выполнение итераций цикла, а вторая группа (master), которая чаще всего состоит из одного корневого процесса, служит для организации всей вычислительной работы, т.е. раздачи заданий рабочим процессорам и слияния результатов обработки.

Тот факт, что часть процессоров непосредственно не участвуют в полезной обработке данных, теряет своё значение с увеличением общего числа процессоров. Зато этот метод хорошо отвечает именно циклу из итераций с сильно различающейся трудоёмкостью, поскольку новые задания раздаются по мере освобождения рабочих процессоров. Здесь используется двухточечный (асинхронный) обмен сообщениями, а в коллективных операциях MPI нет необходимости. Данный способ несколько усложняет алгоритм, но в целом он остаётся достаточно прозрачным.

Приведём в качестве примера фрагмент такой параллельной программы, эквивалентной алгоритму (4). В ней управляющая ветвь (с *rank* = 0) сообщает свободной рабочей ветви значение параметра для функции *Work* или выделенное значение EOJ в качестве сигнала конца работы. Эти сообщения передаются с идентификатором (тегом) TAGJOB. Рабочие ветви ожидают получения значения параметра (либо сигнала конца), вызывают функцию обработки тела цикла, а после её завершения передают значение результата корневой ветви; эти сообщения передаются с тегом

TAGDONE. Управляющая ветвь выполняет слияние результатов индивидуальных итераций (в данном случае просто суммирует их) до окончания цикла. В начале работы управляющая ветвь последовательно раздаёт задания всем доступным рабочим ветвям (и сигнал конца оставшимся, если таковые есть), а затем ожидает готовности результата от какой-либо ветви, после чего передаёт ей новое задание и т.д., пока не будут выполнены все итерации.

```
#define TAGJOB  1
#define TAGDONE 2
double EOJ = -1.0;
double sum = 0.0, w, p, *param;
int n, nfree;

if (rank == 0) {    /* Master node */
  free = size - 1;
  /* Initial distribution */
  for (n = 0; n < N && nfree > 0; n++) {
    MPI_Send(param+n, 1, MPI_DOUBLE,
      n+1, TAGJOB, MPI_COMM_WORLD);
    nfree--;
  }
  /* Free extra processors if any */
  for (int i = n+1; nfree > 0; i++) {
    MPI_Send(&EOJ, 1, MPI_DOUBLE,
      i, TAGJOB, MPI_COMM_WORLD);
    nfree--;
  }
  /* Collection/distribution loop */
  MPI_Status status;
  while (nfree < size-1) {
    MPI_Recv(&w, 1, MPI_DOUBLE,
      MPI_ANY_SOURCE, TAGDONE,
      MPI_COMM_WORLD, &status);
    int k = status.MPI_SOURCE;
    sum += w;                                    (6)
    if (n < N) {
      MPI_Send(param+n, 1, MPI_DOUBLE,
        i, TAGJOB, MPI_COMM_WORLD);
      n++;
    }
    else {
      MPI_Send(&EOJ, 1, MPI_DOUBLE,
        i, TAGJOB, MPI_COMM_WORLD);
      nfree++;
    }
  }
  sum /= N;
}

else {              /* Slave nodes */
  while (true) {
    MPI_Recv(&p, 1, MPI_DOUBLE, 0,
      TAGJOB, MPI_COMM_WORLD, &status);
    if (p == EOJ)
      break;
    w = Work(p);
    MPI_Send(&w, 1, MPI_DOUBLE, 0,
      TAGDONE, MPI_COMM_WORLD);
  }
}
```

При реализации РЦ по схеме master/slave для большого числа процессоров (порядка тысяч – десятков тысяч) и/или сложной функции слияния частичных результатов эффективность распараллеливания может уменьшиться из-за перегрузки управляющей ветви, с которой взаимодействуют все рабочие ветви. Обычное решение в таких ситуациях состоит в расширении управляющей группы. Например, приведённый выше алгоритм легко модифицируется так, чтобы не один, а два процессора (с рангом 0 и 1) управляли рабочими ветвями, соответственно, с чётными и нечётными номерами.

К сожалению, данный метод не только сложнее в реализации, но его эффективность плохо предсказуема. Как показывают наши эксперименты, для реалистичных моделей неоднородности итераций его эффективность часто уступает показателям алгоритмов из раздела III.

V. Экспериментальная проверка

Экспериментальное сравнение алгоритмов проводилось под управлением CentOS 5.6 с использованием библиотеки MVAPICH 1.2 на суперкомпьютере МВС-100К Межведомственного суперкомпьютерного центра РАН [13] при числе процессоров 64–2048 (с шагом увеличения в 1,5 раза). Была реализована однопоточная функция *Work*, обеспечивающая 100% занятость процессорного ядра и канала доступа к памяти в течение заданного времени (длительность интервала передаётся как параметр функции).

Общее время выполнения распараллеленного цикла определялось следующим образом. Непосредственно перед началом и после конца цикла в программу вставлялась операция MPI_Barrier для синхронизации всех ветвей и измерялась разность между моментами синхронизации. Результаты для каждого набора параметров усреднялись по трём измерениям.

Для обоих вариантов неоднородности (см. раздел II) общее число итераций N и средняя длительность τ выбирались так, чтобы время выполнения цикла на одном ядре составляло 1000 с (ниже приводятся данные для трёх случаев). Затем для каждого варианта с помощью псевдослучайного датчика формировался свой набор значений параметра функции обработки и для этого набора проводились измерения при каждом числе процессоров. Всего было обработано 10 таких серий, ниже в качестве примера приводятся результаты только для одной серии, т.е. для одной случайной реализации каждого из двух вариантов распределения.

Ниже в таблицах и на графиках используются следующие обозначения алгоритмов, участвовавших в эксперименте:

A1 – алгоритм (2),
A2 – алгоритм (2) с упорядочением согласно III-C,
A3 – алгоритм (2) с Z-упорядочением согласно III-D,
A4 – алгоритм (4),
A5 – алгоритм (4) с упорядочением согласно III-C,
A6 – алгоритм (5),
A7 – алгоритм (6),
A8 – алгоритм (6) с упорядочением согласно III-C.

A. *Случай $N = 10^5$, $\tau = 10$ мс*

Время счёта алгоритмов для варианта неоднородности U при различном числе процессоров приведено в таблице I, а эффективность в зависимости от числа процессоров – на рис. 1. Видно, что эффективность РЦ снижается с увеличением числа процессоров для всех алгоритмов, однако при реализации по схеме master/slave это снижение носит катастрофический характер. Для базового алгоритма (2) эффективность составляет менее 75%, а после вывода коллективной операции MPI из цикла она становится более 90%, однако снижается быстрее, чем в первом случае. Лишь после изменения порядка итераций согласно разделу III-C эффективность превышает 95%, падая только при очень большом числе процессоров. Переупорядочение дает одинаковый эффект для всех алгоритмов, кроме master/slave. Разница между обычным и Z-упорядочением (раздел III-D) для данной серии невелика и становится заметной только при числе процессоров 512 и более.

Таблица I. Время счета, с (вариант U, $N = 10^5$, $\tau = 10$ мс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	21.19	15.69	15.69	16.26	15.68	15.69	16.19	16.01
96	14.15	10.50	10.50	10.95	10.48	10.46	10.92	10.66
128	10.63	7.86	7.87	8.35	7.86	7.85	8.25	8.24
192	7.23	5.26	5.25	5.66	5.24	5.25	5.44	5.62
256	5.39	3.97	3.95	4.27	3.95	3.94	4.35	4.52
384	3.65	2.65	2.65	2.90	2.64	2.63	2.93	3.52
512	2.76	2.02	1.99	2.18	2.01	2.00	2.41	2.97
768	1.92	1.34	1.36	1.55	1.33	1.33	1.99	3.79
1024	1.45	1.02	1.02	1.16	1.00	1.01	4.11	3.67
1536	0.99	0.70	0.70	0.78	0.68	0.69	5.84	2.28
2048	0.74	0.56	0.54	0.62	0.52	0.53	6.53	11.58

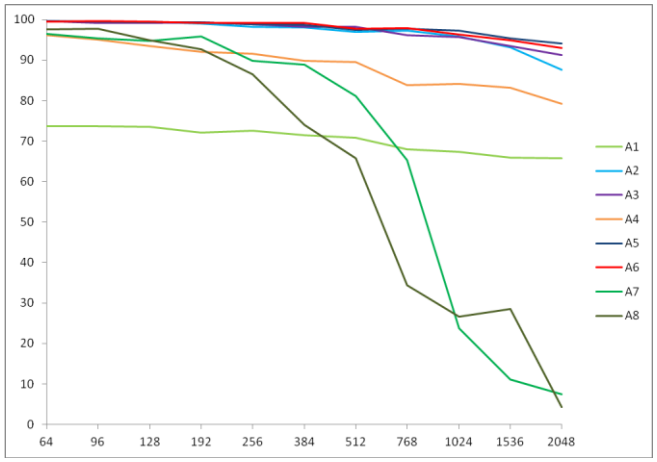

Рис. 1. Эффективность РЦ, % (вариант U, $N = 10^5$, $\tau = 10$ мс).

Для варианта P аналогичные данные представлены в таблице II и на рис. 2. Как видим, при неоднородности пуассоновского типа показатели ощутимо хуже только в случае произвольного порядка итераций (A1 и A4), а для упорядоченных итераций показатели приблизительно те же, что и в случае равномерного распределения трудоёмкости. В этом варианте более заметна разница между обычным и Z-упорядочением; последнее становится более эффективным уже при 256 и более процессорах.

Таблица II. Время счета, с (вариант P, $N = 10^5$, $\tau = 10$ мс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	30.77	15.77	15.73	16.74	15.74	15.70	16.27	16.01
96	20.83	10.56	10.50	11.35	10.56	10.48	10.77	10.81
128	15.50	7.91	7.90	8.71	7.91	7.87	8.13	8.28
192	10.67	5.31	5.29	6.11	5.29	5.28	5.79	5.98
256	8.15	4.01	3.99	4.60	4.01	3.98	4.19	4.16
384	5.59	2.69	2.66	3.13	2.69	2.67	2.98	3.85
512	4.19	2.04	2.05	2.41	2.07	2.05	2.91	3.49
768	2.92	1.39	1.36	1.76	1.43	1.36	3.74	5.13
1024	2.24	1.07	1.04	1.31	1.10	1.05	5.60	3.90
1536	1.57	0.74	0.72	0.93	0.74	0.72	2.38	2.69
2048	1.26	0.58	0.56	0.80	0.58	0.61	6.61	6.96

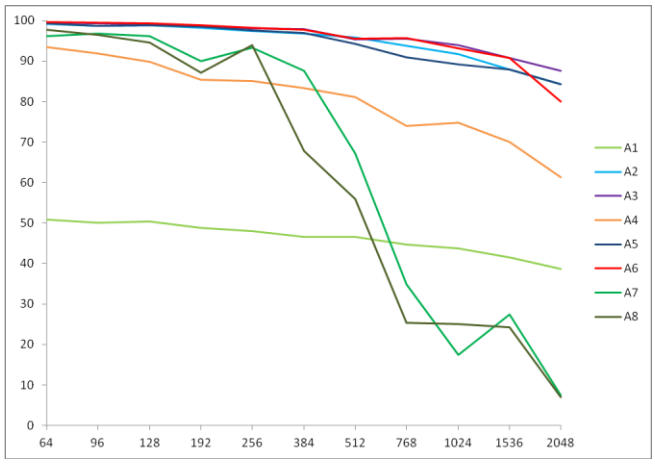

Рис. 2. Эффективность РЦ, % (вариант P, $N = 10^5$, $\tau = 10$ мс).

В целом сравнительные характеристики алгоритмов соответствуют ожидаемым, но неприятным сюрпризом стала низкая эффективность РЦ по схеме master/slave: этот алгоритм во всех случаях уступает более простым алгоритмам с разделением области изменения индекса цикла и даже переупорядочение итераций качественно не исправляет ситуацию.

Мы предполагаем, что здесь сказывается большое общее число операций MPI: для алгоритма (6) это $2N$ операций «точка-точка», а для алгоритма (4) – около N/m коллективных операций между m–1 и одним процессорами. Даже если бы коллективная операция выполнялась в m раз дольше двухточечной, имеем для алгоритма (6) двукратный проигрыш во времени. В действительности, коллективные операции MPI с помощью каскадного обмена реализуются эффективнее, чем с линейным ростом времени от числа процессоров, что только усиливает наблюдаемый эффект.

B. *Случаи $\tau = 1$ мс и $\tau = 100$ мкс*

Согласно спецификации используемого кластера [13], скорость обмена данными между узлами составляет 1400 Мбайт/с при латентности не выше 5 мкс. Поэтому коммуникационные затраты в рассмотренном выше случае в 100–1000 раз меньше длительности итерации, что и позволило лучшим алгоритмам достигать почти стопроцентной эффективности.

Рассмотрим теперь, как изменяется картина при уменьшении средней длительности итерации цикла в 10 и 100 раз.

Таблица III. Время счета, с (вариант U, $N = 10^6$, $\tau = 1$ мс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	21.11	15.93	15.89	15.84	15.76	15.74	16.10	18.12
96	14.10	10.63	10.59	10.61	10.50	10.50	12.16	11.04
128	10.61	8.03	8.02	8.00	7.88	7.87	24.45	16.17
192	7.09	5.33	5.36	5.35	5.27	5.27	5.55	19.15
256	5.38	4.06	4.05	4.04	3.95	3.95	24.83	7.14
384	3.61	2.70	2.75	2.70	2.65	2.64	7.03	23.60
512	2.72	2.07	2.06	2.04	1.99	1.99	25.22	8.95
768	1.88	1.38	1.38	1.37	1.33	1.34	9.40	9.29
1024	1.37	1.07	1.08	1.04	1.05	1.01	9.62	24.53
1536	0.97	0.72	0.72	0.71	0.68	0.68	11.96	25.94
2048	0.72	0.55	0.55	0.54	0.51	0.52	27.72	27.08

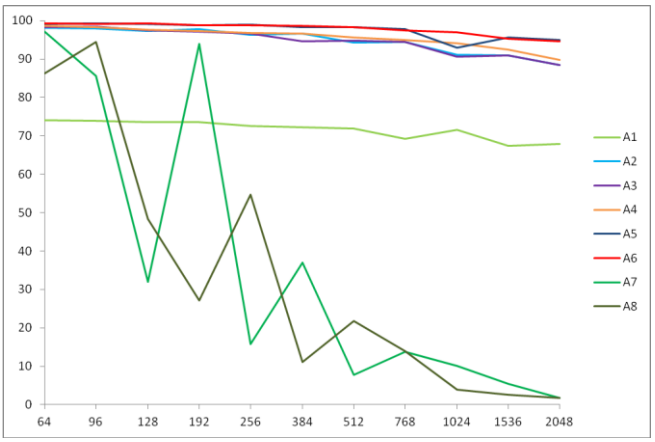

Рис. 3. Эффективность РЦ, % (вариант U, $N = 10^6$, $\tau = 1$ мс).

Таблица IV. Время счета, с (вариант P, $N = 10^6$, $\tau = 1$ мс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	30.03	16.03	15.92	15.99	15.78	15.75	16.30	16.10
96	20.13	10.68	10.73	10.74	10.51	10.50	23.01	16.85
128	15.15	8.05	8.03	8.06	7.87	7.89	24.84	17.19
192	10.11	5.37	5.44	5.42	5.26	5.26	25.02	18.54
256	7.71	4.06	4.10	4.08	3.95	3.95	25.21	21.25
384	5.19	2.71	2.70	2.76	2.64	2.65	8.40	7.89
512	3.91	2.02	2.04	2.10	1.99	1.99	25.60	8.79
768	2.67	1.37	1.47	1.41	1.36	1.34	8.91	8.57
1024	2.00	1.03	1.07	1.08	1.01	1.00	9.41	8.75
1536	1.39	0.74	0.87	0.75	0.68	0.68	8.34	8.00
2048	1.05	0.55	0.55	0.56	0.52	0.52	11.71	27.14

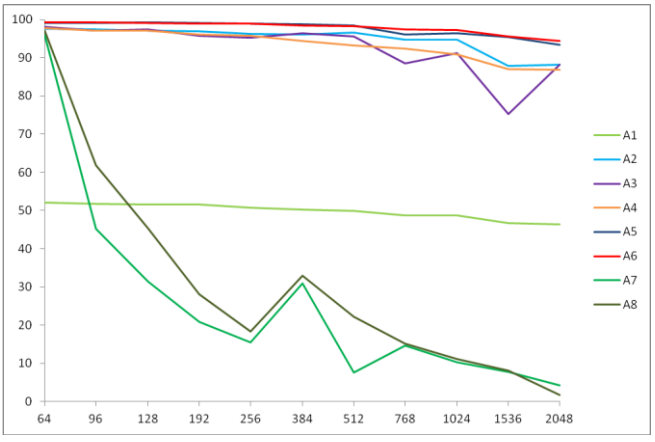

Рис. 4. Эффективность РЦ, % (вариант P, $N = 10^6$, $\tau = 1$ мс).

Таблица V. Время счета, с (вариант U, $N = 10^7$, $\tau = 100$ мкс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	1061	812	888	15.90	15.90	15.85	17.2	18.9
96	617	338	585	10.60	10.93	10.60	98.0	15.4
128	246	301	299	7.97	8.23	7.98	16.4	90.9
192	94.8	218	126	5.51	5.49	5.48	35.3	91.3
256	49.4	75.5	46.9	4.14	4.12	4.12	98.3	47.7
384	21.7	24.7	28.8	2.76	2.76	2.75	98.3	47.1
512	9.33	16.1	15.2	2.08	2.07	2.07	98.5	91.3
768	4.70	11.3	9.67	1.38	1.37	1.37	101	91.5
1024	3.07	6.09	7.46	1.03	1.04	1.05	101	93.7
1536	1.87	4.44	5.25	0.71	0.68	0.69	101	93.6
2048	1.31	4.07	4.07	0.54	0.53	0.53	105	94.9

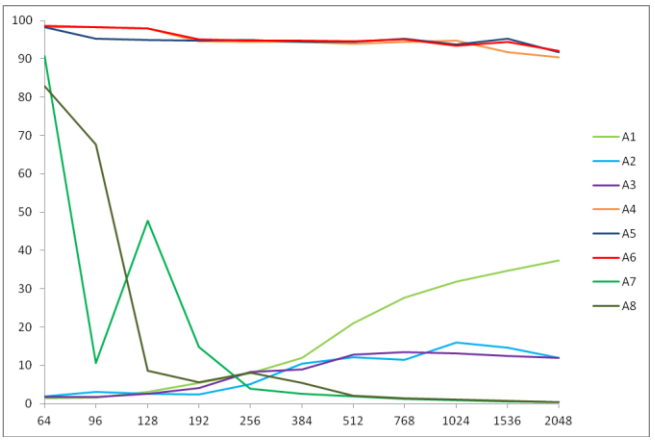

Рис. 5. Эффективность РЦ, % (вариант U, $N = 10^7$, $\tau = 100$ мкс).

Таблица VI. Время счета, с (вариант P, $N = 10^7$, $\tau = 100$ мкс).

CPU	A1	A2	A3	A4	A5	A6	A7	A8
64	611	826	1026	16.47	15.91	15.87	96.1	77.6
96	295	617	592	10.99	10.99	10.97	97.4	81.7
128	107	253	287	8.24	8.21	8.23	16.4	17.3
192	64.8	85.9	79.9	5.49	5.49	5.48	98.1	33.1
256	21.6	43.6	44.4	4.16	4.12	4.12	38.2	44.1
384	9.29	25.5	29.8	2.75	2.75	2.75	100.3	83.2
512	7.36	14.7	16.0	2.03	2.00	2.02	98.4	83.3
768	4.13	10.8	11.4	1.41	1.38	1.37	100.6	83.4
1024	2.72	6.91	6.36	1.05	1.03	1.03	101.0	85.7
1536	3.85	6.09	4.53	0.70	0.71	0.69	103.7	90.4
2048	1.07	5.19	4.20	0.52	0.53	0.53	102.0	86.7

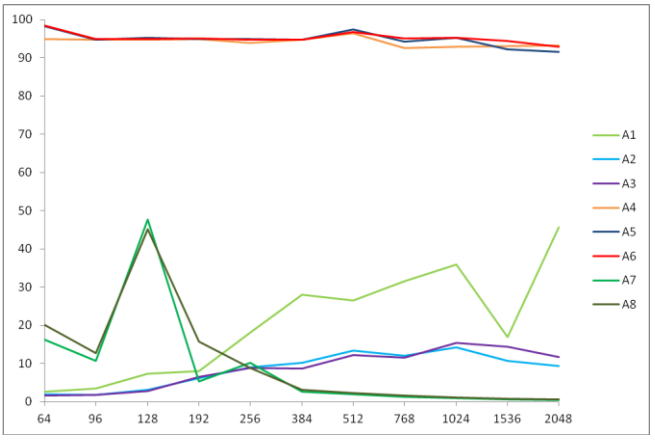

Рис. 6. Эффективность РЦ, % (вариант P, $N = 10^7$, $\tau = 100$ мкс).

Для случая $N = 10^6$, $\tau = 1$ мс данные о времени счета приведены в таблицах III (вариант U) и IV (вариант P), а зависимость эффективности от числа процессоров – на рис. 3 и 4, соответственно. В этом случае поведение алгоритмов A1–A6 аналогично случаю V-A, но эффективность РЦ несколько меньше.

Что касается реализации по схеме master/slave, ее поведение становится непредсказуемым. Наряду с общим снижением эффективности по мере увеличения числа процессоров, в соседних точках наблюдаются резкие броски, которые можно объяснить явлениями резонансного характера в коммуникационной среде кластера. Для одной серии эти броски стабильно наблюдаются при повторном счёте в одних и тех же точках, а для другой серии – в других. Эти колебания имеют большую амплитуду, несмотря на то, что продолжительность выполнения итерации по-прежнему на несколько порядков превышает время обмена.

Причина в том, что для одной и той же серии итераций с равномерно распределённой длительностью при одном числе процессоров многочисленные операции двухстороннего обмена инициируются на протяжении коротких интервалов (что ведет к перегрузке коммуникационной среды и управляющей ветви), а при другом числе процессоров итерации раздаются иначе, и описанная ситуация возникает реже. Это и отражают графики на рис. 3. Интересно, что в варианте P такая интерференция проявляется в меньшей степени (рис. 4). Заметим, что алгоритмы A1–A6 этому эффекту почти не подвержены, разве что при большом числе процессоров – от тысячи и более.

В случае $N = 10^7$, $\tau = 100$ мкс продолжительность итерации соизмерима со временем обмена, и все алгоритмы, где в цикле выполняется коллективный или двухточечный обмен, становятся несостоятельными (см. таблицу V, рис. 5 и таблицу VI, рис. 6, соответственно для вариантов U и P). Например, базовый алгоритм A1 на 64 процессорах считает даже дольше, чем на одном! Только алгоритмы A4–A6 применимы в такой ситуации и дают примерно одинаковые результаты.

VI. Заключение

Можно предположить, что представленные в разделе V данные измерений косвенно отражают специфику использованного кластера, хотя автору неизвестно о чём-то подобном. Этот суперкомпьютер в ноябре 2008 г. занял 36 место в списке Top500 и даже по сегодняшним меркам представляет собой весьма производительную установку, с действующими аналогами за рубежом. В этом смысле полученные результаты являются типичными, хотя интересно было бы провести аналогичные измерения на сегодняшних суперкомпьютерах-лидерах.

Поэтому ограничимся только самыми осторожными выводами, которые можно сделать по результатам проведённого исследования.

1. Эффективное распараллеливание циклов с разными по трудоёмкости итерациями на системах с распределённой памятью является нетривиальной задачей, требующей анализа и натурных экспериментов.

2. Итерации с экспоненциальным распределением трудоёмкости распараллеливаются в целом менее эффективно, чем в случае равномерного распределения (при одном и том же суммарном времени обработки).

3. Использование коллективных операций MPI внутри параллельного цикла сильно снижает эффективность из-за эффекта синхронизации. Следует по возможности проводить слияние результатов обработки локально внутри каждой параллельной ветви, а объединение результатов ветвей выполнять вне цикла.

4. Если вывести коллективную операцию MPI из цикла невозможно, настоятельно рекомендуется вести обработку итераций в порядке убывания их трудоёмкости. При этом в ряде случаев дополнительный выигрыш можно получить с помощью Z-упорядочения.

5. Реализация РЦ по схеме master/slave, часто рекомендуемая именно для циклов с неоднородными итерациями, показала в наших экспериментах меньшую эффективность, чем более простые алгоритмы, и к тому же демонстрирует плохо предсказуемое поведение.

Библиография

[1] Рубанов Л. И. Обработка больших данных в параллельных циклах //Сборник избранных трудов VIII Международной конференции «Современные информационные технологии и ИТ-образование». – 2013. – М.: ИНТУИТ.РУ, 2013. – С. 769-772.

[2] Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.

[3] Лупин С. А., Посыпкин М. А. Технологии параллельного программирования. – М.: ИД «ФОРУМ», 2011. – 208 с.

[4] Любецкий В. А., Рубанов Л. И., Селиверстов А. В., Пирогов С. А. Модель регуляции экспрессии генов у бактерий на основе формирования вторичных структур РНК //Молекулярная биология. – 2006. – Т. 40, № 3. – С. 497-511.

[5] Lev Rubanov and Vassily Lyubetsky. RNAmodel Web Server: Modeling Classic Attenuation in Bacteria //In Silico Biology. – 2007. – Vol. 7, no. 3. – P. 285-308. http://www.bioinfo.de/isb/2007/07/0044/

[6] Любецкий В. А., Жижина Е. А., Рубанов Л. И. Гиббсовский подход в задаче эволюции регуляторного сигнала экспрессии гена //Проблемы передачи информации. – 2008. – Т. 43, выпуск 4. – С. 52-71.

[7] Lyubetsky V. A., Zverkov O. A., Rubanov L. I., Seliverstov A. V. Modeling RNA polymerase competition: the effect of sigma-subunit knockout and heat shock on gene transcription level //Biology Direct. – 2011. Vol. 6, no. 3. http://www.biologydirect.com/content/6/1/3

[8] Lyubetsky V. A., Zverkov O. A., Pirogov S. A., Rubanov L. I., Seliverstov A. V. Modeling RNA polymerase interaction in mitochondria of chordates //Biology Direct. – 2012. – Vol. 7, no. 26. http://www.biologydirect.com/content/7/1/26

[9] The OpenMP API specification for parallel programming. http://openmp.org/wp/openmp-specifications/

[10] Message-passing interface forum. MPI Documents http://www.mpi-forum.org/docs/docs.html

[11] CENTAUR software tools for hybrid supercomputing http://centaur.botik.ru/home

[12] Любецкий В. А., Селиверстов А. В., Зверков О. А. Построение разделяющих паралоги семейств гомологичных белков, кодируемых в пластидах цветковых растений //Математическая биология и биоинформатика. – 2013. – Т. 8, № 1. – С. 225-233.

[13] Межведомственный суперкомпьютерный центр Российской академии наук. http://www.jscc.ru/scomputers.shtml

Parallel Monte Carlo Modeling on Distributed Memory Systems

Lev I. Rubanov

***Abstract*—**The modeling of multiple Monte Carlo iterations in parallel is considered for distributed memory supercomputers. The study aims at effective utilization of the computer capacity in the cases of essentially different complexity of iterations and/or performance of processors. Existing methods of the process loop parallelization are studied and empirical estimates of efficiency are obtained for two patterns of nonuniform iterations.

***Keywords*—**distributed memory, Monte Carlo modeling, nonuniform loop, parallelization.