Информационные процессы, Том 13, № 4, 2013, стр. 295–305.

ПРОГРАММИРОВАНИЕ

О распараллеливании неоднородных циклов на суперкомпьютерах с распределённой памятью[1]

Л.И. Рубанов

Институт проблем передачи информации, Российская академия наук, Москва, Россия

Поступила в редколлегию 17.10.2013

Аннотация—Рассматривается задача построения типового алгоритма для параллельного исполнения независимых итераций многократного цикла на мультипроцессорной вычислительной системе с распределённой памятью. Алгоритм должен обеспечивать эффективное использование вычислительной мощности системы при существенно различающейся трудоёмкости отдельных итераций и/или производительности отдельных процессоров, независимо от числа используемых процессоров. Предполагается, что межпроцессорный обмен данными и управление параллельными вычислениями реализуются посредством стандартного интерфейса передачи сообщений (MPI), широко применяемого на подобных системах. Проанализированы существующие методы распараллеливания циклов и получены эмпирические оценки их эффективности для различных моделей неоднородности итераций.

КЛЮЧЕВЫЕ СЛОВА: высокопроизводительные вычисления, распараллеливание, распределённая память, MPI, неоднородный цикл.

1. ВВЕДЕНИЕ

Для большинства компьютерных программ характерно, что операторы цикла встречаются в них относительно редко по сравнению с арифметическими и логическими операторами, зато внутри тела цикла выполняется основной объём вычислений. Если ход выполнения очередной итерации цикла алгоритма зависит от результатов предшествующих итераций, то единственная возможность ускорения вычислений состоит в оптимизации тела цикла, что лежит за рамками данной статьи. Мы будем рассматривать только алгоритмы, в которых итерации цикла могут выполняться независимо друг от друга. В частности, это означает, что порядок выполнения итераций также может быть произвольным. Подобные алгоритмы часто возникают в задачах имитационного моделирования физико-химических и биологических процессов, обработки больших массивов данных, численного интегрирования и др. Например, при моделировании случайного процесса методом Монте-Карло с целью оценки вероятностей конечных состояний каждую траекторию модели можно представлять как одну итерацию внешнего цикла, предназначенного для вычисления необходимых оценок. Чтобы получить достоверные и точные оценки, такой цикл необходимо повторить большое число раз, и отсюда возникает потребность в высокопроизводительных вычислениях. Естественный путь ускорения цикла с независимыми итерациями состоит в одновременном выполнении сразу нескольких итераций цикла на отдельных процессорах, т.е. в *распараллеливании цикла* (РЦ).

Возможности и методы РЦ существенно зависят от архитектуры мультипроцессорной системы, на которой производятся вычисления. В суперкомпьютерах с *общей памятью*, где каждый процессор имеет независимый доступ к общему полю оперативной памяти, РЦ часто реализуется средствами системы параллельного программирования OpenMP [1], которая

[1] Работа выполнена при поддержке гранта Министерства образования и науки РФ, соглашение №8481.

сочетает удобство для программиста с эффективностью. Системы с общей памятью широко распространены (сегодня большинство настольных рабочих станций и ПК оснащаются многоядерными процессорами), однако число процессоров в них редко превышает 64, что сильно ограничивает возможности повышения производительности. Увеличить скорость вычислений в сотни и тысячи раз возможно только на системах с *распределённой памятью* (кластерах), где процессоры организованы в группы (узлы), каждая со своей отдельной памятью, недоступной для прочих узлов кластера. Связь между процессами, выполняющимися на различных узлах (в том числе и территориально разнесённых), осуществляется средствами коммуникационных библиотек параллельного программирования, среди которых широкое распространение получил стандартный интерфейс передачи сообщений (MPI) [2]. Далее мы будем рассматривать именно такой случай. Разумеется, это не исключает РЦ внутри узла средствами OpenMP, однако особой необходимости в таком усложнении конструкции цикла нет, поскольку в большинстве реализаций MPI обмен сообщениями внутри узла организуется через его общую память, т.е. достаточно эффективно, что позволяет программисту разрабатывать единый код для параллельного исполнения как внутри узла, так и на кластере в целом.

Другой важный аспект, который необходимо учитывать при РЦ — это сравнительная трудоёмкость индивидуальных итераций, т.е. насколько *однородным* является цикл. Если длительность каждой итерации приблизительно постоянная, их можно выполнять синхронно, как это делается в систолической архитектуре. Однако в упомянутых выше задачах вычислительная сложность итерации часто меняется в широких пределах в зависимости от размерности данных, обрабатываемых на данной итерации, или от текущего значения псевдослучайной последовательности. Аналогичная картина возникает и при одинаковой трудоёмкости итераций, если узлы кластера имеют различную производительность. Использование в таких ситуациях алгоритмов РЦ, подразумевающих обработку по систолическому принципу, приводит к неэффективным решениям, как будет показано ниже. Нас в первую очередь будет интересовать распараллеливание существенно неоднородных циклов и притом в условиях распределённой памяти.

2. ОБЩАЯ ПОСТАНОВКА И МЕТОДИКА ОЦЕНКИ ЭФФЕКТИВНОСТИ

Рассматриваемую задачу алгоритмически можно представить следующим одномерным циклом (здесь и далее мы будем использовать для записи алгоритмов синтаксис C, что несущественно для изложения), который требуется распараллелить:

Алгоритм 1.

```
for (n = 0; n < N; n++) {
   f = Func(param[n]);
   Merge(f);
}
```

Здесь предполагается, что функция `Func` без побочного эффекта, однако в ней сосредоточена основная вычислительная сложность тела цикла. Скалярная функция используется в качестве основного примера, но возможны и ситуации, когда `Func` возвращает пару значений или вектор. То же относится к её параметру `param`. Заметим, что это именно параметр, а не аргумент; в частном случае он может отсутствовать. Мы будем предполагать, что все необходимые данные для вычисления значения функции имеются внутри неё, например, содержатся в статическом массиве или генерируются с помощью псевдослучайной последовательности. Это не исключает возможности циклической обработки массивов данных, коль скоро функция может получать в качестве параметра само значение переменной цикла n.

Напротив, функция `Merge` вычислительно простая, но обладает побочным эффектом, который и приводит к результату всего цикла вычислений. Например, эта функция может вычислять сумму или произведение отдельных значений f, находить минимальное или максимальное значения, формировать гистограмму и т.п. Таким образом, в данной постановке не предполагается хранить результаты каждой итерации цикла (что может вызвать затруднения при больших N), а ставится задача получения сводных данных статистического характера по большой выборке вычисленных значений.

Изложенная постановка задачи РЦ отличается от рассматриваемой в проекте CENTAUR [3], где в теле цикла отсутствует побочный эффект, а основная функция обеспечивает только покоординатное преобразование исходного вектора данных, размерность которого равна числу повторений цикла N. Неявно предполагается, что вычислительная сложность такого преобразования практически не зависит от n (является постоянной). Тем самым поиск наиболее эффективного решения по РЦ сводится к выбору надлежащего распределения элементов данных по узлам вычислительной сети с гибридной архитектурой. Не умаляя важности этого вопроса, мы хотим сосредоточиться на эффективном распараллеливании существенно неоднородных циклов, число повторений которых слабо связано с размерностью исходных данных. В том числе, с возможностью вложенных циклов, что бывает полезно в комбинаторных алгоритмах (например, обработка всевозможных пар исходных данных удобнее реализуется в виде двойного вложенного цикла).

Оценка эффективности рассматриваемых ниже способов РЦ проводилась по следующей методике. Была реализована функция `Func`, обеспечивающая практически 100% загрузку процессора и канала обращения к памяти в течение заданного интервала времени, длительность которого передаётся в качестве параметра. Непосредственно перед началом и после конца цикла производилась синхронизация процессоров и измерялась разность между этими моментами времени, которая принималась за время счёта данного цикла. Чтобы сохранить точность измерений при изменении числа используемых процессоров в широком диапазоне, увеличение числа процессоров сопровождалось пропорциональным увеличением длительности интервала времени работы функции `Func`. Таким образом, при идеальном РЦ время счёта цикла независимо от числа процессоров должно оставаться постоянным. Так, в случае однородного цикла с длительностью итерации τ (при одном процессоре) время счёта в идеале должно быть равно $N\tau$ при любом числе процессоров. Ниже в таблицах представлена (в процентах) величина отклонения от этого идеального значения. Естественно, она во всех случаях положительная, поскольку идеал недостижим.

Результаты для постоянного τ приводятся всюду для сравнения (обозначение “модель C”). Помимо этого, рассматриваются следующие модели неоднородности цикла:

U: длительность итерации есть случайная величина, равномерно распределённая на интервале $(0, 2\tau)$;

P: длительность итерации определяется временем между последовательными событиями пуассоновского процесса с интенсивностью τ;

L: итерация имеет линейную сложность относительно n, т.е. длительность равна $C_1(n + 1)$, где константа $C_1 \approx 2\tau/N$ выбирается из условия, чтобы средняя длительность итерации составляла τ;

Q: итерация имеет квадратичную сложность по n, т.е. длительность равна $C_2(n + 1)^2$, где константа $C_2 \approx 3\tau/N^2$ выбирается так, чтобы средняя длительность составляла τ.

Таким образом, для всех пяти моделей ожидаемое “идеальное” время счёта равно $N\tau$ при любом числе процессоров.

Описанные в статье результаты были получены на суперкомпьютере МВС-100К Межведомственного суперкомпьютерного центра РАН [4].

3. РАЗДЕЛЕНИЕ ОБЛАСТИ ИЗМЕНЕНИЯ ПЕРЕМЕННОЙ ЦИКЛА

Поскольку итерации цикла не зависят друг от друга, естественно появляется идея РЦ за счет выполнения на каждом процессоре своей части итераций. Вся область изменения целочисленной переменной цикла $[0, N)$ делится на m приблизительно одинаковых порций по числу процессоров, и k-й процессор выполняет только итерации с номерами $[ks, (k+1)s)$, где $s = \lfloor (N + m - 1)/m \rfloor$ — размер порции ($\lfloor \cdot \rfloor$ обозначает целую часть). Тем самым Алгоритм 1 приобретает следующий вид (мы опускаем очевидные объявления некоторых переменных):

Алгоритм 2.

```
MPI_Comm_size(MPI_COMM_WORLD, &m);
MPI_Comm_rank(MPI_COMM_WORLD, &k);

int s = (N+m-1)/m;
double f0 = -1.0;
double *ff = (double*)malloc(m*sizeof(double));

for (n = k*s; n < (k+1)*s; n++) {
   f = n < N ? Func(param[n]) : f0;
   MPI_Gather(&f, 1, MPI_DOUBLE, ff, 1, MPI_DOUBLE,
              0, MPI_COMM_WORLD);
   if (k==0)
      Merge1(ff);
}
```

Здесь после вычисления очередного значения f в каждой ветви эти значения собирает корневая ветвь с номером $k = 0$, для чего используется отличающаяся от первоначальной функция слияния `Merge1`. Отличие состоит в том, что она принимает векторный аргумент; размерность вектора равна общему числу процессоров. Поскольку N может быть некратно m, пришлось ввести особое выделенное значение f_0, в данном случае -1, которое в последнем проходе цикла передают неиспользуемые ветви параллельной программы. Наряду с прочим, задача функции `Merge1` — узнавать и игнорировать эти данные.

Алгоритм 2 относится к относительно сложному случаю, когда функция слияния выполняет нетривиальные действия, например, строит гистограмму значений f. Для более простых случаев стандарт MPI предусматривает готовые функции коллективного обмена сообщениями со слиянием и возможность определения новых типов функций. Например, для вычисления среднего значения f цикл Алгоритма 2 можно упростить до следующего:

Алгоритм 3.

```
double sum = 0;
double r;
for (n = k*s; n < (k+1)*s; n++) {
   f = n < N ? Func(param[n]) : 0.0;
   MPI_Reduce(&f, &r, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   if (k==0)
      sum += r;
}
```

По окончании цикла переменная `sum` в корневой ветви содержит суммарное значение f по всем итерациям цикла, остается лишь разделить его на число итераций N. Роль "особого" значения f_0 в Алгоритме 3 играет константа 0, не изменяющая величины накопленной суммы.

Результаты РЦ по Алгоритмам 2–3 для разных моделей неоднородности итераций цикла представлены в Таблице 1. Здесь и далее использовались значения $N = 10^6$, $\tau = 1$ мс, так что идеальное время счета всегда равно 1000 с.

Таблица 1. Отклонение в процентах от идеального времени счета при РЦ по Алгоритмам 2–3.

Число процессоров	Модель неоднородности итераций цикла				
	C	U	P	L	Q
1	0.25	0.18	0.16	0.19	0.17
2	0.25	17.5	33.5	50.1	75.1
4	0.13	26.9	62.1	75.1	131
8	0.34	33.5	89.7	87.6	164
16	0.36	33.5	89.8	93.8	182
32	0.07	33.7	90.0	96.9	191
64	0.06	33.9	91.4	98.4	195
128	0.06	34.6	92.2	99.2	198
256	0.04	35.0	95.4	99.6	199
512	0.06	36.4	98.9	99.8	199
1024	0.06	37.5	101.4	99.9	200

Из таблицы хорошо видно, что данный способ РЦ применим к однородным итерациям, но плохо подходит для всех рассмотренных моделей неоднородного цикла. С ростом числа процессоров эффективность распараллеливания падает в 2 раза и более. Такого результата можно было ожидать, потому что стандарт MPI [2] разрешает разработчику библиотеки выполнять коллективные операции (включая MPI_Gather, MPI_Reduce) с синхронизацией всех участвующих процессов. Это значит, что коллективный обмен сообщениями завершится не ранее, чем будет вычислена функция `Func` в самой трудоёмкой из m параллельно выполняемых итераций. Остальные ветви ждут этого момента, что и отражает Таблица 1. Если вычислительную сложность каждой итерации удаётся оценить априори, то указанный эффект можно ослабить, переупорядочив итерации так, чтобы параллельно выполнялись итерации примерно одинаковой трудоёмкости, однако описанная схема разделения области изменения переменной цикла для этого неудобна, и мы вернёмся к данному вопросу позже.

Итак, видна основная возможность повышения эффективности РЦ в Алгоритмах 2–3, а именно: отказаться от коллективного ввода/вывода внутри цикла, вынося его вовне. Слияние результатов работы функции `Func` при этом будет выполняться локально в каждой параллельной ветви. По завершении цикла во всех ветвях выполняется слияние этих сводных результатов, для чего потребуется коллективная операция. Тем самым получим

Алгоритм 4.

```
double *ff = (double*)malloc(m*sizeof(double));
double r;
for (n = k; n < N; n+=m) {
   f = Func(param[n]);
   r = Merge(f);
}
MPI_Gather(&r, 1, MPI_DOUBLE, ff, 1, MPI_DOUBLE,
           0, MPI_COMM_WORLD);
if (k==0)
   Merge1(ff);
```

В Алгоритме 4 реализован другой способ разделения области изменения переменной цикла, при котором подобласти выделяются не слитно, а с чередованием. Это позволяет отказаться от использования особого численного признака выхода за границу цикла (как в Алгоритмах 2– 3) и тем существенно упрощает программу. Кроме того, этот способ лучше применим к распараллеливанию вложенных циклов, для чего схема чередования слегка модифицируется, например, с помощью отображения вектора индексов циклов на натуральный ряд и назначения k-му процессору элементов из k-го класса вычетов по модулю числа процессоров:

```
int n = 0;
for (i = 0; i < M; i++) {
   for (j = 0; j < N; j++) {
      if (n++ % m != k)
         continue;
      f = Func(...)
      ...
   }
}
```

Как и в случае Алгоритма 2, если операция слияния результатов индивидуальных итераций уже предусмотрена в стандарте MPI или может быть доопределена программистом, Алгоритм 4 значительно упрощается. Например, для вычисления среднего значения приходим к алгоритму

Алгоритм 5.

```
double sum;
double r = 0;
for (n = k; n < N; n+=m)
   r += Func(param[n]);
MPI_Reduce(&r, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Результаты РЦ по усовершенствованному алгоритму представлены в Таблице 2.

Таблица 2. Отклонение в процентах от идеального времени счета при РЦ по Алгоритмам 4–5.

Число процессоров	Модель неоднородности итераций цикла				
	C	U	P	L	Q
1	0.25	0.18	0.17	0.19	0.17
2	0.10	0.11	0.22	0.11	0.09
4	0.05	0.18	0.24	0.07	0.06
8	0.20	0.22	0.36	0.21	0.17
16	0.29	0.37	0.64	0.14	0.18
32	0.06	0.83	1.17	0.12	0.09
64	0.05	1.14	2.13	0.07	0.06
128	0.06	2.20	2.70	0.04	0.04
256	0.05	2.74	4.04	0.04	0.05
512	0.06	4.29	6.24	0.06	0.08
1024	0.05	6.06	9.28	0.10	0.15

Как видим, отказ от коллективных операций MPI внутри цикла с неоднородными итерациями существенно повышает эффективность РЦ. Однако, если трудоёмкость итераций подвержена значительным случайным флуктуациям (модели U, P), то с увеличением числа процессоров эффективность РЦ заметно снижается, и это представляет резерв для дальнейшего улучшения.

4. ИЗМЕНЕНИЕ ПОРЯДКА ИТЕРАЦИЙ

На первый взгляд, данные Таблицы 2 способны вызвать недоумение. Средняя длительность итерации во всех случаях равна τ, значит для модели U длительность лежит в интервале $(0, 2\tau)$, для L — тоже $(0, 2\tau)$, и для Q — $(0, 3\tau)$. Сложнее обстоит дело с моделью P (пуассоновский поток), где верхнюю границу указать нельзя, хотя вероятность больших значений крайне мала. Для определённости сообщим, что данные в части модели P были получены на псевдослучайной серии длиной $N = 10^6$, которая реализовалась на интервале $(0, 13.3\tau)$, что отчасти объясняет более плохие результаты для этой модели. Но почему при одинаковом интервале длительностей у моделей U и L для первой РЦ осуществляется настолько хуже, тем более при сравнении с моделью Q, где длительность итерации меняется в даже более широких пределах?

Причина становится понятной, если вспомнить, что в моделях L и Q длительность итерации монотонно возрастает с увеличением n. Поэтому суммарная длительность итераций, выполняемых каждой параллельной ветвью Алгоритмов 4 или 5, приблизительно одинаковая, во всяком случае, сопоставимая. Напротив, в модели U длительность итераций распределена равномерно по n, поэтому в одну ветвь могут попасть итерации большой длительности, а в другие — малой. Общее время счёта цикла, разумеется, определяет худший случай: все раньше завершившиеся ветви программы простаивают, ожидая завершения коллективной операции, идущей вслед за концом цикла. С ростом N этот эффект ослабляется (сумма нормализуется в силу центральной предельной теоремы), но из-за медленной сходимости его приходится принимать в расчёт.

Отсюда возникает естественная идея, применимая в ситуациях, когда трудоёмкость каждой итерации можно оценить априори. А именно, упорядочим все итерации по убыванию их сложности, так чтобы в каждой параллельной ветви Алгоритмов 4 и 5 длительность выполняемых итераций монотонно убывала. Это способствует выравниванию суммарного времени работы ветвей и, следовательно, уменьшению времени их простоя в конце работы.

Силу этого метода продемонстрируем на примере рассмотренного вначале Алгоритма 3, т.е. с коллективной операцией внутри распараллеливаемого цикла. Мы лишь изменим схему разделения области изменения переменной цикла подобно Алгоритму 5, поскольку первоначальная схема не балансирует загруженность ветвей. В итоге получим

Алгоритм 6.

```
MPI_Comm_size(MPI_COMM_WORLD, &m);
MPI_Comm_rank(MPI_COMM_WORLD, &k);
double sum = 0;
double r, f;
int s = ((N+m-1)/m)*m;
for (n = k; n < s; n+=m) {
   f = n < N ? Func(param[n]) : 0.0;
   MPI_Reduce(&f, &r, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   if (k==0)
      sum += r;
}
```

Если выполняемые итерации упорядочены по убыванию длительности с ростом n, то Алгоритм 6 показывает результаты, представленные в Таблице 3. Как видим, эффективность РЦ во всех случаях уступает идеальному менее 1%, несмотря на наличие коллективной операции MPI внутри исполняемого цикла.

Таблица 3. Отклонение в процентах от идеального времени счета при РЦ по Алгоритму 6.

Число процессоров	Модель неоднородности итераций цикла				
	C	U	P	L	Q
1	0.25	0.21	0.18	0.19	0.19
2	0.25	0.15	0.15	0.18	0.15
4	0.17	0.10	0.09	0.09	0.16
8	0.35	0.31	0.10	0.36	0.31
16	0.36	0.20	0.20	0.24	0.17
32	0.06	0.12	0.15	0.12	0.21
64	0.06	0.08	0.11	0.06	0.06
128	0.07	0.05	0.13	0.04	0.05
256	0.05	0.05	0.22	0.05	0.06
512	0.06	0.06	0.45	0.06	0.09
1024	0.05	0.12	0.93	0.12	0.16

В заключение этого раздела заметим, что возможность априорной оценки трудоёмкости итерации цикла не является чем-то особенным. Так, если в цикле проводится обработка больших данных, то вычислительная сложность обработки может быть прямо связана с размерностью обрабатываемого элемента данных, причем зачастую известным образом. Например, в задаче кластеризации белков [5] на каждой итерации требуется вычислять методом динамического программирования парное выравнивание двух аминокислотных последовательностей с длинами m и n, что характеризуется вычислительной сложностью $O(mn)$. Это позволяет осуществить перебор пар последовательностей в порядке убывания величины mn, реализуя тем самым описанный здесь метод РЦ. Более того, наличие априорной оценки трудоёмкости итерации позволяет строить более сложные, чем простое упорядочение, динамические стратегии планирования цикла обработки для заданного числа процессоров. Этот вопрос, однако, выходит за рамки данной статьи.

5. ФУНКЦИОНАЛЬНОЕ РАЗДЕЛЕНИЕ ВЕТВЕЙ

Обсуждение задачи распараллеливания неоднородного цикла было бы неполным без рассмотрения методов РЦ, основанных на функциональной специализации процессоров, т.е. ветвей параллельной программы. Эти методы реализуют архитектуру master/slave, в которой все процессоры разбиваются на две группы — рабочие и управляющие. Процессоры первой группы (slave) осуществляют непосредственно выполнение итераций цикла, а вторая группа (master), чаще всего состоящая из одного корневого процесса, служит для организации всей вычислительной работы, т.е. раздачи заданий рабочим процессорам.

Тот факт, что часть процессоров непосредственно не участвуют в полезной обработке данных, теряет свое 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. При такой организации используется двухточечный обмен сообщениями, а в коллективных операциях обычно нет необходимости. Данный метод несколько усложняет алгоритм, но в целом он остается достаточно прозрачным.

В качестве примера на рис. 1 представлен фрагмент полного текста такой параллельной программы. В ней управляющая ветвь (с $rank = 0$) сообщает свободной рабочей ветви значение параметра для функции Func или “особое” значение -1 в качестве сигнала завершения работы. Эти сообщения передаются с идентификатором (тегом) 1. Рабочие ветви ожидают получения значения параметра (либо сигнала окончания), вызывают основную функцию тела цикла, а после ее завершения передают значение f корневой ветви; эти сообщения передаются с тегом 2. Управляющая ветвь выполняет слияние результатов индивидуальных итераций (в

данном примере просто суммирует их) вплоть до окончания цикла. В начале работы управляющая ветвь последовательно раздаёт задания всем доступным рабочим ветвям (и сигнал завершения оставшимся, если таковые есть), а затем ожидает готовности результата от какой-либо ветви, после чего передает ей новое задание и т.д., пока не будут выполнены все итерации.

Характеристики алгоритма РЦ на рис. 1 для разных видов неоднородности цикла приведены в Таблице 4. Видно, что за счёт полного отказа от коллективных операций и более эффективного управления рабочими ветвями уже при 128 и более процессорах эффективность РЦ уступает идеальной менее 1%. В то же время, при числе доступных процессоров 64 и менее этот метод уступает ранее рассмотренным. Как и следовало ожидать, качество распараллеливания мало зависит от вида неоднородности, а определяется в первую очередь числом процессоров. Эффективность можно дальше повышать путём переупорядочения итераций, как описывалось в разделе 4.

Таблица 4. Отклонение в процентах от идеального времени счета при РЦ по схеме master/slave.

Число процессоров	Модель неоднородности итераций цикла				
	C	U	P	L	Q
2	100	100	100	100	100
4	33.5	33.5	33.4	33.5	33.4
8	14.5	14.6	14.3	14.4	14.4
16	6.86	6.76	6.76	6.84	6.85
32	3.30	3.36	3.30	3.28	3.28
64	1.64	1.66	1.71	1.67	1.70
128	0.85	0.83	0.90	0.86	0.96
256	0.43	0.44	0.55	0.47	0.71
512	0.25	0.29	0.53	0.30	0.47
1024	0.24	0.31	0.84	0.44	0.78

При реализации изложенного метода РЦ следует учитывать, что при очень большом числе процессоров (порядка тысяч и десятков тысяч) и/или сложном алгоритме слияния частичных результатов эффективность закономерно снижается из-за перегрузки управляющей ветви, с которой взаимодействуют все рабочие ветви. В таких ситуациях следует увеличивать управляющую группу. Например, программа на рис. 1 легко переделывается так, чтобы не один, а два процессора (с рангами 0 и 1) управляли рабочими ветвями, соответственно, с чётными и нечётными номерами.

6. ЗАКЛЮЧЕНИЕ

Описанные в статье методы распараллеливания циклов были успешно применены в Лаборатории математических методов и моделей в биоинформатике ИППИ РАН при решении ряда задач [6]–[9], для которых были разработаны оригинальные параллельные алгоритмы. Более подробная информация об этих разработках представлена на сайте лаборатории http://lab6.iitp.ru.

СПИСОК ЛИТЕРАТУРЫ

1. The OpenMP API specification for parallel programming, http://openmp.org/wp/

2. Message-passing interface forum. MPI Documents, http://www.mpi-forum.org/docs/docs.html

3. CENTAUR software tools for hybrid supercomputing, http://centaur.botik.ru/home

4. Межведомственный суперкомпьютерный центр Российской академии наук. Вычислительные системы. http://www.jscc.ru/scomputers.shtml

```
int      size, rank;
int      N = 100000;
int      n, i, free;
double  *param;
double   f, p;
double   Sum = 0.0;
double   term = -1.0;
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
if (rank == 0) {     // Master node
   free = size-1;

   // Initial distribution
   for (n = 0; n < N && free > 0; n++) {
      MPI_Send(param+n, 1, MPI_DOUBLE, n+1, 1, MPI_COMM_WORLD);
      free--;
   }

   // Free extra processors if any
   for (i = n+1; free > 0; i++) {
      MPI_Send(&term, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
      free--;
   }

   // Collection/distribution loop
   MPI_Status status;
   while (free < size-1) {
      MPI_Recv(&f, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
               2, MPI_COMM_WORLD, &status);
      i = status.MPI_SOURCE;
      Sum += f;
      if (n < N) {
         MPI_Send(param+n, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
         n++;
      }
      else {
         MPI_Send(&term, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
         free++;
      }
   }
}

else {               // Slave node(s)
   while (true) {
      MPI_Recv(&p, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status);
      if (p < 0)       // End-of-processing signal
         break;
      f = Func(p);
      MPI_Send(&f, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
   }
}
```

Рис. 1. Распараллеливание цикла по схеме master/slave.

5. Зверков О.А., Селиверстов А.В., Любецкий В.А. Белковые семейства, специфичные для пластомов небольших таксономических групп водорослей и простейших. *Молекулярная биология*, 2012, том 46, №5, стр. 799–809.

6. Lyubetsky V.A., Rubanov L.I., Seliverstov A.V. Lack of conservation of bacterial type promoters in plastids of Streptophyta. *Biology Direct*, 2010, vol. 5, no. 34.

7. Lyubetsky V.A., Zverkov O.A., Rubanov L.I., Seliverstov A.V. Modeling RNA polymerase competition: the effect of σ-subunit knockout and heat shock on gene transcription level. *Biology Direct*, 2011, vol. 6, no. 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.

9. Lyubetsky V.A., Rubanov L.I., Rusin L.Yu., Gorbunov K.Yu. Cubic time algorithms of amalgamating gene trees and building evolutionary scenarios. *Biology Direct*, 2012, vol. 7, no. 48.

On the parallelization of nonuniform loops in distributed memory supercomputers

L.I. Rubanov

We consider the problem of a template algorithm building for parallel execution of independent iterations of the repetitive loop on a multiprocessor supercomputer with distributed memory. Irrespective of the processor quantity, the algorithm must ensure efficient utilization of the computing capacity in the cases of essentially different complexity of iterations and/or performance of processors. We assume that interprocessor data communications and the control of parallel computations are accomplished through the standard message-passing interface (MPI) widely used on such systems. Existing methods of the loop parallelization are studied and empirical estimates of efficiency are obtained for various models of the iterations nonuniformity.

KEYWORDS: high-performance computing, parallelization, distributed memory, MPI, nonuniform loop.