УДК 681.3.06

СРАВНЕНИЕ Т-СИСТЕМЫ И MPI НА ЗАДАЧЕ EP ИЗ ПАКЕТА ТЕСТОВ NPB 2.3

В. А. Евсеенко[1], **М. Н. Иванов**[1], **В. Ю. Радыгин**[1]

Выполнено сравнение Т-системы автоматического динамического распараллеливания вычислительных алгоритмов с широко известным подходом, основанным на применении MPI (Message Passing Interface) для составления параллельных программ. Сравнение осуществлялось на одной из задач, включенных в стандартный набор тестов NASA.

Ключевые слова: динамическое распараллеливание, Message Passing Interface, параллельные программы, генераторы случайных чисел, гауссово распределение, равномерное распределение.

Введение. В данной работе выполнено сравнение возможностей Т-системы [1] автоматического динамического распараллеливания, разработанной в Институте программных систем РАН, с другим, наиболее широко используемым на данный момент средством MPI для составления распределенных программ. Сравнивались реализации одной задачи из стандартного набора тестов NASA, написанные на языках t2cp (Т-система) и Fortran 77 (MPI). В последнем случае использовались исходные тексты, взятые без каких-либо изменений с сервера NASA [2].

Для сравнения был выбран тест "Embarrassing Parallel" (EP), основанный на порождении пар псевдослучайных нормально распределенных чисел. Для удобства сравнения результата строится гистограмма и считаются суммы полученных пар чисел.

Равномерное распределение случайных чисел. Многие компьютерные системы содержат генераторы случайных чисел в своих стандартных библиотеках. Обычно используются линейные генераторы псевдо-случайных чисел, которые создают последовательности целых чисел $I_1, I_2, I_3, \ldots$, принадлежащих некоторому отрезку $[0, m-1]$. Генерация этих чисел осуществляется по рекуррентному соотношению

$$I_{j+1} = aI_j + c \,(\mathrm{mod}\ m).$$

Множитель a и слагаемое c — положительные целые, а модуль m — некоторое достаточно большое положительное целое. Если эти параметры выбраны удачно, то последовательность будет максимальной длины m. В этом случае мы получим все числа от 0 до $m-1$, прежде чем последовательность начнет повторяться. Для того чтобы получить действительные числа из полуинтервала $[0, 1)$, достаточно поделить элементы последовательности на m: $R_j = I_j/m$. Качество такого генератора зависит лишь от параметров рекуррентного соотношения и не является удовлетворительной для практического применения.

Нормальное (гауссово) распределение случайных чисел. Попробуем преобразовать числа, полученные указанным выше способом так, чтобы получить псевдо-случайные числа с нормальным распределением

$$p(y)\,dy = \frac{1}{2\pi}\, e^{-y^2/2}\, dy.$$

Можно предложить следующий способ установления соответствия между двумя представителями равномерного распределения и двумя представителями нормального распределения:

$$y_1 = \sqrt{-\ln(x_1)}\,\cos(2\pi x_2),$$

$$y_2 = \sqrt{-\ln(x_1)}\,\sin(2\pi x_2).$$

Обратное преобразование задается соотношениями

$$x_1 = e^{-\frac{1}{2}(y_1^2+y_2^2)},$$

$$x_2 = \frac{1}{2\pi}\arctan\left(\frac{y_2}{y_1}\right).$$

[1] Московский государственный индустриальный университет, Автозаводская ул., 16, 109280, Москва; e-mail: evseenko@msiu.ru, ivanov@msiu.ru, radigin@msiu.ru

Алгоритм тестовой задачи. Рассмотренный способ был положен за основу при разработке алгоритма тестовой задачи. Линейную часть алгоритма можно представить в виде нескольких шагов.

Первый шаг. Положим $a = 5^{13}$, $n = 2^{28}$, $s = 271828183$. Вычислим $r_i = 2^{-46}x_i$, где i меняется в пределах от 1 до n, а x_i определяются следующим образом:

$$x_0 = s,$$

$$x_{i+1} = ax_i \,(\mathrm{mod}\, 2^{46}),$$

$$x_i \;= a^i s \,(\mathrm{mod}\, 2^{46}).$$

Второй шаг. Вычислим x_j и y_j по формулам

$$x_j = 2\, r_{2j-1} - 1,$$

$$y_j = 2\, r_{2j} - 1,$$

где j принимает все целые значения из отрезка $[1, n]$. Полученные значения x_j и y_j будут принадлежать отрезку $[-1, 1]$.

Третий шаг. Положим $k = 0$ и $j = 1$. Если $t_j = {x_j}^2 + {y_j}^2 \leqslant 1$, то k полагается равным $k + 1$ и вычисляется пара чисел

$$X_k = x_j \sqrt{(-2\log t_j)/t_j},$$

$$Y_k \;= y_j \sqrt{(-2\log t_j)/t_j}.$$

В противном случае значение j увеличивается на 1. Эти вычисления повторяются, если $j \leqslant n$.

Таким образом, на третьем шаге мы получаем последовательность $\{X_k, Y_k\}$ искомых пар псевдослучайных чисел с гауссовым распределением.

Четвертый шаг. Строится таблица частот попадания $\max(|X_k|, |Y_k|)$ в каждый из десяти полуинтервалов $[l, l + 1)$, где l меняется от 0 до 9.

Реализация теста EP в T-системе. Общий алгоритм теста EP, реализованный нами на T-языке, описан в [3]. Изучение алгоритма показывает, что задачу можно легко дробить по размеру отрезка для индекса в рекуррентных соотношениях. Расчет по каждому отрезку будет абсолютно независим. Для каждого отрезка можно получить собственную гистограмму. Таким образом, передача данных между параллельными процессами будет происходить лишь на заключительном этапе, когда происходит сложение полученных гистограмм в одну общую. На этой идее основаны реализации теста на T-языке и с использованием средств MPI.

Структура реализации теста на T-языке имеет следующий вид.

```
// Определение основных структур
...
// Обычная C функция, которая подсчитывает
// гистограмму для полуотрезка [k, k+packet_len)
void make_gist(u64 k,u64 packet_len, u64 a, u64 s, pack_res *res)
...
// T-функция, дробящая на гранулы
‘func iterator(parm)-> (res);// parm - отрезок, res - гистограмма
   ...
   if(params.rec_count == 0 ){// расчет гистограммы для отрезка parm
      // результат формируется в my_res
     ...
   }else{ // разбиваем отрезок пополам: t1 и t2
     ...
     t1‘newPackedHolder(sizeof(pack_prm));
     t2‘newPackedHolder(sizeof(pack_prm));
     ...
     ‘[res1]=iterator(t1); // рекурсивный вызов для t1
     ‘[res2]=iterator(t2); // рекурсивный вызов для t2
     // суммирование гистограмм
```

```
      ...
   }
‘res <== my_res;
‘end_func

‘func tmain(arg) -> (rc);
‘vars res,prm,retval;
   ...
  ‘[res]=iterator(prm);
   ...
‘end_func
```

Задачу распараллеливания в нашей реализации выполняет функция iterator, которая рекурсивно вызывает саму себя, порождая тем самым бинарное дерево вызовов функций глубиной 10. Таким образом, мы получаем 1024 гранулы параллелизма. На заключительном этапе те же вызванные функции, соответствующие внутренним узлам дерева, собирают все 1024 гистограммы в одну.

Результаты сравнения. В процессе сравнения тест проходил испытания на различных кластерах с различным количеством рабочих процессоров и с различными показателями их тактовой частоты. Наибольший интерес для нас представляет результат, полученный на кластере НИВЦ МГУ, одной из отличительных особенностей которого является однородность всех его узлов.

Структура кластера НИВЦ МГУ. Кластер (по состоянию на 2000 г.) содержит 13 узлов: 12 машин 2×Pentium III 500 и Pentium III 500. Основные 12 машин, помимо Fast Ethernet, связаны при помощи SCI (Scalable Coherent Interface).

Кратко характеристики межпроцессорных связей кластера можно описать следующим образом: латеность для SCI — 5.6 мксек, для Fast Ethernet — 158 мксек (для сообщений нулевой длины), скорость для SCI — 80 MB/sec, для Fast Ethernet — 10 MB/sec (для больших сообщений).

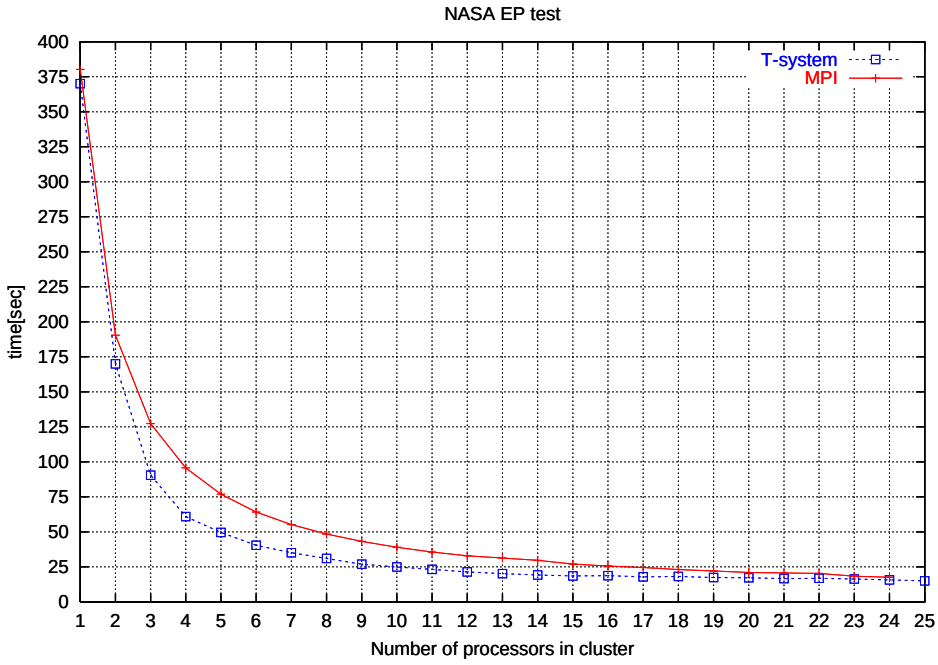

Рис. 1. График зависимости времени вычисления от числа задействованных процессоров

В процессе тестирования количество процессоров изменялось от 1 до 25. Графическое представление полученных результатов приведено на рис. 1.

Важно отметить, что T-система использовала в качестве средства связи Fast Ethernet, а MPI-программа на Fortrane 77 использовала SCI. Несмотря на эту заметную разницу, наша реализация не только не отстает от реализации на MPI, но и обгоняет ее практически на любом сочетании процессоров.

т	t_c T-sys	t_c MPI	t_p T-sys	t_p MPI	K T-sys	K MPI	f T-sys	f MPI	$\frac{K_{\text{T}}}{K_{\text{MPI}}}$	$\frac{t_{c\text{MPI}}}{t_{c\text{T}}}$
1	370.08	380.30	100.0	100.00	1.000	1.00	1.00	1.00	1.00	1.03
2	169.97	190.50	45.92	50.09	2.177	2.00	1.09	1.00	1.09	1.12
3	90.45	127.30	24.44	33.47	4.091	2.99	1.36	1.00	1.37	1.40
4	60.95	95.70	16.46	25.16	6.072	3.97	1.52	0.99	1.53	1.57
5	49.58	76.95	13.39	20.23	7.464	4.94	1.49	0.99	1.51	1.55
6	40.52	64.15	10.94	16.87	9.133	5.93	1.52	0.99	1.54	1.58
7	35.16	55.25	9.503	14.53	10.52	6.88	1.50	0.98	1.53	1.57
8	31.09	48.50	8.400	12.75	11.90	7.84	1.49	0.98	1.52	1.56
9	26.95	43.28	7.282	11.38	13.73	8.79	1.52	0.98	1.56	1.61
10	24.91	39.07	6.730	10.27	14.85	9.73	1.48	0.97	1.53	1.57
11	23.23	35.63	6.276	9.37	15.93	10.67	1.45	0.97	1.49	1.53
12	21.46	32.93	5.801	8.66	17.23	11.55	1.44	0.96	1.49	1.53
13	20.18	31.37	5.452	8.25	18.33	12.12	1.41	0.93	1.51	1.55
14	19.28	29.75	5.209	7.82	19.19	12.78	1.37	0.91	1.50	1.54
15	18.61	27.05	5.028	7.11	19.88	14.06	1.32	0.94	1.41	1.45
16	18.74	25.58	5.063	6.73	19.74	14.87	1.23	0.93	1.33	1.37
17	17.93	24.55	4.844	6.46	20.64	15.49	1.21	0.91	1.33	1.37
18	18.19	23.12	4.915	6.08	20.34	16.45	1.13	0.91	1.24	1.27
19	17.44	22.14	4.712	5.82	21.22	17.18	1.12	0.90	1.24	1.27
20	17.24	21.02	4.658	5.53	21.46	18.09	1.07	0.90	1.19	1.22
21	16.70	20.67	4.512	5.44	22.16	18.40	1.06	0.88	1.20	1.24
22	16.80	20.40	4.542	5.36	22.01	18.64	1.00	0.85	1.18	1.21
23	16.34	18.36	4.415	4.83	22.64	20.71	0.98	0.90	1.09	1.12
24	15.74	17.61	4.253	4.63	23.51	21.60	0.98	0.90	1.09	1.12

Анализ результатов. Для более полного анализа введем следующие дополнительные параметры: n — количество использованных процессоров; t_s — время в секундах; t_p — время в процентах; K — коэффициент ускорения; $f = \dfrac{t(1)}{t(n) * n}$ — функция утилизации, где $t(x)$ — время расчета задачи на x процессорах.

Функция утилизации показывает, насколько отличается реальный коэффициент ускорения с увеличением числа используемых процессоров от "желаемого" линейного роста. Обычно ожидается, что $0 \leqslant f \leqslant 1$. В нашем случае функция утилизации иногда превышает единицу. Для объяснения этого существует несколько версий. Мы приведем наиболее правдоподобную. Она заключается в том, что из-за локализации данных процессор лучше работает с кэшем, отсюда и получается более чем линейное ускорение. В таблице приведены полученные результаты для обеих реализаций.

Выводы. Конечно, сравнение Т-системы и MPI всего на одной задаче не позволяет формировать далеко идущие выводы. Однако для теста EP имеет место следующее:

— Т-система позволяет более эффективно распараллелить задачу;

— в случае Т-системы функция утилизации f имеет "более хорошие" значения;

— с ростом числа процессоров "падение" значений функции f меньше для случая Т-системы (масштабируемость задачи — лучше).

При этом еще раз подчеркнем, что Т-система использовала для межпроцессорных передач Fast Ethernet и TCP/IP (Transmission Control Protocol/Internet Protocol), а NASA-программы — SCI и MPI.

Вторая тестовая задача. Постановку задачи о *реализации поиска подграфа в графе* можно сформулировать следующим образом. Пусть даны два графа: основной, в котором будем искать вхождения, и более маленький (по количеству вершин), выполняющий роль разыскиваемого подграфа. Необходимо определить все возможные вхождения подграфа в граф или выдать сообщение, что их нет.

Для решения этой задачи была составлена программа на Т-языке, которая проходила тестирование на кластере НИВЦ МГУ. Количество процессоров изменялось от 1 до 25. По полученным результатам был построен график представленный на рис. 2. На графике можно видеть сравнение полученных нами результатов с линейным ускорением. Видно, что наша реализация для Т-системы отклоняется от него незначительно.

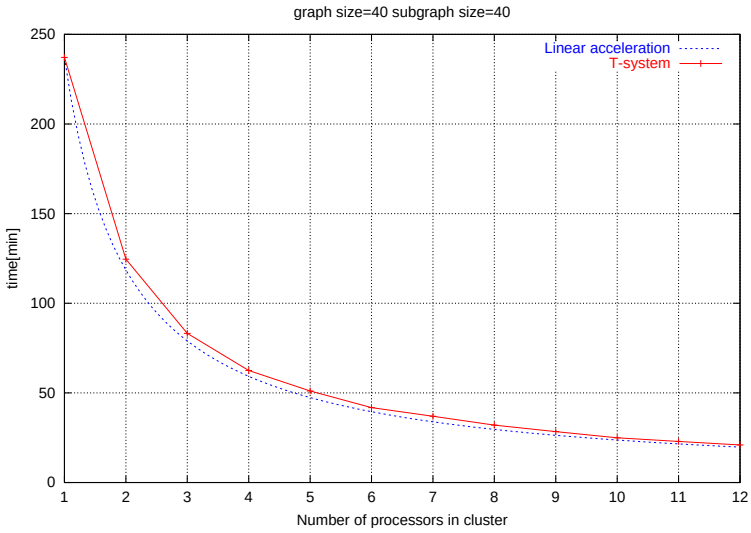

Рис. 2. Результаты работы программы поиска подграфа

Авторы выражают благодарность НИВЦ МГУ за возможность работы на его кластере. Особенно хотелось бы поблагодарить А. Андреева за оказание огромной помощи в проведении работы.

СПИСОК ЛИТЕРАТУРЫ

1. Документация по T-системе и T-языку. `http://www.botik.ru/~t-system`
2. Исходные тексты тестов NASA. `http://www.nas.nasa.gov/Software`
3. Официальный сервер NASA, документация по тестам. `http://www.nas.nasa.gov/Software/npb`

Поступила в редакцию
27.01.2001