ЭФФЕКТИВНОСТЬ ПАРАЛЛЕЛЬНОЙ РЕАЛИЗАЦИИ АЛГОРИТМА RADIX-4 БЫСТРОГО ПРЕОБРАЗОВАНИЯ ФУРЬЕ

А.А. Морозов[1], А.В. Тимофеев[2,1]

[1]*Департамент прикладной математики,*
Национальный исследовательский университет "Высшая школа экономики",

[2]*ОИВТ РАН, Москва, Россия*

timofeevalvl@gmail.com

Поступила 03.08.2015

Параллельная программа для быстрого преобразования Фурье реализована на основе технологии параллельного программирования MPI (Message Passing Interface). В качестве базового метода для быстрого преобразования Фурье использован алгоритм Radix-4. Исследована зависимость ускорения параллельного расчёта от числа процессоров на примере двух вычислительных кластеров. Предложена формула, описывающая зависимость времени расчёта от числа процессоров, объёма входных данных и характеристик вычислительной системы. Сделаны оценки числа узлов, при котором достигается максимальное ускорение расчёта.

УДК 004.421.2, 004.032.24

1. Введение

Круг задач, в которых необходимо использовать преобразование Фурье, очень велик. Это, например, и цифровая обработка сигналов, и сжатие данных, и криптография. Для анализа мы выбрали дискретное преобразование Фурье (ДПФ), поскольку современная вычислительная техника является преимущественно цифровой, т.е. оперирует с дискретными величинами.

ДПФ активно используется в научных исследованиях, в частности, в квантовых расчётах и моделировании систем частиц, взаимодействующих по дальнодействующим потенциалам. Одним из важных ограничений на пути моделирования больших систем является время расчёта. Необходимость уменьшения времени расчёта делает вопрос параллелизации алгоритмов вычисления ДПФ востребованными в научной среде. Проблема параллелизации состоит в том, что временные затраты на межпроцессорную коммуникацию и принципиально последовательные части алгоритмов способны нивелировать всё преимущество параллелизма [1].

В данный момент самыми популярными являются два способа решения этой проблемы. Во-первых, можно попытаться модифицировать последовательный алгоритм. Во-вторых, можно запускать параллельную программу на массивах очень большой длины. В этом случае временные затраты на пересылку сообщений между узлами будут сильно меньше, чем временные затраты на собственно “полезные” вычисления. Таким образом, даже во втором случае мы всё равно получим выигрыш, если не для всевозможных входных данных, то хотя бы для некоторых.

Далее мы рассматриваем алгоритмы БПФ и параллельную реализацию одного из алгоритмов БПФ. Изучено поведение параллельной версии алгоритма на различных кластерах, предложена формула для описания зависимости ускорения от числа процессоров для параллельной реализации алгоритма Radix-4.

2. БПФ и последовательная реализация алгоритма Radix-4

ДПФ задаётся следующей формулой $X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk}$, где $\omega_N^{nk} = e^{\frac{2\pi i n k}{N}}$. Напомним, что преобразование Фурье сигнала, зависящего от времени, на выходе даёт массив с информацией о гармониках, формирующих данный сигнал.

Алгоритмы быстрого преобразования Фурье (БПФ) способны вычислять эту сумму в среднем за $O(n \log_2 n)$.

В качестве основы для будущей параллельной версии мы выбрали алгоритм БПФ Radix-4 [2-4]. Этот алгоритм работает следующим образом: массив разбивается на четыре части, вычисляется ДПФ каждой части (существенно, что для каждой части будет использован тот же метод), а затем все части особым образом склеиваются, и мы получаем массив с результирующим ДПФ.

Из этого описания следует, что Radix-4 применим только для массивов длиной 4^n. Мы предложим здесь некоторую модификацию, которая позволит вычислять ДПФ от массивов длиной 2^n [1].

Рассмотрим два способа разбиения входного массива на части. Во-первых, можно просто поместить в первую часть первую четверть элементов, во вторую часть – вторую четверть и т. д. Во-вторых, можно в первую часть поместить все элементы, индексы которых кратны 4, во вторую – элементы, индексы которых дают остаток 1 при делении на 4 и т. д. (рис. 1). Первый подход называется прореживанием по частоте (англ. Decimation In Frequency, DIF), а второй – прореживание по времени (англ. Decimation In Time, DIT). Эти способы совершенно равнозначны (за исключением особенностей реализации), и мы в данной статье будем использовать DIT.

Более конкретно алгоритм Radix-4 DIT выглядит так:

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk}$$

$$= \sum_{n=0}^{\frac{N}{4}-1} x_{4n} \omega_N^{4nk} + \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} \omega_N^{(4n+1)k} + \sum_{n=0}^{\frac{N}{4}-1} x_{4n+2} \omega_N^{(4n+2)k}$$

$$+ \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} \omega_N^{(4n+3)k}$$

$$= \sum_{n=0}^{\frac{N}{4}-1} x_{4n} \omega_{\frac{N}{4}}^{nk} + \omega_N^k \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} \omega_{\frac{N}{4}}^{nk} + \omega_N^{2k} \sum_{n=0}^{\frac{N}{4}-1} x_{4n+2} \omega_{\frac{N}{4}}^{nk} + \omega_N^{3k} \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} \omega_{\frac{N}{4}}^{nk}$$

$$= A_k + \omega_N^k B_k + \omega_N^{2k} C_k + \omega_N^{3k} D_k$$

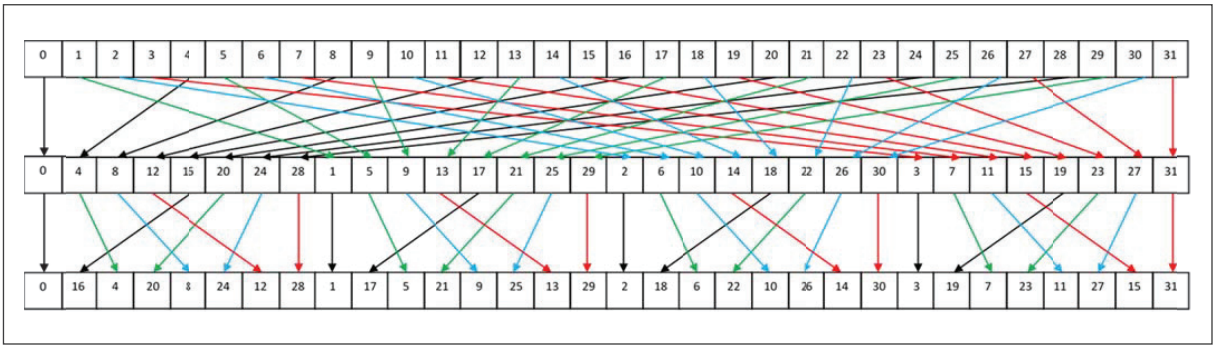


Рис. 1. Перестановка элементов входного массива в алгоритме Radix-4.

[1] Строго говоря, это будет уже алгоритм Mixed-Radix. Подробнее о нём см. в [3].

Из этого равенства находим:

$$\begin{cases} X_r = A_r + \omega_N^r B_r + \omega_N^{2r} C_r + \omega_N^{3r} D_r \\ X_{r+\frac{N}{4}} = A_r + \omega_N^{r+\frac{N}{4}} B_r + \omega_N^{2(r+\frac{N}{4})} C_r + \omega_N^{3(r+\frac{N}{4})} D_r \\ X_{r+\frac{N}{2}} = A_r + \omega_N^{r+\frac{N}{2}} B_r + \omega_N^{2(r+\frac{N}{2})} C_r + \omega_N^{3(r+\frac{N}{2})} D_r \\ X_{r+\frac{3N}{4}} = A_r + \omega_N^{r+\frac{3N}{4}} B_r + \omega_N^{2(r+\frac{3N}{4})} C_r + \omega_N^{3(r+\frac{3N}{4})} D_r \end{cases}, \qquad 0 \le r < \frac{N}{4}$$

Окончательно получаем:

$$\begin{cases} X_r = A_r + \omega_N^r B_r + \omega_N^{2r} C_r + \omega_N^{3r} D_r \\ X_{r+\frac{N}{4}} = A_r + i\omega_N^r B_r - \omega_N^{2r} C_r - i\omega_N^{3r} D_r \\ X_{r+\frac{N}{2}} = A_r - \omega_N^r B_r + \omega_N^{2r} C_r - \omega_N^{3r} D_r \\ X_{r+\frac{3N}{4}} = A_r - i\omega_N^r B_r - \omega_N^{2r} C_r + i\omega_N^{3r} D_r \end{cases}, \qquad 0 \le r < \frac{N}{4}$$

Последние соотношения называются преобразованием бабочки для алгоритма Radix-4 DIT. Именно они показывают, как следует склеивать четыре ДПФ в одно. Коэффициенты ω_N^r, ω_N^{2r} и ω_N^{3r} называются поворачивающими множителями (англ. Twiddle Factors).

Кроме того, для нашей модификации алгоритма Radix-4 (позволяющей вычислять ДПФ от массив длины 2^m) нам потребуется иногда склеивать на четыре части в одно ДПФ, а две. Пользуясь аналогичным приёмом, мы можем получить преобразование бабочки и для этого случая:

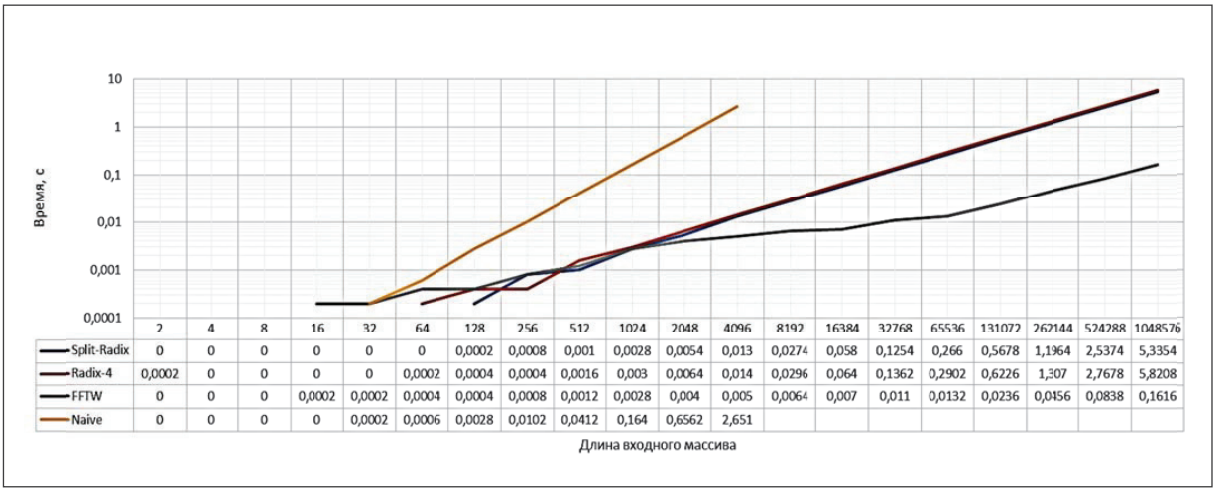
$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \omega_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \omega_N^{(2n+1)k} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \omega_{\frac{N}{2}}^{nk} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \omega_{\frac{N}{2}}^{nk}$$
$$= A_k + \omega_N^k B_k$$

Отсюда получаем:

$$\begin{cases} X_k = A_k + \omega_N^k B_k, & 0 \le k < \frac{N}{2} \\ X_k = A_k - \omega_N^k B_k, & \frac{N}{2} \le k < N \end{cases}$$

Здесь поворачивающими множителями будут ω_N^k.

Для анализа быстродействия работы последовательных алгоритмов проведено сравнение алгоритмов Split-Radix, Radix-4, профессионального оптимизированного программного пакета FFTW и простого ДПФ (Naive) (рис. 2).



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
Split-Radix	0	0	0	0	0	0	0,0002	0,0008	0,001	0,0028	0,0054	0,013	0,0274	0,058	0,1254	0,266	0,5678	1,1964	2,5374	5,3354
Radix-4	0,0002	0	0	0	0	0,0002	0,0004	0,0004	0,0016	0,003	0,0064	0,014	0,0296	0,064	0,1362	0,2902	0,6226	1,307	2,7678	5,8208
FFTW	0	0	0	0,0002	0,0002	0,0004	0,0004	0,0008	0,0012	0,0028	0,004	0,005	0,0064	0,007	0,011	0,0132	0,0236	0,0456	0,0838	0,1616
Naive	0	0	0	0	0,0002	0,0006	0,0028	0,0102	0,0412	0,164	0,6562	2,651								

Рис. 2. Сравнение последовательных алгоритмов ДПФ для набора случайных входных данных

Видно, что графики рассмотренных нами алгоритмов в целом представляют собой прямые. Это ещё раз подтверждает (уже на практике) оценку сложности $O(n \log_2 n)$. Во-вторых, заметно, что алгоритм Split-Radix работает хоть и ненамного, но быстрее алгоритма Radix-4. Вызвано это тем, что он сам по себе производит меньше операций [3]. Надо так же отметить, что в приведённых здесь реализациях не применялись серьёзные оптимизации, которые можно обнаружить в профессиональных библиотеках БПФ. К таким оптимизациям относятся эффективная работа с памятью [4], применение схем разбиения, которые наилучшим образом подходят для данной архитектуры [4], а иногда и применение SIMD [4]. По этой причине имеется такой выигрыш во времени у библиотеки FFTW. Кроме того, наблюдается ожидаемое преимущество всех алгоритмов БПФ по сравнению с простой реализацией ДПФ.

3. Параллельная реализация алгоритма Radix-4

Программа разработана для числа узлов, равного 2^m. Такая программа позволяет достаточно точно оценить ускорения вычисления БПФ при параллелизации.

Мы будем использовать декомпозицию по данным. Работа программы схематично изображена на рисунке ниже:

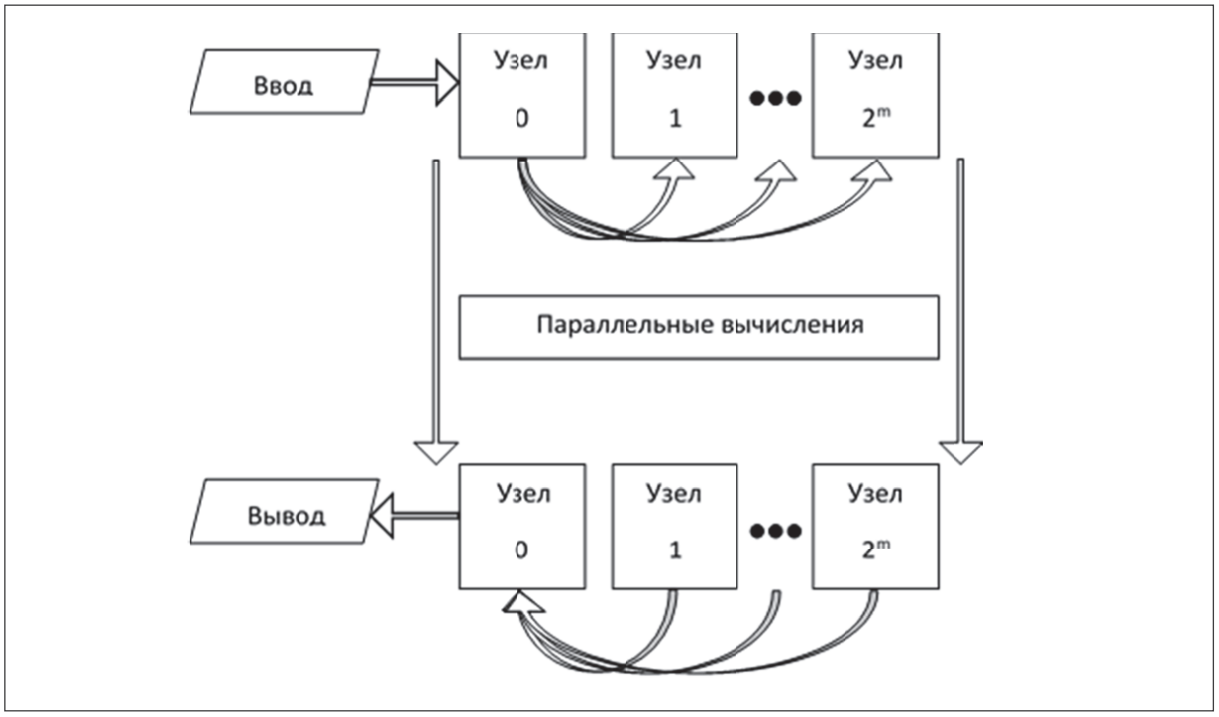


Рис. 3. Схема параллельного алгоритма БПФ

Последовательность действий, выполняемых программой (рис. 3), такова:

1. Узел 0 считывает входные данные (либо из файла, либо генерирует массив случайных величин)
2. Узел 0 осуществляет перестановку элементов входного массива
3. Подготовленный входной массив распределяется по узлам
4. Каждый узел вычисляет ДПФ своей части (эта работа выполняется параллельно)
5. Полученные ДПФ собираются на узле 0
6. Все имеющиеся на нулевом узле ДПФ склеиваются в одно
7. Узел 0 производит вывод результирующего массива

Для распределения данных по узлам и для сборки их на одном узле мы будем использовать соответствующие функции MPI: Scatter() и Gather().

Отличие описанного алгоритма от Radix-4 состоит в том, что входной массив разбивается не на четыре части, а на 2^m частей. Если m чётное, то на узле 0 произойдёт склеивание всех этих частей в точности так, как это требуется в алгоритме Radix-4. Сначала объединятся первые четыре части, затем вторые четыре и т. д.. Затем четвёрки из образовавшихся новых частей тоже будут объединятся между собой, и так будет происходить до тех пор, пока не останется одна часть, которая и будет являться ДПФ исходного массива.

Если же m нечётное, то всё будет происходить так же, как и в предыдущем случае, за исключением того, что на последнем шаге у нас будут не четыре части, а две. Их-то и надо будет объединить между собой с помощью преобразования бабочки для алгоритма Radix-2, которое мы описали выше.

Отсюда следует, что перестановка элементов входного массива (рис. 1) также будет зависеть от чётности m. Если m чётное, то она такая же, как в алгоритме Radix-4, если же m нечётное, то вначале массив следует разделить на две части – в одну попадут элементы с чётными индексами, а в другую – с нечётными. Далее в этом случае всё делается так же, как и в предыдущем.

Теперь мы можем перейти к обсуждению результатов, которые дало распараллеливание.

Главным показателем, позволяющем судить, насколько параллельная программа эффективнее последовательной, является ускорение. По определению ускорение на p процессорах задаётся формулой:

$$S_p = \frac{T_1}{T_p},$$

где T_p— время выполнения программы на p процессорах.

Мы запускали программу на двух вычислительных кластерах: NWO5 (ОИВТ РАН) (рис.4) и K-100 (ИПМ РАН) (рис.6). Первая вычислительная система уже является устаревшей, в то время как вторая активно используется в настоящее время в серьёзных расчётах. Значения ускорений на этих двух кластерах дают оценку диапазона ускорений для большинства современных вычислительных кластеров.

В качестве входных данных использовались массивы случайных комплексных чисел. Длины массивов были в пределах от 2^{20} до 2^{25}.

Для кластера NWO5 результаты показаны на графиках на рис. 4.

Как мы видим, ускорение составляет в среднем два раза на кластере nwo5. Так же видно, что некоторые массивы лучше считать на четырёх узлах, а некоторые на восьми (возможно, лучше даже на 16, но на NWO5 нет такого числа вычислительных узлов).

Единственное непонятное обстоятельство – это резкое падение ускорения на втором и последнем графиках. Чтобы объяснить это, выполним запуск нашей программы не 10 раз, а 50, а затем построим график эмпирической функции распределения, чтобы понять, вокруг какой величины “группируются” в основном времена (рис. 5).

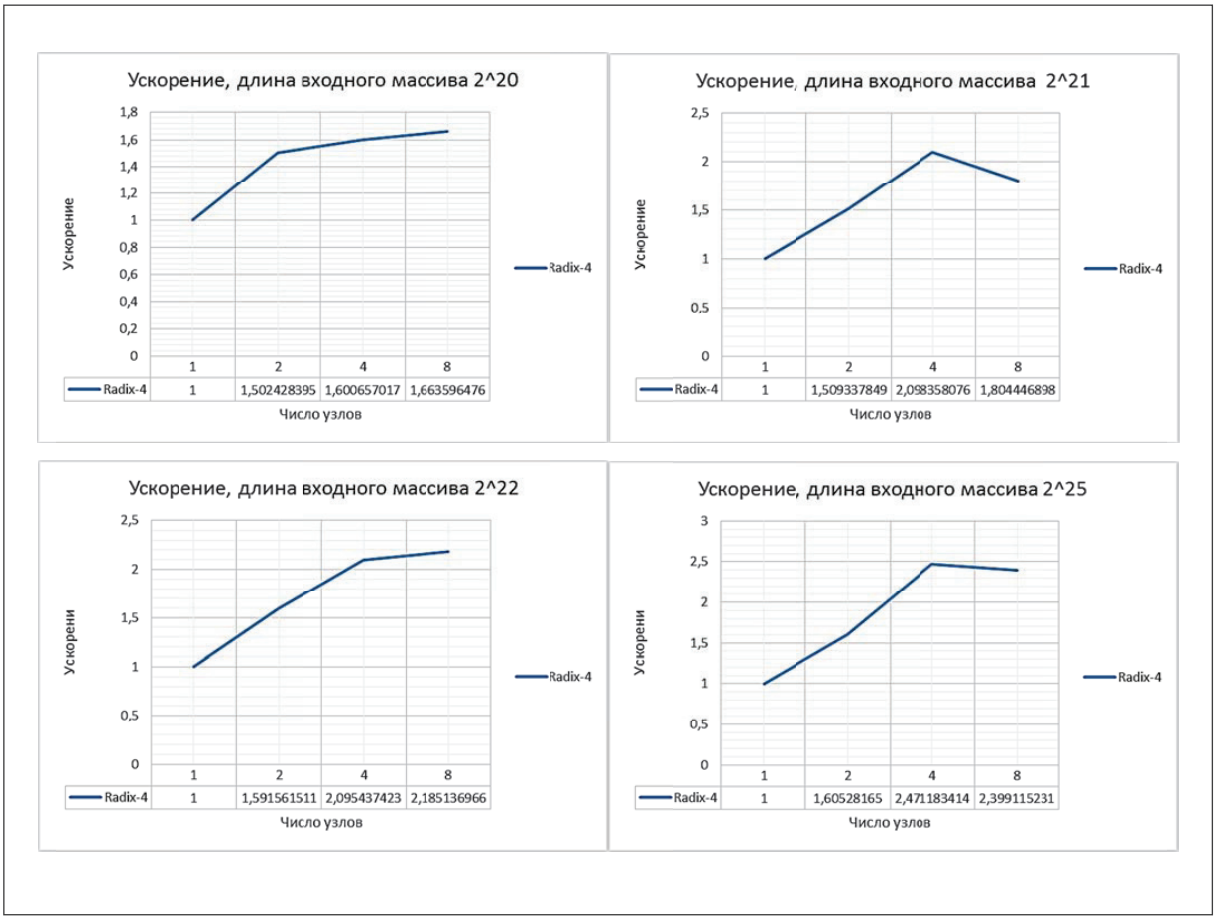


Рис. 4. Ускорение расчёта БПФ по алгоритму Radix-4 для набора случайных входных данных в зависимости от числа процессоров на кластере NWO5

Рис. 5а. Функция распределения времени расчёта БПФ по алгоритму Radix-4 для набора случайных входных данных с длиной 2^21 на 8 узлах кластера NWO5.

Рис. 5б. Функция распределения времени расчёта БПФ по алгоритму Radix-4 для набора случайных входных данных с длиной 2^25 на 8 узлах кластера NWO5.

Функция распределения времени для массива 2^25 явно показывает, что 20% всех времён попадают в отрезок [41.8; 43.9].

Про первый график столь же однозначно утверждать ничего нельзя, но на нём видно, что бóльшая часть измерений попадает в отрезок [3.2; 3.5]. Возможно, такое заметное падение ускорения на первом графике для восьми узлов происходит из-за того, что для такого числа процессоров приходится в конце алгоритма применять ещё преобразование бабочки из Radix-2, что может быть не выгодно для такого небольшого массива (в сравнении с массивами более длинными, для которых на восьми узлах ускорение больше, чем на четырёх).

Отличающиеся результаты (рис. 6) по ускорению расчёта получаются на вычислительном кластере К-100, т.к. этот кластер более современный и его характеристики не сильно отстают от мировых лидеров.

Как видно из этих графиков (рис. 6), ускорение находится в пределах от 1.5 до 4.5. Можно предположить, что на вычислительных системах, которые используются сегодня, ускорение будет также лежать между этими значениями.

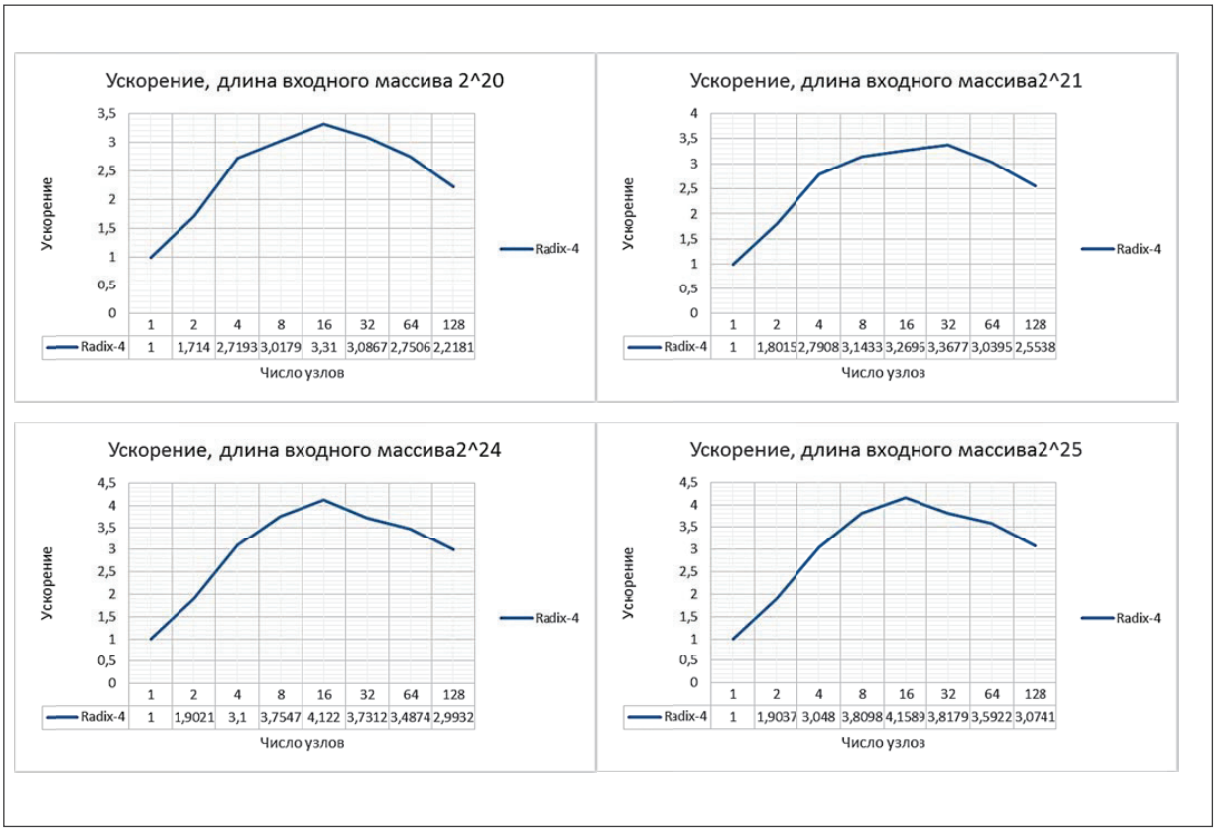


Рис. 6. Ускорение расчёта БПФ по алгоритму Radix-4 для набора случайных входных данных в зависимости от числа процессоров на кластере K-100.

4. Теоретический анализ ускорения параллельной программы

Для расчётов необходимо знать оптимальное число процессоров, при котором будет получено наибольшее ускорение. Это позволит использовать нашу параллельную программу с максимальной эффективностью.

Поскольку сложность БПФ есть $O(n \log_2 n)$, время работы программы на 1 процессоре $T_1 = Cn \log_2 n$, где C – некоторая константа, вычисляемая опытным путём.

Время вычисления на p процессорах T_p складывается из трёх слагаемых: времени, которое затрачивается на межпроцессорную коммуникацию, времени непосредственно параллельных вычислений и, наконец, времени объединения полученных ДПФ на нулевом узле. Время межпроцессорной коммуникации мы будем вычислять по формуле $t = a + \frac{m}{b}$, где a – латентность, b – пропускная способность, а m – величина передаваемого сообщения [5,1].

Для p узлов получаем

$$T_p = 2(p-1)(a + \frac{16n}{pb})k + C\frac{n}{p}\log_2\frac{n}{p} + Cn(\log_2 n - \log_2\frac{n}{p}).$$

Окончательная формула, таким образом, принимает следующий вид:

$$S_p = \frac{Cn\log_2 n}{2(p-1)(a + \frac{16n}{pb})k + C\frac{n}{p}\log_2\frac{n}{p} + Cn(\log_2 n - \log_2\frac{n}{p})} \quad (1)$$

Время межпроцессорной коммуникации в этой формуле мы умножаем на т. н. “поправочный” коэффициент k порядка единицы. Он необходим, поскольку мы используем слишком грубую оценку для времени коммуникации. Его мы найдём с помощью метода наименьших квадратов для каждой длины входного массива.

Значения k приведены в Таблице 1:

Таблица 1

Значение коэффициента k для разного объёма входных данных

Длина входного массива	Значение k
2^{20}	0.308
2^{21}	0.422
2^{22}	0.537
2^{23}	0.654
2^{24}	0.772
2^{25}	0.891

Из этой таблицы видно, что с увеличением длины массива, поправочный коэффициент приближается к единице. Следует ожидать того, что для массива определённой длины, он будет очень близок к единице. Правда, с ещё бóльшим увеличением объема входных данных он может продолжить расти.

Чтобы построить графики аппроксимаций (рис. 7), нам необходимо также знать характеристики вычислительной системы, на которой мы запускали нашу программу. Приведём их для кластера К-100:

1. Латентность: $a = 1.2 * 10^{-6}\ c$
2. Пропускная способность $b = 700 * 2^{20}$ байт/с

Кроме этого нам надо также знать значение константы C. Чтобы найти его, воспользуемся данными о времени работы программы на одном узле. Тогда $c = \frac{t}{n\log_2 n}$, где t – время работы программы, n – длина входного массива. Для кластера К-100 получаем $C = 2 * 10^{-8}$.

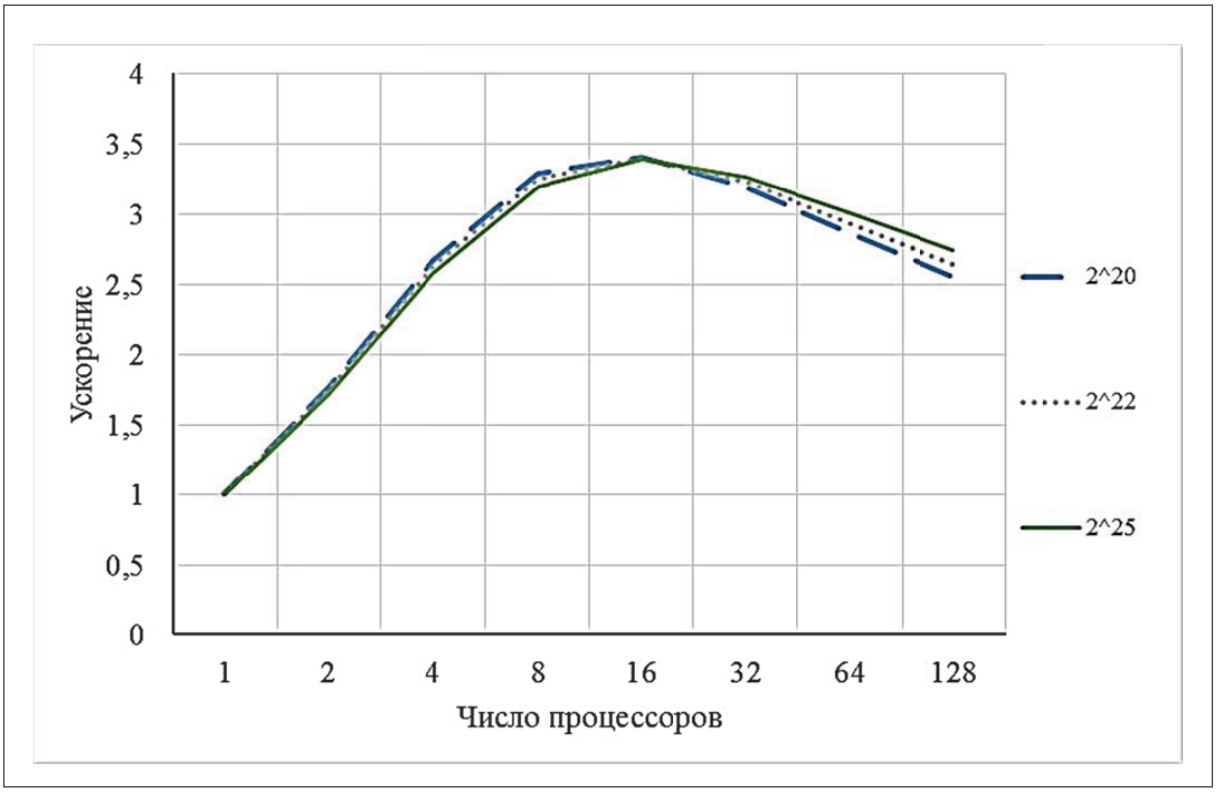


Рис. 7. Зависимость ускорения вычисления БПФ от числа процессоров согласно теоретической формуле с параметрами кластера К-100

Полученная формула (1) хорошо описывает зависимость ускорения расчёта БПФ от числа процессоров и справляется со своей задачей: её максимум действительно приходится на 16 узлов (рис. 7).

5. Заключение

Для задачи быстрого преобразования Фурье разработана и реализована версия алгоритма Radix-4 для многопроцессорных вычислительных систем на основе технологии MPI. Алгоритм протестирован на кластере nwo5 (ОИВТ РАН) и на кластере К-100 (ИПМ РАН). Зависимость ускорения расчёта от числа вычислительных узлов показала, что оптимальное число узлов для данного алгоритма лежит в диапазоне от 4 до 16 для длины входного массива в пределах от 2^{20} до 2^{25}. Построена теоретическая модель описывающая ускорение расчёта на многопроцессорных системах. Она оказалась в удовлетворительном согласии с результатами анализа расчётов. Эта формула позволит определять оптимальное число узлов для расчёта для каждой конкретной вычислительной системы.

Таким образом, данная параллельная реализация даёт ускорение от 1.5 до 4.5 раз как на современных, так и на более старых кластерах при использовании от 4 до 16

процессоров, если длина входного массива находится в пределах от 2^{20} до 2^{25} элементов. Увеличение числа процессоров ведёт к уменьшению ускорения расчёта и к увеличению времени расчёта.

Эти показатели ускорения не являются предельными. Время расчёта можно уменьшить за счёт:

1) оптимизации последовательной части программы (технология SIMD и др.) [4];
2) декомпозиции задачи для случая большого объёма входных данных (например, пока один процесс будет вычислять поворачивающие множители для дальнейших вычислений, все остальные будут расставлять элементы входного массива в нужном порядке);
3) оптимизации расходов на передачу информации между узлами с помощью подготовки подходящей топологии вычислительного кластера.

В процессе работы над статьёй использованы кластеры nwo5 (ОИВТ РАН) и K-100 (ИПМ РАН).

Литература

1. *Гергель В.П.* Теория и практика параллельных вычислений // БИНОМ, М., 2007, 424 стр.
2. *Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.* Introduction to algorithms. // MIT press, Cambridge, 2001, 640 pp.
3. *Chu E., George A.* Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms // CRC Press, 1999, 336pp.
4. *Postpischil E.* Construction of High Performance FFT // URL: http://edp.org/work/Construction.pdf, 2004, 93pp.
5. *Grama A., Gupta A., Karypis G., Kumar V.* Introduction to Parallel Computing // Addison Wesley, 2003, 612pp.
6. *Stasiński R., Potrymajło J.* Mixed-Radix FFT for improving cache performance. // Доклад на конференции «ESIPCO'04», 2004, 1525-1528.

EFFICIENCY OF FAST FOURIER TRANSFORM PARALLEL ALGORITHM RADIX-4

A.A. Morozov[1], A.V. Timofeev[2,1]

[1]*Department of Applied Mathematics,*
National Research Univercity "Higher School of Economics"

[2]*JIHT RAS, Moscow, Russia*

timofeevalvl@gmail.com

Received 03.08.2015

Parallel program for the fast Fourier transform is implemented on the basis of MPI (Message Passing Interface) technology. Radix-4 algorithm is chosen as a basic method to use. The dependence of parallel calculation acceleration on the number of processors is studied for two supercomputers. The formula describing the dependence of the calculation time on the number of processors is proposed for the range of the input data volume and supercomputer characteristics. The number of nodes providing you with maximum acceleration of calculation is estimated.