УДК 004.4242

МЕТОДЫ И ПРОГРАММНЫЕ СРЕДСТВА МАКРОМОДУЛЬНОЙ РАЗРАБОТКИ ПРОГРАММ

В.П. Гергель, А.А. Сиднев

Нижегородский госуниверситет им. Н.И. Лобачевского

alexey.sidnev@itlab.unn.ru

Поступила в редакцию 25.09.2012

Библиотечный подход является общепризнанным при разработке ПО. На данный момент существует множество библиотек для решения широкого круга задач, но отсутствие стандартов на интерфейсы этих библиотек приводит к сложности их использования за счёт изучения уникальных особенностей каждой библиотеки и высокой трудоёмкости миграции на новые библиотеки. Как следствие, усложняется разработка ПО, поддерживающего большое количество программно-аппаратных платформ. Возможное решение перечисленных проблем может состоять в применении излагаемого в работе макромодульного подхода к разработке программ.

Ключевые слова: макромодульный подход, стандартизация, модули, адаптеры для библиотек.

Введение

Разработка программного обеспечения является трудоёмкой профессиональной деятельностью [13, 14]. Для снижения сложности этого процесса действенным подходом является разложение задачи на более мелкие и простые подзадачи. Реализация подзадач выполняется в виде отдельных вычислительных блоков (модулей). Разработанные модули в дальнейшем могут быть использованы повторно, будучи оформленными в виде отдельных библиотек. Модульный подход к разработке программного обеспечения является на данный момент общепризнанным.

Одним из нерешённых вопросов в модульном подходе является отсутствие стандартов на интерфейсы модулей. Разработчик библиотеки сам определяет удобные ему структуры хранения данных и интерфейсы функций, которые их обрабатывают. В результате каждая библиотека получается уникальной и возникает сложность, связанная с заменой используемых модулей (библиотек).

Другая проблема при использовании библиотек – выбор наиболее оптимальной библиотеки под текущие задачи проекта. Для решения широкого круга задач существует достаточно много библиотек, каждая из которых может иметь свою сложность внедрения, эффективность реализации, удобство использования структур данных и поддержку различных программно-аппаратных платформ. Наличие большого количества библиотек приводит к необходимости выбора наиболее подходящей из них. Задача выбора библиотеки часто является многокритериальной, в связи с чем приходится идти на компромисс, выбирая некоторое «среднее» решение, теряя в эффективности реализации, удобстве использования или количестве поддерживаемых программно-аппаратных платформ. Кроме того, со временем возможностей текущей библиотеки под решение очередной задачи может не хватать. Тогда разработчики могут столкнуться с задачей перехода на другую библиотеку. Можно выделить следующие причины такого перехода:

– переход на другую программно-аппаратную платформу;

– повышение эффективности;

– увеличение функциональных возможностей;

– отказ от использования текущей версии библиотеки в связи с прекращением её поддержки.

При переходе к использованию новой библиотеки разработчику придётся столкнуться с необходимостью выполнить модификацию разработанных структур данных и функций под те, которые используются в библиотеке. Такой переход может быть очень трудоёмким.

Итак, классическая модульная разработка программного обеспечения имеет следующие недостатки:

– сильная зависимость от конкретной реализации библиотеки;

– проблема выбора наиболее оптимальной библиотеки под текущие задачи проекта или

поддержка нескольких библиотек (несколько версий кода);

– проблема миграции на новую библиотеку.

В работе предлагается макромодульный подход к разработке программ, позволяющий снизить остроту перечисленных проблем.

1. Общая характеристика подхода

Будем рассматривать класс вычислительных библиотек. Суть макромодульного подхода можно сформулировать в виде следующих основных положений.

1. Для библиотек, предназначенных для решения выбранного класса задач, должны быть разработаны стандартные интерфейсы.

2. Реализации библиотеки должны предоставлять интерфейсы, соответствующие принятым стандартам.

3. Программы, разрабатываемые с использованием некоторой библиотеки, должны обращаться к ней через стандартные интерфейсы.

4. Среди всех реализаций библиотеки должна использоваться наиболее эффективная для текущей программно-аппаратной платформы.

Хорошим примером предлагаемого подхода является интерфейс передачи сообщений MPI (Message Passing Interface) [1]. Для операций данного интерфейса разработан общепризнанный международный стандарт [2]. К текущему моменту разработано множество реализаций, поддерживающих этот стандарт, например: MPICH [3], Intel MPI [4], Open MPI [5] и другие. Если программа разработана с соблюдением стандарта MPI, то она может быть собрана с использованием любой из перечисленных реализаций. При этом переход с одной реализации на другую заключается в простой замене статических библиотек при сборке программы. Таким образом, разрабатывая параллельную программу с использованием интерфейса MPI, программист не должен задумываться о предварительном выборе библиотеки MPI. Этот выбор может быть сделан позднее, на этапе получения исполняемой программы (и этот выбор может быть легко изменен при смене платформы выполнения или появлении новых более эффективных реализаций). Программа будет работать с любой реализацией, которая поддерживает стандарт MPI. При этом, однако, следует отметить, что выбор конкретной реализации библиотеки должен осуществляться непосредственно программистом.

Однако пример с MPI является исключением из общей тенденции отсутствия стандартов. Во многих других областях есть либо стандарты де-факто, т.е. подходы, ставшие общепринятыми, либо просто соглашения об интерфейсах.

Рассмотрим одну из базовых операций линейной алгебры – задачу матричного умножения. Матричное умножение описано на третьем уровне стандарта BLAS (Basic Linear Algebra Subprograms) [9], но многие библиотеки отходят от этого стандарта и реализуют свои интерфейсы. Реализация BLAS представлена в большом количестве библиотек, рассмотрим две из них: MKL [10] и cuBLAS [11]. Программа, написанная автором, выполняет умножение матриц с использованием библиотеки MKL (функция **`sgemm`**) на языке C/C++ и занимает 49 строк кода[1].

Для того чтобы переписать программу с использованием библиотеки cuBLAS, потребовалось выполнить следующие действия:

– заменить статические библиотеки, используемые при линковке программы;

– заменить заголовочные файлы;

– заменить прототип и имя функции, которая выполняет умножение матриц;

– добавить код для выделения, освобождения, копирования памяти на графическом процессоре.

Полученная программа, выполняющая умножение матриц с использованием библиотеки cuBLAS, занимает 68 строк кода. В общей сложности переписывание программы с одной библиотеки на другую заняло у автора час времени (изучение библиотеки cuBLAS включено в это время), а текст программы увеличился на 19 строк.

Таким образом, использование библиотек, реализующих стандарт BLAS, приводит к необходимости изучения конкретной библиотеки и особенностей её реализации, что отрицательно сказывается на эффективности разработки ПО.

Рассмотрим еще один пример – задачу вычисления быстрого преобразования Фурье (БПФ) [12]. Библиотека FFTW является одной из самых эффективных бесплатных библиотек[2], которая содержит реализацию алгоритмов БПФ. Для работы с последней версией библиотеки используется интерфейс FFTW 3.x. Библиотека MKL поддерживает этот интерфейс, но для того, чтобы перейти в программе от использования библиотеки FFTW к MKL необходимо скомпилировать обёртку (статическая библио-

[1] Программа выполняет разбор аргументов командной строки, генерацию случайным образом исходных матриц, умножение матриц и печатает на экран время, затраченное на умножение матриц.

[2] FFTW распространяется под лицензией GNU GPL.

тека) под этот интерфейс либо использовать собственные интерфейсы MKL [6] (в обоих случаях также необходимо заменить статические библиотеки и заголовочные файлы).

Таким образом, можно сделать вывод, что на практике поддержка рассмотренных четырёх положений во многих областях либо отсутствует, либо неполна.

Обеспечить реализации существующих библиотек, решающих задачи одного класса, единым интерфейсом (даже при наличии стандарта) – тяжелая задача. Во многих случаях вычислительные библиотеки разработаны вообще без стандарта – каждая библиотека получается уникальной. Для решения этой проблемы предлагается разрабатывать программы-переходники (*адаптеры*) для каждой библиотеки, которые обеспечивают «стыковку» стандартных и используемых в библиотеке интерфейсов. Таким образом, модули программных библиотек должны либо разрабатываться при строгом соблюдении стандартных интерфейсов, либо библиотеки должны содержать *адаптеры*.

Описания действий, необходимых для подготовки вызова модуля в соответствии со стандартным интерфейсом, выполняется с помощью *макроописаний* (эти описания могут выноситься в метафайл с описанием программы).

Для выбора наиболее эффективной реализации для текущей программно-аппаратной платформы среда выполнения вычислительной системы должна содержать *планировщик*, определяющий для вызываемых модулей по стандартному описанию их наилучшие реализации (и в каких библиотеках).

2.1. Сборка проекта. Пусть у разработчика имеется программный код, использующий операции из некоторой вычислительной библиотеки. Необходимо решить задачу выбора конкретной реализации библиотеки из доступных разработчику. С использованием *макромодульного подхода* для этого потребуется выполнить следующие шаги (рис. 2).

1. Выполнить *макроописание* участков кода программы с указанием выполняемых действий и аргументов, использующихся в программном блоке.

Описание выполняется с помощью специальных директив препроцессора, предназначенных для расширения возможностей языка C/C++ (директивы pragma). Все макроописания начинаются с *директивы MMT*: «**`#pragma mmt`**». Такие директивы по умолчанию игнорируются компиляторами и не оказывают влияния на сборку программы. Если *макроописание* достаточно большое, то его содержательная часть может быть вынесена в отдельный файл.

2. Обработать исходные коды с *макроописаниями MM-препроцессором*, который генерирует программный код, передаваемый далее на вход стандартному компилятору языка C/C++.

3. Собрать проект стандартными средствами компиляции и линковки.

При использовании среды разработки Microsoft Visual Studio процесс компиляции и линковки (шаги 2 и 3) автоматизирован за счёт добавления предварительного этапа сборки проекта[3] (использование *MM-препроцессора* будет одним из этапов предварительной обработки исходных кодов).

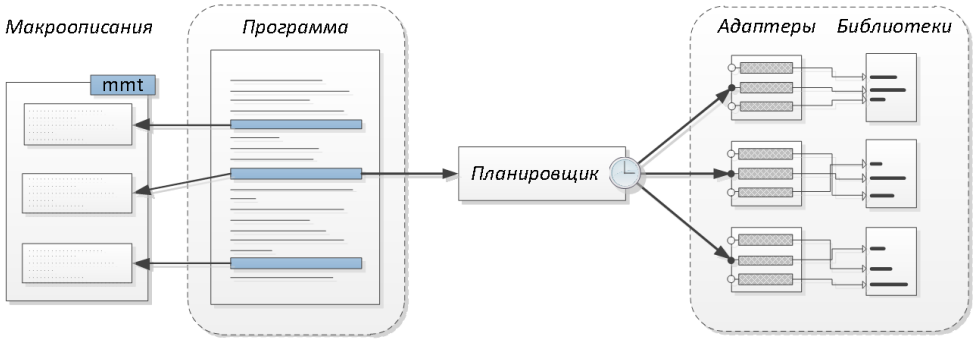

Рис. 1. Схема макромодульного подхода

2. Реализация

Рассмотрим последовательность сборки проекта, использующего *макромодульный подход*, и последовательность исполнения программы, написанной с использованием *макромодульного подхода*.

В качестве примера приведем макроописание умножения плотных матриц типа **`double`** (**`RowStoredMatrix`** – указывает на то, что матрица хранится по строкам – стандартный способ хранения матриц на языке C/C++).

[3] С помощью платформы сборки проекта MSBuild [12].

```
  #pragma mmt MMult(RowStoredMatrix
<double> (rowA, colA) matrA,

RowStoredMatrix<double>(rowB,     colB)
matrB,

RowStoredMatrix<double>(rowA,     colB)
result)
```

Программный код, содержащий *макроописание*, и блок кода, к которому оно применяется, будем называть *макроблоком*.

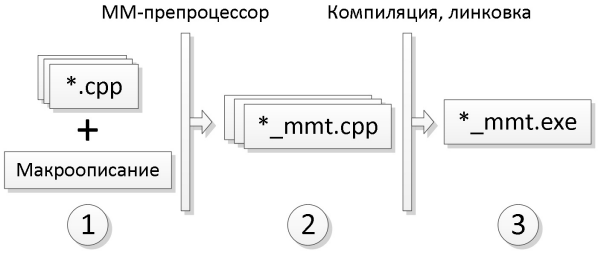

Рис. 2. Сборка проекта при использовании макромодульного подхода

ММ-препроцессор выполняет поиск *макроописаний* в программном коде и их анализ на корректность. Если в процессе анализа обнаруживается ошибка, то пользователю выдаётся соответствующее предупреждение. Если на этапе анализа ошибок не обнаружено, выполняется генерация программного кода с вызовом функции времени выполнения **MMKernel**, которая принимает все параметры, указанные в *макроописании* (рис. 3).

```
                MMType(matrA,
"RowStoredMatrix",   "double",   rowA,
colA),
                MMType(matrB,
"RowStoredMatrix",   "double",   rowB,
colB),
                MMType(result,
"RowStoredMatrix",   "double",   rowA,
colB)) )
      <Блок кода>;
```

Макромодульный подход может быть применим как при разработке новой программы, когда у разработчика отсутствует собственная реализация алгоритма, так и при модификации уже существующей. В случае отсутствия реализации алгоритма **<Блок кода>**, к которому применяется *макроописание*, может быть пустым. Тогда при выполнении программы на компьютере обязательно должна быть установлена хотя бы одна библиотека, из которой можно вызвать функцию, реализующую требуемый алгоритм.

2.2. Выполнение программы. Программный код функции **MMKernel** будем называть *ММТ-ядро*. Выполнение *ММТ-ядра* состоит из следующих этапов (рис. 4).

1. Определение библиотеки, которая будет использоваться для реализации алгоритма, соответствующего макроописанию.
2. Преобразование входных данных к формату, использующемуся в библиотеке (выполняется конвертером).

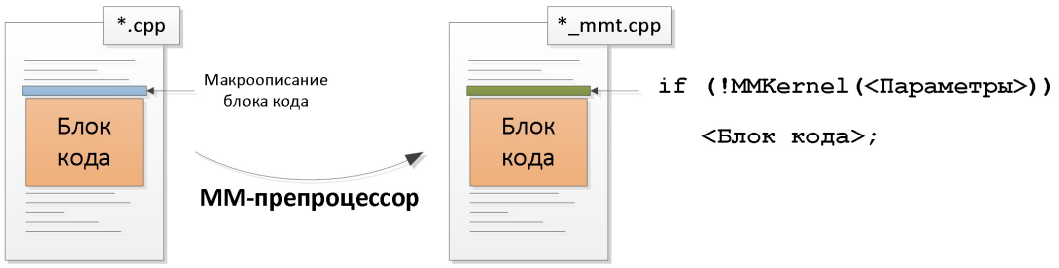

Рис. 3. Преобразования, выполняемые ММ-препроцессором

Функция **MMKernel** возвращает **true**, если действие, соответствующее *макроописанию*, было выполнено успешно и **false**, если действие не было выполнено (не найдена требуемая библиотека, не поддерживаются указанные типы данных). В последнем случае выполняется программный код, написанный пользователем.

Пример вызова **MMKernel** для умножения плотных матриц типа **double**.

```
  if (!MMKernel( MMFunction("MMult"),
```

3. Вызов реализации алгоритма из выбранной библиотеки (выполняется адаптером).
4. Преобразование выходных данных к формату, использующемуся в программе пользователя (выполняется конвертером).

2.2.1. Выбор библиотеки. Определение используемой библиотеки возможно двумя способами:

– явное указание используемой библиотеки в конфигурационном файле;

– автоматическое определение наиболее подходящей библиотеки.

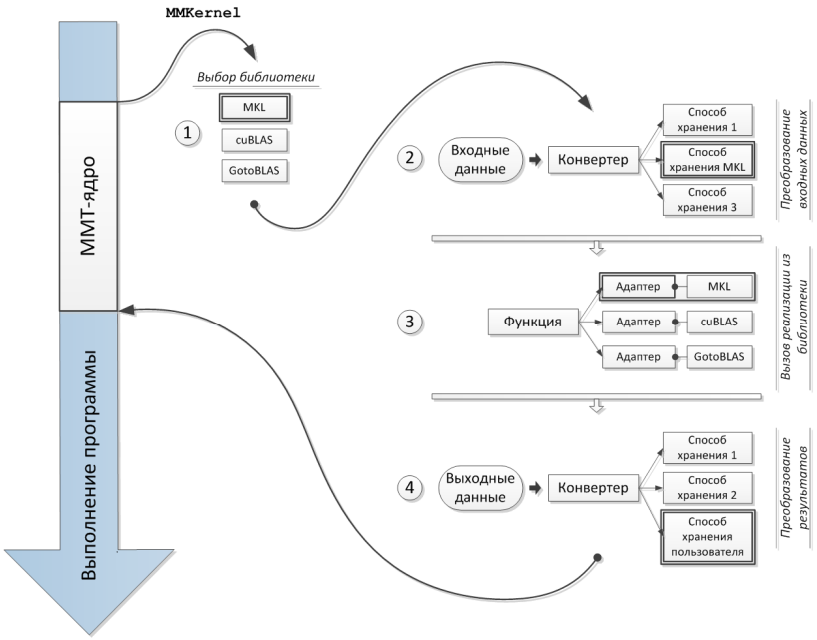

Рис. 4. Выполнение программы, содержащей вызов ММТ-ядра, на примере использования библиотеки MKL

Если выбранную библиотеку нельзя использовать (библиотека не установлена, требуемая функция не реализована в библиотеке), то выполняется программный код, написанный пользователем.

Для автоматического определения наиболее подходящей библиотеки необходимо выполнить оценку эффективности алгоритмов, реализованных в этих библиотеках, одним из следующих способов:

– предварительное тестирование вычислительной системы;

– профилирующий запуск программы;

– построение аналитических оценок.

С помощью предварительного тестирования вычислительной системы выполняется оценка эффективности библиотек на машине пользователя. Такое тестирование необходимо, т.к. время выполнения алгоритмов в библиотеках зависит от конфигурации вычислительной системы (тип и модель центрального процессора, тип и модель графического процессора, пропускная способность памяти и др.). Для каждой функции из *адаптера* выполняется ряд тестов с различными размерами входных данных. Результаты тестирования записываются в таблицы, которые содержат времена выполнения алгоритмов для выбранных входных параметров. Диапазон использующихся данных может быть указан пользователем перед тестированием. Подобное тестирование может занимать продолжительное время и выполняется один раз, если на компьютере пользователя установлены все используемые библиотеки. В случае установки новой библиотеки, выполняется тестирование только этой библиотеки.

Полученные после тестирования таблицы используются в момент выполнения ММТ-ядра для оценки эффективности библиотек и выбора лучшей реализации.

Профилирующий запуск программы позволяет собирать информацию о временах выполнения только тех функций из библиотек, которые используются в программе пользователя. При выполнении каждого ММТ-ядра вызываются функции из всех установленных библиотек. Таким образом, выполняется оценка эффективности использования библиотек для каждого *макроблока* программы. Профилирующие запуски могут выполняться пользователем несколько раз (для разных входных данных). Во всех последующих запусках данные профилирования используются для автоматического выбора наиболее эффективной библиотеки.

Аналитический метод предполагает оценку эффективности реализации из библиотеки за счёт построения модели вычислительной системы. Подобные модели являются не совсем точными [7].

2.2.2. Конвертер. Для каждого алгоритма разработчики выбирают наиболее подходящий способ хранения структур данных. Одни и те же данные можно хранить множеством различны-

ми способами. Например, двухмерные матрицы можно хранить следующими наиболее часто используемыми способами:

– по строкам – стандартный способ хранения матриц на языке C/C++;

– по столбцам – стандартный способ хранения матриц на языке Fortran;

– в виде массива строк (столбцов);

– в разреженном формате CRS, CSC.

Конвертер предназначен для преобразования данных из одного формата хранения в другое. Это необходимо для того, чтобы подготовить исходные данные в формате, использующемся в выбранной библиотеке, и преобразовать результаты в формат, который использует разработчик программы.

Необходимость конвертации определяется форматом хранения, который используется в *адаптере*.

2.2.2. Адаптер. Предназначен для вызова требуемой функции из библиотеки. Адаптер выполняет следующую последовательность действий:

– инициализация библиотеки;

– вызов функции или набора функций, которые реализуют требуемый алгоритм.

– освобождение ресурсов библиотеки.

Адаптер может иметь реализацию как для библиотеки, установленной на локальной машине пользователя, так и на удалённой машине.

3. Примеры использования макромодульного подхода

Рассмотрим несколько примеров использования макромодульного подхода при разработке программ.

3.1. Умножение матриц. Для большого числа матричных операций существует стандарт BLAS (базовые подпрограммы линейной алгебры), так операция матричного умножения определена с помощью функции **xGEMM** [8] (General Matrix Multiply). Первая буква определяет тип элементов матрицы:

S – вещественные числа одинарной точности,

D – вещественные числа двойной точности,

C – комплексные числа одинарной точности,

Z – комплексные числа двойной точности.

Функция **xGEMM** выполняет следующую операцию:

$$\mathbf{C} = \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C},$$

где **A**, **B** и **C** – матрицы, а α и β – скаляры. Для матриц **A** и **B** можно указать предварительную обработку перед умножением[4]: транспонирование или вычисление сопряжённой матрицы. Матрицы должны храниться по строкам или столбцам непрерывным блоком памяти.

Макроописание матричного умножения может выглядеть следующим образом:

```
#pragma mmt GEMM(<Формат хранения матрицы> A, <Формат хранения матрицы> B,
                 <Тип скаляра> alpha, <Тип скаляра> betta,
                 <Формат хранения матрицы> C)
```

В формате хранения матриц можно указать необходимость предварительного транспонирования или вычисления сопряжённой матрицы. Так же можно использовать сокращённую форму записи матричного умножения, которая определяет только умножение матриц (**C** = = **A**·**B**):

```
#pragma mmt GEMM(<Формат хранения матрицы> A, <Формат хранения матрицы> B,
                 <Формат хранения матрицы> C)
```

3.2. Быстрое преобразование Фурье. Быстрое преобразование Фурье – широко применяемый алгоритм, использующийся в задачах повышения качества изображений, создания спецэффектов, обработки изображений (выделение объектов, поиск лиц, улыбок) и др. БПФ определено для массива комплексных чисел. Выделяют прямое и обратное преобразование Фурье. При этом существуют функции, которые работают с массивом действительных чисел, так как на практике часто приходится работать именно с такими данными. *Макроописание* преобразования Фурье может выглядеть следующим образом:

```
#pragma mmt FFT(<Направление>,
                <Формат хранения массива> IN, <Формат хранения массива> OUT)
```

<Направление> определяет прямое или обратное преобразование Фурье и принимает одно из двух значений: FORWARD, BACKWARD. Формат хранения входного и результирующего массивов определяют, какое использовать БПФ (одномерное, многомерное).

[4] Задаётся в параметрах функции **xGEMM**.

Заключение

В работе рассмотрен *макромодульный подход* для разработки программ. Этот подход позволяет упростить разработку программного обеспечения за счёт сокращения затрат на внедрение библиотек. При *макромодульной* разработке выполняется описание программы с использованием набора стандартных действий (таких как «Умножение матриц», «Вычисление быстрого преобразования Фурье», «Решение СЛАУ») и стандартных структур хранения, которые участвуют в операциях. На основании подобного описания выполняется вызов соответствующих функций из библиотек через специальный *адаптер,* уникальный для каждой библиотеки. В том случае, если структуры хранения данных в программе и библиотеке отличаются, с помощью *конвертера* выполняется их преобразование. Выбор используемой библиотеки осуществляется вручную или автоматически из установленных библиотек. В результате решается проблема с выбором наиболее оптимальной библиотеки, а переход на использование новой значительно упрощается.

Макромодульный подход может применяться как при разработке новых программ, так и для модификации уже реализованного пользователем приложения (например, для перехода на новую программно-аппаратную платформу). В обоих случаях разработчику достаточно сделать *макроописание* вычислительных модулей независимо от того, существует ли их реализация.

Результаты докладывались на Всероссийской молодежной школе, проводимой в Нижегородском университете.

Работа выполнена при финансовой поддержке Минобрнауки России, государственное соглашение о предоставлении гранта № 14.B37.21.0878.

Список литературы

1. MPI: The Message Passing Interface. [http://parallel.ru/tech/tech_dev/mpi.html].

2. Message Passing Interface Forum. [http://www.mpi-forum.org/].

3. MPICH-A Portable Implementation of MPI. [http:// www.mcs.anl.gov/research/projects/mpi/mpich1-old/].

4. Intel® MPI Library. Message Passing Interface Library. [http://software.intel.com/en-us/articles/intel-mpi-library/].

5. Open MPI: Open Source High Performance Computing. [http://www.open-mpi.org/].

6. The Intel® Math Kernel Library and its Fast Fourier Transform Routines. [http://software.intel.com/en-us/articles/the-intel-math-kernel-library-and-its-fast-fourier-transform-routines/].

7. Lizy K. John. Performance Evaluation: Techniques, Tools and Benchmarks. Computer Engineering Handbook. CRC Press. 2002. [http://lca.ece.utexas. edu/pubs/john_perfeval.pdf].

8. Киреев С. Пример использования процедуры xGEMM из библиотеки BLAS в программе на Си. [http://ccfit.nsu.ru/~kireev/lab4/lab4blas.htm].

9. BLAS (Basic Linear Algebra Subprograms). [http://www.netlib.org/blas/].

10. Intel® Math Kernel Library (Intel® MKL). [http://software.intel.com/en-us/articles/intel-mkl/].

11. CUBLAS. [http://developer.nvidia.com/cublas].

12. Нуссбаумер Г. Быстрое преобразование Фурье и алгоритмы вычисления сверток: Пер. с англ. М.: Радио и связь, 1985. 248 с.

13. Гергель В.П., Стронгин Р.Г. Опыт Нижегородского университета по подготовке специалистов в области суперкомпьютерных технологий // Вестник Нижегородского университета им. Н.И. Лобачевского. 2010. № 3 (1). С. 191–199.

14. Баркалов К.А., Гергель В.П., Гергель А.В., Мееров И.Б., Сысоев А.В. Организация и проведение Всероссийской школы по суперкомпьютерным технологиям // Открытое и дистанционное образование. 2010. № 2. С. 24–29.

METHODS AND SOFTWARE OF MACROMODULE PROGRAMMING

V.P. Gergel, A.A. Sidnev

Creating libraries is a universally recognized approach to software development. At present there are a lot of libraries for a wide range of tasks, but the lack of library interface standards leads to complexities in the usage, integration and migration between different implementations. As a result, the development of cross-platform software capable of adapting libraries to each other becomes a complicated problem. In this paper, we describe methods and software of macromodule programming as a possible solution to the problem of interlibrary relations.

Keywords: macromodule approach, standardization, modules, library adapters.