REDUCTION OF INVOCATION OVERHEAD IN AUTOMATICALLY GENERATED PROGRAMS WITH THE ACTIVE KNOWLEDGE CONCEPT

V. E. Malyshkin, V. A. Perepelkin, Yu. Yu. Nushtaev*,**

*Institute of computational mathematics and mathematical geophysics SB RAS,
630090, Novosibirsk, Russia
**Novosibirsk State University,
630090, Novosibirsk, Russia
***Novosibirsk State Technical University,
630073, Novosibirsk, Russia

DOI: 10.24412/2073-0667-2025-3-34-51
EDN: CBKGZK

Parallel programs development automation is a relevant research direction, potentially beneficial in multiple ways. It allows to reduce complexity and labor intensity for human, improve efficiency of constructed programs and support software and algorithms accumulation and reuse. One of the problems here is to reduce the invocation overhead which arises from the fact that in practice programs have to be constructed mostly out of modules. This fact implies modules unification and overhead, related to their invocation, data transfer, run-time environment setup, etc. The overhead significantly affects the constructed program efficiency (i.e. program execution time, memory consumption, network load, etc.), which is essential in high performance computing. Programs construction system capabilities in reduction of the overhead highly depend on the computational model employed by the system. In the work we consider the invocation overhead reduction problem through the active knowledge concept [10] — a methodology for efficient programs construction automation in particular subject domains. The concept is based on the theory of parallel programs and systems synthesis on the basis of computational models [11]. It implies that to perform automatic construction of efficient-enough programs in a particular subject domain one has to make a machine-oriented partial formal description of the subject domain called active knowledge base [9]. It contains description of various algorithms, related software modules and peculiarities of the subject domain. Based on active knowledge base it is possible to formulate a class of applied problems to solve and automatically construct a program to solve any of the problems. The key concept here is computational model, which for simplicity can be concerned as a bipartite directed graph of operations and variables vertices. Ingoing and outgoing arcs for particular operation vertex denote its input and output variables. Computational model describes a subject domain in sense that the domain has some variables and there is an ability to compute some variables from some other variables. Each operation can be given a suitable computational module, called code fragment, capable of computing values of its output variables from values of its input variables. Conventional subroutine of given form can serve as an example of a code fragment. The computational process then is concerned as follows. Some variables are assigned with arbitrary values. Any operation can be executed if all its input variables have values. Operation execution is code fragment invocation with values of input and output variables' values as input and output arguments.

This work was carried out under state contract with ICMMG SB RAS FWNM-2025-0005.

Operations are executed (maybe in parallel) until all variables marked as demanded are computed. The computational model can be employed for automatic programs construction. A constructed program consists of two parts. The first one is a set of code fragments contained in the active knowledge base. The second one is generated code, which can be called "glue" code. Its main purpose is to invoke code fragments, pass arguments to them, organize network data transfer and perform other similar tasks. To provide high efficiency of a constructed program the following two conditions have to be satisfied. Firstly, "glue" code has to be efficient. Secondly, the code fragments invocation overhead has to be low enough. For example, if a code fragment is a conventional subroutine, then its invocation requires control passing (call) and data movement between different memory locations and or registers. In conventional compilers this overhead can sometimes be reduced using the inlining technique. If a code fragment is a program written in another language, then corresponding run-time environment and data conversion has to be made. Notably, the inlining technique not always can be employed by the compiler because it relies on complex static code analysis. Unless the compiler is able to extract all necessary information to perform inlining it cannot be applied. An alternative approach is to manually provide code fragments with necessary metainformation. In such case invocation of the code fragment can be implemented not as a procedure call, but as an inline code snippet. Code snippet of particular form is an example of a code fragment with less overhead than a conventional procedure. The active knowledge concept supports this approach by allowing the inclusion of different code fragment types with necessary metainformation into active knowledge base. Another advantage the active knowledge concept suggests is automatic operations aggregation (batching). The idea behind this technique is to combine a group of similar operations into a single code fragment, thus reducing overhead. A practical example is aggregating multiple operations for GPU to reduce input/output data transfer between main memory and GPU memory. Provided necessary metainformation is given, multiple GPU operations can be aggregated into one GPU call. Such low-level techniques as CUDA Graph [20] can be applied automatically. Some subject domains have additional possibilities of batching. For example, cuFFT library provides an API to perform batch processing of multiple fast Fourier transforms more efficiently. With the active knowledge concept, it is possible to perform such batching automatically. For that an active knowledge base has to be supplied with corresponding metainformation and batching algorithm implementation. The system will be able to analyze the computational model graph in order to find operations to batch. In the paper we concern a practical example — automatic construction of a hybrid parallel program which uses both CPU and GPU to achieve satisfactory performance in seismic data processing [12].

Key words: active knowledge concept, computational model, automatic program construction.

References

1. Kale L. V., Krishnan S. Charm++ a portable concurrent object oriented system based on c++ //Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. — 1993. — S. 91-108.

2. Charm++. Parallel Computer Network [Electron. Res.]: `http://charmplusplus.org/`. (accessed: 01.05.2025).

3. OpenCL [Electron. Res.]: `https://www.khronos.org/opencl/` (accessed: 01.05.2025).

4. Coarray Fortran [Electron. Res.]: `http://caf.rice.edu` (accessed: 01.05.2025).

5. Reid J. Coarrays in the next fortran standard //ACM SIGPLAN Fortran Forum. New York, NY, USA : ACM, 2010. V. 29. N 2. P. 10–27.

6. DVM — sistema razrabotki parallel'nykh programm [Electron. Res.]: `http://dvm-system.org/ru/about/` (accessed: 01.05.2025).

7. Bakhtin V. A. [et al.]. Rasshireniye DVM-modeli parallel'nogo programmirovaniya dlya klasterov s geterogennymi uzlami // Vestnik Yuzhno-Ural'skogo universiteta. Chelyabinsk: Izdatel'skiy tsentr

YuUrGU, 2012. Seriya: Matematicheskoye modelirovaniye i programmirovaniye. N 18 (277). Vypusk 12. S. 82–92.

8. Kataev N., Kolganov A. The experience of using DVM and SAPFOR systems in semi automatic parallelization of an application for 3D modeling in geophysics // The Journal of Supercomputing. 2019. T. 75. N 12. S. 7833–7843.

9. Malyshkin V. E., Perepyolkin V. A. Postroenie baz aktivnyx znanij dlya avtomaticheskogo konstruirovaniya reshenij prikladnyx zadach na osnove sistemy LuNA // Parallelnye vychislitelnye texnologii — XVIII vserossijskaya nauchnaya konferenciya s mezhdunarodnym uchastiem, PaVT'2024, g. Chelyabinsk, 2–4 aprelya 2024 g. Korotkie statyi i opisaniya plakatov. Chelyabinsk: Izdatelskij centr YuUrGU, 2024. s. 57–68.

10. Victor Malyshkin. Active Knowledge, LuNA and Literacy for Oncoming Centuries. In Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security - Volume 9465. Springer-Verlag, Berlin, Heidelberg, 2015. p. 292–303.

11. Sintez parallelnykh programm i sistem na vychislitelnykh modelyakh / V. A. Valkovsky, V. E. Malyshkin; Onv. red. V. E. Kotov; AN SSSR, Sib. otd-nie, VC. Novosibirsk : Nauka. Sib. otd-nie, 1988. 126 s. (In Russian).

12. Vyrodov A. Yu. et al. Printsipy organizatsii programmno-analiticheskoy sistemy dlya parallel'noy obrabotki seysmicheskikh dannykh // Vestnik SibGUTI. 2024. T. 18. N 2. S. 57–68.

13. Ragan-Kelley J. et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines // Acm Sigplan Notices. 2013. T. 48. N 6. S. 519–530.

14. PLUTO [Electron. Res.]: `https://pluto-compiler.sourceforge.net/` (accessed: 01.03.2025).

15. Bondhugula U. et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model // Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings 17. Springer Berlin Heidelberg, 2008. S. 132–146.

16. Bondhugula U. et al. A practical automatic polyhedral parallelizer and locality optimizer // Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008. S. 101–113.

17. Polyhedral Compilation [Electron. Res.]: `http://polyhedral.info/` (accessed: 01.03.2025).

18. Malyshkin V. E., Perepelkin V. A. LuNA fragmented programming system, main functions and peculiarities of run-time subsystem // International Conference on Parallel Computing Technologies. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. S. 53–61.

19. Malyshkin V. E., Perepelkin V. A. Opredelenie ponyatiya programmy // "Problemy' informatiki", 2024, N 2, S. 16–31.

20. CUDA Graphs [Electron. Res.]: `https://developer.nvidia.com/blog/cuda-graphs/` (accessed: 01.05.2025).

21. NVIDIA. cuFFT Library [Electron. Res.]: `https://docs.nvidia.com/cuda/cufft/index.html` (accessed: 01.05.2025).

22. OpenMP [Electron. Res.]: `http://www.openmp.org/` (accessed: 01.03.2025).

23. NVIDIA CUDA [Electron. Res.]: `https://developer.nvidia.com/cuda-toolkit` (accessed: 01.05.2025).

24. Malyshkin V. Active Knowledge, LuNA and Literacy for Oncoming Centuries // In Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security. V. 9465. Springer-Verlag, Berlin, Heidelberg, 2015. P. 292–303.

УМЕНЬШЕНИЕ НАКЛАДНЫХ РАСХОДОВ НА ВЫЗОВ МОДУЛЕЙ В АВТОМАТИЧЕСКИ КОНСТРУИРУЕМЫХ ПРОГРАММАХ НА ОСНОВЕ КОНЦЕПЦИИ АКТИВНЫХ ЗНАНИЙ

В. Э. Малышкин, В. А. Перепелкин, Ю. Ю. Нуштаев*,**

*Институт вычислительной математики и математической геофизики СО РАН, 630090, Новосибирск, Россия
**Новосибирский национальный исследовательсций государственный университет, 630090, Новосибирск, Россия
***Новосибирский государственный технический университет, 630073, Новосибирск, Россия

УДК 004.4'242
DOI: 10.24412/2073-0667-2025-3-34-51
EDN: CBKGZK

Одной из проблем, возникающих при автоматическом конструировании параллельных программ, является проблема уменьшения «межмодульного трения» — накладных расходов на взаимодействие структурных элементов конструируемой программы (вызов подпрограмм, передачу аргументов, создание необходимого исполнительного окружения и т. п.). Эти накладные расходы в конструируемой программе существенно влияют на ее эффективность (время выполнения, расход памяти, нагрузка на сеть и т. п.). Возможности системы автоматического конструирования программ во многом зависят от модели вычислений, лежащей в основе ее входного языка. В статье этот вопрос рассматривается с позиций концепции активных знаний — методологии автоматизации конструирования программ в конкретных предметных областях. В частности, на примере задачи обработки сейсмических данных показывается, как на основе концепции активных знаний могут быть уменьшены накладные расходы на вызов модулей и автоматически реализованы такие техники оптимизации конструируемой программы как «монолитизация» — объединение нескольких структурных элементов программы в один с соответствующим снижением накладных расходов — за счет наличия формального описания свойств структурных элементов программы и машинно-ориентированного описания особенностей предметной области в виде базы активных знаний.

Ключевые слова: параллельное программирование, активные знания, системы автоматического конструирования программ, вычислительные модели, сейсмические сигналы.

Введение. Разработка программ — это сложный и трудоемкий процесс, требующий высокой квалификации. Высокая востребованность профессии программиста обуславливает потребность в автоматизации разработки программ для снижения трудоемкости и сложности разработки программ, повышения их качества и высвобождения людских ресурсов из этой сферы.

Исследования выполнены в рамках государственного задания ИВМиМГ СО РАН FWNM-2025-0005.

В общей постановке задача автоматического конструирования достаточно хорошей для практического использования программы является алгоритмически труднорешаемой, о чем свидетельствует отсутствие универсального решения этой проблемы и разнообразие различных частных и эвристических подходов к автоматическому конструированию программ [1–9]. В этой связи важно исследовать различные подходы к решению этой проблемы.

Для практического применения средств автоматического конструирования программ существенно, чтобы программы строились не из отдельных операций и переменных, а по возможности из уже сложившихся в ручном программировании крупных блоков — программных модулей. Без такого переиспользования существующего программного обеспечения (ПО) создание практически пригодных программ возможно только в узких нишах, т. к. без переиспользования существующего ПО алгоритмическая сложность задачи существенно выше.

Это, в свою очередь, требует некоторой унификации существующего программного обеспечения (ПО), приведение его к некоторому стандартному виду, чтобы обеспечить единообразную работу с ПО со стороны системы автоматизации. Например, включение кода в обычную библиотеку подпрограмм требует его оформления в виде процедуры, включение кода в виде модуля расширения (plugin) требует его оформления с соответствующим интерфейсом, и т. п. Программные пакеты (deb, rpm), контейнеры (docker), образы виртуальных машин — примеры различных модульных оболочек, обеспечивающих разные возможности автоматизации.

Каким бы ни был стандартный вид модуля, это создает «межмодульное трение» — накладные расходы на обращение к модулю, передачу ему аргументов, создание необходимого для него исполнительного окружения и т. п. Чем более универсальна модульная оболочка, тем выше межмодульное трение. Это обуславливает тот факт, что вместо универсальных интерфейсов программных модулей на практике используют частные механизмы, обеспечивающие, с одной стороны, приемлемую долю накладных расходов в своих предметных областях, но, с другой стороны, ограничивающие область возможного автоматического применения соответствующего ПО.

Сложности унификации представления ПО для автоматизации эффективного его переиспользования для решения новых задач в предметной области проистекают из того, что достижение эффективноси существенно зависит от особенностей этой предметной области, в частности, с того, как понимается эффективность в данной предметной области и какими методами возможно ее обеспечение в этой предметной области.

Концепция активных знаний [10] — это методология автоматизации конструирования программ, основанная на теории синтеза параллельных программ и систем на вычислительных моделях [11]. Эта методология предлагает подход к автоматическому построению достаточно хороших для практического применения программ в конкретной предметной области на основе построения базы активных знаний — частичного формального описания предметной области, накопленных в нем готовых решений (готового ПО) и сложившейся практики их эффективного применения. Наличие базы активных знаний позволяет автоматически строить решения новых задач в предметной области за счет того, что существенные особенности этой предметной области явно описаны в базе активных знаний. В частности, концепция активных знаний позволяет обеспечивать низкий уровень межмодульного трения, вплоть до несущественного. В статье рассматривается этот вопрос на примере конкретной задачи из предметной области сейсмического мониторинга [12].

Дальнейшая часть статьи организована следующим образом. В разделе 1 представлен краткий обзор ключевых подходов к решению рассматриваемой проблемы, в разделе 2 вводятся необходимые термины и рассматривается постановка задачи, в разделе 3 представлено описание генератора программ, основанного на предлагаемом подходе. В разделе 4 представлены результаты экспериментального исследования. Завершает статью заключение, где подводятся итоги работы.

1. Обзор родственных работ. Системы автоматического конструирования параллельных программ стремятся упростить разработку высокопроизводительных приложений, но часто сталкиваются с проблемой накладных расходов модульных оболочек. Эти накладные расходы могут существенно снижать эффективность параллельных программ, нивелируя преимущества автоматизации. Проведем анализ существующих систем и подходов с этой позиции.

Одним из подходов является использование предметно-ориентированных языков, таких как Halide [13]. Предметно-ориентированные языки (DSL) позволяют описывать некоторый класс задач на высоком уровне абстракции. Но их ограниченность рамками конкретной предметной области может потребовать усилий для интеграции с существующими программными компонентами, порождая накладные расходы, связанные с преобразованием данных и вызовами библиотек.

Другой подход заключается в автоматическом распараллеливании существующих последовательных программ, как это реализовано в системе PLUTO [14–16]. Эта система использует полиэдральную компиляцию (Polyhedral Compilation) [17] для оптимизации и распараллеливания циклов в C-коде. Это позволяет повторно использовать существующий код. Но эта система является узкоспециализированной и она не всегда обеспечивает достаточную производительность и, скорее, может рассматриваться как компонент какой-нибудь системы автоматического конструирования параллельных программ. Более гибкий подход предлагает система LuNA [18]. Эта система автоматически конструирует эффективные программы в конкретной предметной области, используя формальное описание области, накопленные решения и практику их применения. Но наличие исполнительной системы может вносить существенные накладные расходы. Такая же проблема существует у системы Charm++ [1–2] и ее модели на основе «чаров».

Такие подходы, как OpenCL [3] и Coarray Fortran [4–5], предоставляют разработчикам более низкоуровневые инструменты для параллельного программирования, но требуют вмешательства специалиста. При этом даже у таких подходов возникают накладные расходы, например у OpenCL из-за JIT-компиляции кода для специализированных устройств часто могут возникать накладные расходы во время выполнения программы. Примерно также обстоят дела у DVM [6–8].

Уменьшение накладных расходов модульных оболочек является одним из ключевых вопросов для достижения высокой производительности, и различные системы автоматического конструирования параллельных программ предлагают разные подходы к решению этой проблемы. Но на данный момент существующие решения не закрывают вопроса, поэтому актуально дальнейшее исследование подходов к уменьшению накладных расходов в различных предметных областях.

2. Постановка задачи. В концепции активных знаний [10], в соответствии с ее базовой теорией [11] процесс автоматического конструирования программы строится на основе баз активных знаний [9]. Ключевым элементом базы активных знаний является вычислительная модель (ВМ) [11], которую в статье мы упрощенно будем рассматривать как

двудольный орграф, вершинами которого являются операции и переменные. Переменные описывают величины предметной области и могут иметь значения произвольных типов данных. Операции обозначают возможность вычислять значения одних переменных из других (какие из каких — обозначается входящими в операцию и исходящими из нее дугами соответственно, и называются входными и выходными переменными операции соответственно). Обеспечивается эта возможность вычислять предоставлением фрагмента кода (ФК) — представленного в заранее оговоренной подходящей форме программного модуля, например последовательной процедуры с известной сигнатурой. ВМ, множество ФК, а также некоторые другие технические элементы и составляют базу активных знаний.

Процесс конструирования программы [19] на основе базы активных знаний начинается с того, что во множестве переменных ВМ выделяется два подмножества — V и W, называемые входными и выходными переменными соответственно. Значения переменных из V считаются известными, а значения переменных из W требуется вычислить. В этом случае говорят, что на ВМ поставлена VW-задача (или просто задача). Затем осуществляется планирование или вывод алгоритма — выделяется подмножество операций ВМ, упорядоченное исполнение которых (т. е. запуск ФК с соответствующими аргументами) приводит к вычислению значений переменных из W. Это подмножество операций задает алгоритм решения задачи.

Далее конструируется программа. Из различных возможных вариантов [19] мы рассмотрим один — генерируется листинг обычной последовательной или параллельной программы, содержание которой сводится к упорядоченному запуску ФК в соответствии с выведенным ранее алгоритмом решения задачи. Помимо собственно запуска, ФК сгенерированная программа может содержать код, обеспечивающий пересылку значений переменных по сети, управление памятью, синхронизацию доступа к данным, загрузку извне значений входных переменных и выдачу вовне значений вычисленных выходных значений переменных, а также другие вспомогательные элементы.

Таким образом, мы рассмотрели, как в своей основе рассматривается процесс конструирования программы в концепции активных знаний. В частности, видно, что «полезные» вычисления, ради которых и существует программа, находятся исключительно внутри ФК, а генерируемый код является «склеивающим слоем», обеспечивающим их запуск и подстановку аргументов. Обеспечение высокой производительности программы по большому счету, определяется следующими факторами: насколько мало ресурсов уходит на работу «склеивающего слоя»; насколько эффективно заложенное в него управление и распределение ресурсов; насколько «тонки оболочки» ФК. Первые два фактора выходят за рамки настоящей статьи. Здесь же рассмотрим третий фактор.

Если ФК — это последовательная процедура, то ее вызов сам по себе требует накладных расходов, таких как перемещение данных (регистры, стек) для передачи аргументов. Для маленьких по объему вычислений ФК доля накладных расходов может оказаться нецелесообразно высокой. Отчасти это может быть преодолено механизмом встраивания кода вместо вызова процедуры (inline). Если же ФК — это программный модуль на другом языке программирования, то накладные расходы могут многократно возрастать. И если часть из них объективно необходимы (например, передача данных из контекста одного языка в контекст другого), то часть возникает вследствие того, что внешний модуль становится «черным ящиком», о котором система конструирования программ не обладает достаточной информацией и, соответственно, работу с которым не может эффективно оптимизировать.

Например, если система может работать только с ФК вида «последовательная процедура с известной сигнатурой», то запуск кода в ином контексте (скажем, на GPU) возможен путем создания «оберточного» ФК (wrapper), который содержит работу с GPU внутри, о чем система не знает. Но это требует в каждом таком ФК перемещать данные из памяти CPU в память GPU и обратно, иначе сгенерированный код не будет работать правильно. Если же ввести в систему поддержку нового вида ФК — ФК для GPU, то система сможет сама в «склеивающем слое» сгенерировать корректную работу с данными и оптимизировать их перемещение (например, оставлять данные в памяти GPU, если вскоре они будут обработаны ФК на GPU).

Принципиально, что ни один вид ФК не может быть универсальным, т. к. это означало бы приведение всех программных модулей к одному виду, создание для всех программных модулей универсальной обертки. Ввиду разнообразия программных модулей универсальная обертка имела бы для подавляющего большинства модулей (особенно малых по объему вычислений) чрезмерно большие накладные расходы на передачу аргументов, создание необходимого исполнительного окружения и т. п. (для примера можно рассмотреть «универсальный» формат, такой как docker-контейнер или предустановленную виртуальную машину). Концепция активных знаний же предполагает наличие различных видов ФК на разные случаи жизни — процедуры, сниппеты, команды командной строки и т. п., причем новый вид ФК может быть добавлен в систему через введение соответствующего модуля расширения в базу активных знаний. Таким образом включение программных модулей в базу активных знаний в любой предметной области становится практичным, а конкретный набор используемых видов ФК зависит от предметной области.

Отметим также, что одной из полезных функций модульности является инкапсуляция модулей, что обеспечивает пользователям модуля (в т. ч. системе конструирования) не знать многие детали внутреннего устройства модуля. Так, например, процедура как модульная оболочка позволяет не заботиться о том, какие локальные переменные использует процедура. Снятие процедурной оболочки с кода (с целью уменьшения накладных расходов) и встраивание самого кода в листинг программы означает, что необходимо исключить возможность конфликта имен переменных во встраиваемом и объемлющем коде. Этот пример иллюстрирует идею о том, что чем «тоньше» модульная оболочка, тем больше информации должна иметь система о ФК для корректного и эффективного его использования, но это дает возможность снижать межмодульное трение.

Вернемся к примеру с GPU. На основе концепции активных знаний возможны и более продвинутые техники оптимизации, такие как использование CUDA Graph [20] или cuFFTplan [21]. Первая техника позволяет формировать пакет задач для обработки на видеокарте без необходимости промежуточной синхронизации с CPU, а вторая реализует пакетную обработку набора задач по вычислению быстрого преобразования Фурье (БПФ) в библиотеке cuFFT.

Эти и другие техники оптимизации исполнения множества операций непосредственно относятся к третьему рассматриваемому фактору — «толщине оболочек» ФК. Обычно применение этих техник возможно только вручную, т. к. корректное их применение требует информации, которую практически невозможно извлекать автоматически из традиционного кода, и этой информацией обладает только программист. В концепции активных знаний эта информация может быть непосредственно включена в базу активных знаний вручную, что позволяет применять подобные техники автоматически.

Отметим, что такая оптимизация как объединение множества операций БПФ в одну пакетную операцию является примером оптимизации, специфичной для конкретной предметной области, и не имеет широкого применения в программах общего назначения. Другие подобные техники оптимизации также могут быть узкоспециализированными. Но для концепции активных знаний поддержка таких оптимизаций не является проблемой, в отличие от систем общего назначения, т. к. база активных знаний как раз и является частичным формальным описанием конкретной предметной области, где есть возможность такие узкоспециализированные оптимизации закладывать. Адекватность таких техник контролирует инженер знаний, составляющий базу активных знаний и являющийся специалистом как в предметной области, так и в автоматизации конструирования программ.

В частности, автоматическая агрегация множества операций БФП в одну пакетную операцию может быть реализована как промежуточный этап конструирования программы, предшествующий выводу алгоритма. Суть его сводится к тому, что система рассматривает граф ВМ, выявляет множество операций, связанных с конкретным ФК — БПФ на GPU, анализирует их аргументы (переменные) на возможность упаковки, и в случае такой возможности добавляет в ВМ новую операцию, входные и выходные переменные которой являются объединением входных и выходных переменных агрегируемых операций соответственно, а ФК для новой операции является ФК для GPU на основе формирования структуры cuFFTPlan библиотеки cuFFT и пакетной обработки набора БПФ. Поддержка таких способов оптимизации конструируемых программ в конкретных предметных областях реализуется с помощью включения в базу активных знаний модулей расширения, осуществляющих соответствующее преобразование графа будущей программы во внутреннем представлении. При работе с соответствующей базой активных знаний система просматривает модули расширения и пытается их применить при конструировании программ.

3. Описание и алгоритм генератора параллельных программ. В разделе излагается, как описанные выше идеи реализованы в конкретном программном средстве — генераторе параллельных программ. Генератор предназначен для преобразования VW-плана в исполняемый код. Интерфейс генератора принимает на вход описание VW-плана, представленного в виде JSON. Этот JSON содержит также и описание фрагментов кода операций. Фрагменты кода в генераторе представлены процедурами с сигнатурой определенного вида. Соответственно, сгенерированная программа будет содержать вызовы этих процедур. Эти вызовы процедур соответствуют операциям.

Для генерации программы необходимо выполнить следующие шаги: определение порядка выполнения операций, распределение операций по вычислительным узлам мультикомпьютера и отображение переменных на память.

Порядок выполнения операций определяется на основе зависимостей между переменными, описанных в VW-плане. Подробности изложены в листинге и далее в описании работы генератора. Распределение операций по узлам — это отдельная сложная научная задача, которая в работе не рассматривается. Обычно данную задачу решает планировщик, который может быть отдельным компонентом, предоставляющим распределение операций генератору параллельных программ. Поэтому распределение подается на вход генератору, или, если его нет, происходит автоматическое распределение, которое стремится равномерно распределить нагрузку между узлами, уменьшая время простоя процессоров.

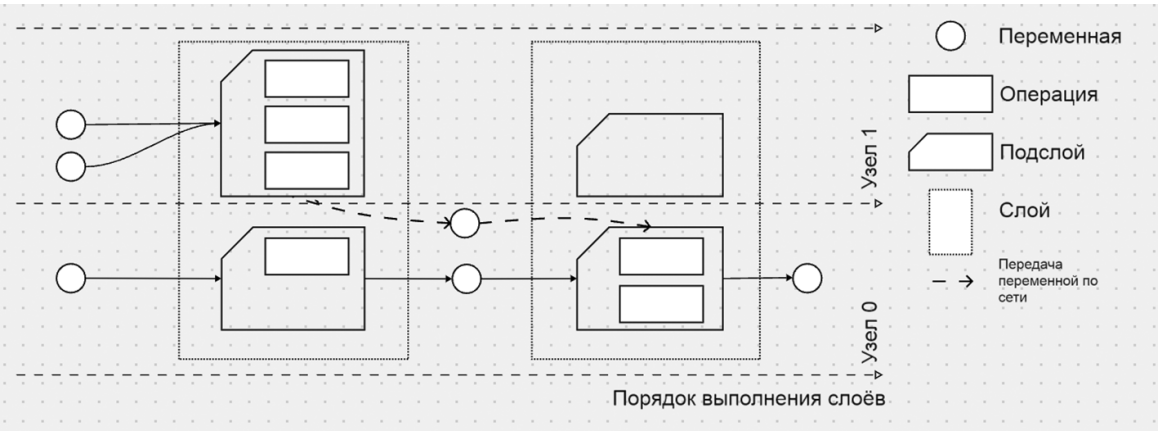


Рис. 1. Изображено послойное представление программы. По вертикали — доступные узлы (0, 1, 2 . . .), по горизонтали — порядок выполнения слоев, который выстраивается во время конструирования программы. Схематично изображены передачи данных между подслоями

Процесс генерации кода проходит в два этапа. Первый этап — преобразование VW-плана в послойное представление. Схематично данное представление можно наблюдать на рис. 1. Слой в данном случае — это множество независимых операций, которые могут быть выполнены параллельно. Подслой — множество операций, принадлежащих одному слою и назначенные на один вычислительный узел. То есть, слой представляет собой множество подслоев операций, которые не пересекаются.

Преобразование VW-плана в послойное представление выполняется итеративно. На каждой итерации происходит поиск всех операций, у которых все зависимости от переменных разрешены. То есть входные переменные операции — выходные переменные операций на предыдущих слоях. Для первой итерации, не имеющими зависимостей будут переменные, которые идут на вход генерируемой программе. Затем для каждой переменной в найденных операциях выполняется проверка на соответствие узла. Если узел, на котором находится переменная, не соответствует узлу, на котором должна выполняться операция, происходит вставка операции пересылки данных между подслоями. Для пересылки данных используется асинхронный механизм. Находится ближайший сформированный подслой, в котором переменная не имеет зависимости от операций, и добавляются операции асинхронной пересылки. В подслое, где будет выполняться операция, добавляется операция асинхронного приема переменной. Из найденных операций формируется подслой, соответствующий узлу, на котором выполняются операции. После помещения операций в подслой, все выходные данные помечаются как не имеющие зависимости.

Листинг 1. Создание слоев вычислений.

```
1: subroutine create_layers
2: input:
3:   ranks ⊆ ℤ     // Множество доступных рангов узлов
4:   X: set     // Множество переменных
5:   F: set     // Множество операций
6:   T: set     // Множество тегов (идентификаторов сообщений)
7:   rank: F → ranks     // Функция отображающая операцию на узел
```

```
  input_vars: F → 𝒫(X)      // Функция отображающая операцию на множество входных переменных операции
  output_vars: F → 𝒫(X)      // Функция отображающая операцию на множество выходных переменных операции
output:
  layers: seq(map (ranks ⇒ sub_layer))      // Последовательность слоев, разбитых по рангам
where:
  sub_layer is {      // структура данных, содержащая именованные поля
    operations ⊆ 𝒫(F),      // Операции подслоя
    message_sends ⊆ (ranks × X × T),      // Передачи сообщений для конкретного узла
    message_receives ⊆ (ranks × X × T)      // Приемы сообщений для конкретного узла
  }
local:
  computed := ∅
  var_ranks: map (X ⇒ 𝒫(ranks));      // Ранги, на которых доступна переменная на данной итерации
  F_local_iter: map (ranks ⇒ 𝒫(F));      // Локальные операции для каждого ранга
  R_local_iter: map (ranks ⇒ 𝒫((ranks × X × T)));      // Локальные сообщения приема для каждого ранга
  layers := []      // Инициализация последовательности слоев пустой последовательностью
initialize_local_variables(var_ranks, F_local_iter, R_local_iter, X, ranks)
      // Инициализирует var_ranks, F_local_iter и R_local_iter.
      // var_ranks указывает, на каких рангах доступна каждая переменная (изначально ни на каких).
    // Основной цикл построения слоев
while F ≠ ∅ do
      // Находим готовые операции: те, для которых все входные переменные вычислены
  ready_ops := {op ∈ F | ∀x ∈ input_vars(op), x ∈ computed}
      // Если не удалось найти готовые операции и еще есть необработанные операции, то решения не существует
  if ready_ops == ∅ and F ≠ ∅ then
    abort «Невозможно найти решение»
  end if
      process_ready_operations(ready_ops, var_ranks, input_vars, F_local_iter, R_local_iter, layers)
        // Определяет, какие переменные нужно передать между узлами, чтобы выполнить готовые операции.
        // Для каждой готовой операции:
        // - Определяет, какие входные переменные недоступны на текущем ранге.
        // - Планирует передачи сообщений (message_sends в layers) с других узлов
        // и приемы сообщений (message_receives в layers) на текущем узле.
        // - Добавляет операцию в F_local_iter для выполнения на текущем узле.
  new_layer := create_new_layer(F_local_iter, R_local_iter, ranks)
        // Создает новый слой на основе F_local_iter и R_local_iter.
```

```
44:       // Для каждого узла:
45:       // - Если на узле есть операции (F_local_iter[j] ≠ ∅), то создается подслой (sub_layer).
46:       // Добавляем новый слой к последовательности слоев
47:   layers := append(layers, new_layer)
48:     // Обновляем computed и var_ranks: отмечаем, какие переменные были вычислены на каких узлах
49:   for all op ∈ ready_ops do
50:    for each y ∈ output_vars(op) do
51:     computed := computed ∪ {y}      // Добавляем выходную переменную в множество вычисленных переменных
52:     j := rank(op)      // Определяем ранг, на котором была вычислена переменная
53:     var_ranks[y] := var_ranks[y] ∪ {j}      // Добавляем этот ранг в множество рангов, на которых доступна переменная
54:    end for each
55:   end for all
56:       // Удаляем обработанные операции из множества необработанных операций
57:   F := F \ ready_ops
58: end while
59:     // Возвращаем результат: последовательность слоев
60: return layers
```

Генератор параллельных программ написан на языке Python. Выходные данные — исполняемый код на языке C++. При этом параллелизм внутри узла поддерживается благодаря OpenMP [22], а распределенность — MPI. Для поддержки операций на GPU используются фрагменты кода, написанные с помощью CUDA. Гетерогенность необходима для эффективной реализации некоторых задач, пример которой будет далее в экспериментальных исследованиях.

Каждый подслой представлен в виде parallel sections (OpenMP). Примеры подслоев можно рассмотреть в листинге 3 и 4.

Листинг 2. Подслой сгенерированной программы с асинхронным приемом сообщения. Если входные данные для операции принимаются асинхронно, перед вызовом процедуры вставляются MPI_Wait и мьютексы для блокировки, чтобы дождаться получения данных. Если выходные данные операции отправляются асинхронно, то после вызова процедуры вставляется отправка выходных данных с помощью MPI_ISend на нужный узел. В данном листинге 1000 — это уникальный тег, который генерируется для каждой пересылки данных.

```
DF sub_result_union_u_1;

if (rank == 0) {
  #pragma omp parallel sections
  {
    omp_lock_t lock_1000;
    omp_init_lock(&lock_1000);
    MPI_Request request_1000;
    IRecv_df(sub_result_union_1_2, 1, request_1000, 1000);
    #pragma omp section
    {
```

```
12        omp_set_lock(&lock_1000);
13        MPI_Wait(&request_1000, MPI_STATUS_IGNORE);
14        omp_unset_lock(&lock_1000);
15        union_sub_result(sub_result_union_0_2, sub_result_union_1_2,
      sub_result_union_u_1);
16      }
17    }
18  }
```

Листинг 3. Показан пример структуры подслоя, реализованного с использованием OpenMP. Каждый pragma omp section представляет собой независимую операцию, которая может быть выполнена параллельно на одном узле. Внутри каждой секции вызывается соответствующая процедура (op, op1 и т. д.), реализующая операцию VW-плана.

```
1  if (rank == 0) {
2    #pragma omp parallel sections
3    {
4      #pragma omp section
5      {
6        op(A, B, C);
7      }
8      #pragma omp section
9      {
10       op1(A1, B1, C1);
11     }
12     ... // op2, op3, op4, op5 ...
13   }
14 }
```

Как следует из постановки задачи (раздел 2), ключевым аспектом предлагаемого подхода к снижению межмодульного трения является отказ от универсальной исполнительной системы в пользу специализированной статической генерации кода. В отличие от традиционных систем, использующих интерпретацию или динамическую компиляцию, статический генератор позволяет устранить характерные для них runtime-накладные расходы, особенно для задач, допускающих послойное представление вычислительного процесса.

Но, как отмечено в постановке задачи, подобная специализация неизбежно накладывает определенные ограничения. Основное из них связано с необходимостью принятия всех решений о распределении операций, планировании коммуникаций и синхронизации исключительно на этапе генерации кода. Это существенно снижает эффективность подхода при работе с задачами, требующими динамической адаптации вычислительного процесса в ходе выполнения. Для эффективной реализации программ с динамическими свойствами следует использовать исполнительные системы и интерпретаторы, где обеспечение динамических свойств более существенно, чем накладные расходы на вызов ФК.

4. Экспериментальные исследования. Для проведения тестирования была выбрана задача многоканальной свертки сейсмических сигналов. Подробное описание этой задачи можно найти в [12]. Упрощенно задача сводится к множественному применению быстрого преобразования Фурье (БПФ) для свертки входных сигналов с опорным сигналом.

В рамках решаемой задачи подготовлены тестовые данные в количестве 50 сейсмотрасс и опорного сигнала. В соответствии с техническими требованиями, время вычислений не

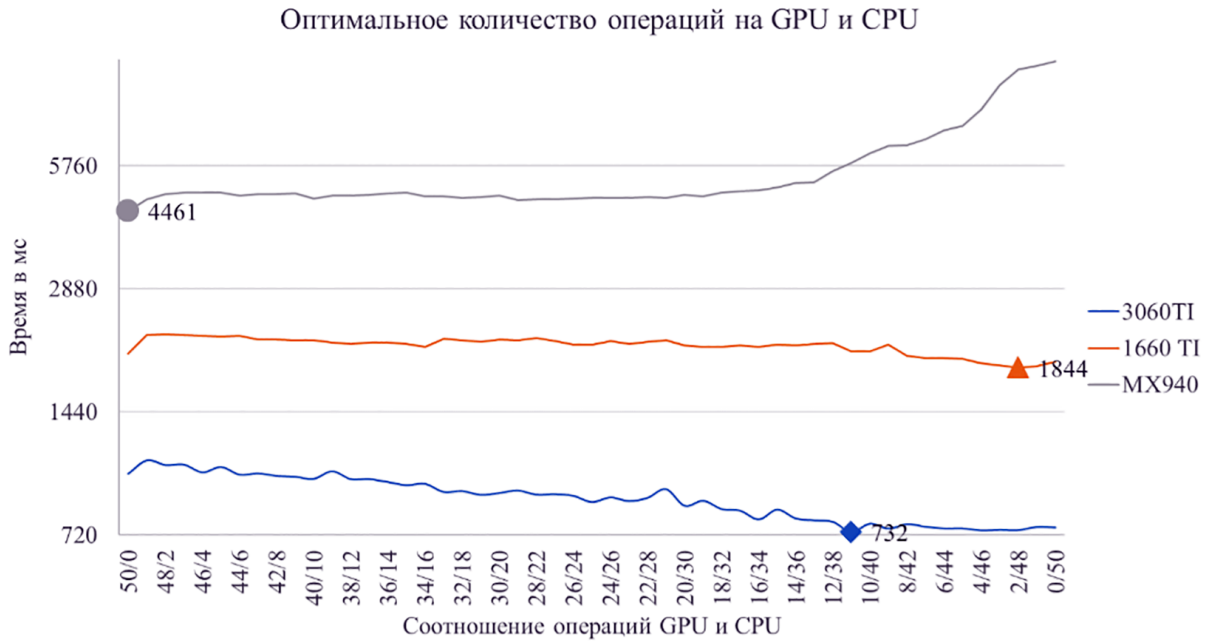


Рис. 2. Представлены результаты сравнительного анализа производительности при различных соотношениях CPU/GPU-вычислений для задачи многоканальной свертки сейсмических сигналов. По оси ординат отложено время выполнения (мс), по оси абсцисс — пропорция распределения операций между CPU и GPU. Исследование проводилось на трех типах графических ускорителей, демонстрирующих различные характеристики: тестирование на видеокарте MX940: минимальное время достигается при 50 операциях на CPU и 0 на GPU. После этого наблюдается плато и последующее увеличение времени, начиная с соотношения 20/30, что указывает на ограничения архитектуры MX940 в параллельной обработке большого количества БПФ на GPU; тестирование на видеокарте 1660 TI: виден резкий скачок времени выполнения после соотношения 50/0, предположительно вызванный накладными расходами на инициализацию CUDA. Далее наблюдается снижение времени, с минимальным значением при соотношении 2/48, что подтверждает эффективность гетерогенных вычислений для этой задачи на данной видеокарте; тестирование на видеокарте 3060 TI: На данной видеокарте удается добиться целевого показателя времени выполнения менее одной секунды. Минимальное время выполнения достигается при соотношении операций 10/40

должно превышать 1 секунды на небольшом компьютере, который можно «брать в поле» (например, ноутбуке с видеокартой).

Как отмечалось в постановке задачи (раздел 2), ключевым аспектом автоматического конструирования программ в концепции активных знаний является уменьшение накладных расходов модульных оболочек, включая оптимизацию передачи данных между CPU и GPU. В данной работе это достигается за счет двух механизмов:

— Использование специализированных фрагментов кода для GPU — в отличие от «оберточных» решений.

— Пакетная обработка операций БПФ через cuFFTPlan — как обсуждалось ранее, агрегация однотипных операций снижает накладные расходы на инициализацию CUDA и пересылку данных.

Для реализации данной задачи использовался генератор параллельных программ с операциями на CPU и GPU. Одна операция на CPU позволяет вычислить свертку одной сейсмической трассы. Фрагмент кода на GPU использует CUDA [23] для реализации быстрого преобразования Фурье. Быстрое преобразование Фурье (БПФ) реализовано с помощью библиотеки cuFFT, предоставляющей высокопроизводительные функции для обработки сигналов на GPU.

Одна из особенностей задачи заключается в том, что каждый сейсмический сигнал имеет одинаковую длительность, что позволяет вычислять несколько сейсмических сигналов в одном фрагменте кода на GPU. Для этого используется cuFFTPlan библиотеки cuFFT.

Оптимальное количество запускаемых операций на CPU и GPU определялось экспериментально. Эксперименты можно наблюдать на рисунке 2.

Результаты показывают, что:

— На слабых GPU (MX940) выгоднее использовать только CPU.

— На мощных GPU (3060 Ti) оптимально 10 CPU-операций + 40 GPU-операций (укладывается в 1 сек).

Это подтверждает тезис из постановки задачи: выбор типа фрагмента кода (CPU/GPU) и их агрегация существенно влияют на производительность.

Заключение. В работе рассмотрен подход к снижению доли накладных расходов на вызов модулей в программах, конструируемых автоматически на основе концепции активных знаний. Уменьшение накладных расходов достигается за счет формального описания свойств модулей в форме, доступной для автоматического использования системой конструирования, а также за счет обеспечения возможности автоматического применения оптимизирующих преобразований, специфичных для конкретной предметной области. Разработан генератор, обеспечивающий конструирование высокоэффективных программ частного вида на основе предлагаемого подхода. Его работа исследована на примере практической задачи многоканальной свертки сейсмических сигналов, где продемонстрировано высокое качество сконструированного кода.

Дальнейшее развитие подхода подразумевает его применение к решению других классов задач, что потребует развития математического аппарата описания модулей и их существенных функциональных и нефункциональных свойств, а также разработки алгоритмов, специфичных для различных предметных областей.

Список литературы

1. Kale L. V., Krishnan S. Charm++ a portable concurrent object oriented system based on C++ // Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. 1993. С. 91–108.

2. Charm++. Parallel Computer Network [Электронный ресурс]: `http://charmplusplus.org/` (дата обращения: 01.05.2025).

3. OpenCL [Электронный ресурс]: `https://www.khronos.org/opencl/` (дата обращения: 01.05.2025).

4. Coarray Fortran [Электронный ресурс] : [сайт]. URL: `http://caf.rice.edu` (дата обращения: 01.05.2025).

5. Reid J. Coarrays in the next fortran standard // ACM SIGPLAN Fortran Forum. New York, NY, USA ACM, 2010. Т. 29. № 2. С. 10–27.

6. DVM — система разработки параллельных программ [Электронный ресурс]: http://dvm-system.org/ru/about/ (дата обращения: 01.05.2025).

7. В. А. Бахтин [и др.]. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами // Вестник Южно-Уральского университета. Челябинск: Издательский центр ЮУрГУ, 2012. Серия: Математическое моделирование и программирование. № 18 (277). Выпуск 12. С. 82–92.

8. Kataev N., Kolganov A. The experience of using DVM and SAPFOR systems in semi automatic parallelization of an application for 3D modeling in geophysics // The Journal of Supercomputing. 2019. Т. 75. № 12. С. 7833–7843.

9. Малышкин В. Э., Перепелкин В. А. Построение баз активных знаний для автоматического конструирования решений прикладных задач на основе системы LuNA // Параллельные вычислительные технологии — XVIII всероссийская научная конференция с международным участием, ПаВТ'2024, г. Челябинск, 2–4 апреля 2024 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2024. С. 57–68.

10. Victor Malyshkin. Active Knowledge, LuNA and Literacy for Oncoming Centuries. In Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security. V. 9465. Springer-Verlag, Berlin, Heidelberg, 2015. P. 292–303.

11. Синтез параллельных программ и систем на вычислительных моделях / В. А. Вальковский, В. Э. Малышкин; Отв. ред. В. Е. Котов; АН СССР, Сиб. отд-ние, ВЦ. Новосибирск: Наука. Сиб. отд-ние , 1988. 126 с.

12. Выродов А. Ю. и др. Принципы организации программно-аналитической системы для параллельной обработки сейсмических данных // Вестник СибГУТИ. 2024. Т. 18. № 2. С. 57–68.

13. Ragan-Kelley J. et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines // Acm Sigplan Notices. 2013. Т. 48. № 6. С. 519–530.

14. PLUTO [Электронный ресурс]: https://pluto-compiler.sourceforge.net/ (дата обращения: 01.03.2025).

15. Bondhugula U. et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model // Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 17. Springer Berlin Heidelberg, 2008. С. 132–146.

16. Bondhugula U. et al. A practical automatic polyhedral parallelizer and locality optimizer // Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008. С. 101–113.

17. Polyhedral Compilation [Электронный ресурс]: http://polyhedral.info/ (дата обращения: 01.03.2025).

18. Malyshkin V. E., Perepelkin V. A. LuNA fragmented programming system, main functions and peculiarities of run-time subsystem // International Conference on Parallel Computing Technologies. Berlin, Heidelberg Springer Berlin Heidelberg, 2011. С. 53–61.

19. Малышкин В. Э., Перепелкин В. А. Определение понятия программы // «Проблемы информатики», 2024, № 2, С. 16–31.

20. CUDA Graphs [Электронный ресурс]: https://developer.nvidia.com/blog/cuda-graphs/ (дата обращения: 01.05.2025).

21. NVIDIA. cuFFT Library [Электронный ресурс]: https://docs.nvidia.com/cuda/cufft/index.html (дата обращения: 01.05.2025).

22. OpenMP [Электронный ресурс]: http://www.openmp.org/ (дата обращения: 01.03.2025).

23. NVIDIA CUDA [Электронный ресурс]: https://developer.nvidia.com/cuda-toolkit (дата обращения: 01.05.2025).

24. Malyshkin V. Active Knowledge, LuNA and Literacy for Oncoming Centuries // In Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security. V. 9465. Springer-Verlag, Berlin, Heidelberg, 2015. С. 292–303.

Виктор Эммануилович Малышкин — получил степень магистра математики в Томском государственном университете (1970), степень доктора технических наук в Новосибирском государственном университете (1993). В настоящее время является заведующим лабораторией синтеза параллельных программ в Институте вычислительной математики и математической геофизики СО РАН. Он также основал и в настоящее время возглавляет кафедру параллельных вычислений в Национальном исследовательском университете Новосибирска. Является одним из организаторов международной конференции PaCT (Parallel Computing Technologies), проводимых каждый нечетный год в России.

Имеет более 110 публикаций по параллельным и распределенным вычислениям, синтезу параллельных программ, суперкомпьютерному программному обеспечению и приложениям, параллельной реализации крупномасштабных численных моделей.

В настоящее время область его интересов включает параллельные вычислительные технологии, языки и системы параллельного программирования, методы и средства параллельной реализации крупномасштабных численных моделей, технологию активных знаний.

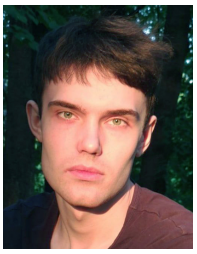
Victor Emmanuilovich Malyshkin graduated from Tomsk State University with the master of sciences degree in 1970, defended his candidate dissertation in physical and mathematical sciences in Computing Centre of SB RAS in 1984, and defended his doctoral dissertation in technical sciences in Novosibirsk State University (1993). Currently is head of parallel programs synthesis laboratory in the Institute of computational mathematics and mathematical geophysics SB RAS.

Is also a founder and head of the chair of parallel computing in Novosibirsk State University. Is one of organizers of the PaCT (Parallel Computing Technologies) international conference, being held each odd year in Russia. Has more than 110 publications on parallel and distributed computing, parallel programs synthesis, supercomputing software and applications, parallel implementation of large-scale numerical models. Current scientific interests include parallel computing technologies, languages and systems of parallel computing, methods and tools of software implementation of large-scale numerical models, the active knowledge technology.

Перепелкин Владислав Александрович — кандидат технических наук, старший научный сотрудник Института вычислительной математики и математической геофизики СО РАН; старший преподаватель каферды параллельных вычислений факультета информационных технологий Новосибирского государственного университета. Тел.: (383) 330-89-94, e-mail: `perepelkin@ssd.sscc.ru`. В 2008 году окончил Новосибирский государственный университет с присуждением степени магистра техники и технологии по направлению «Информатика и вычислительная техника». В 2023 году защитил кандидатскую диссертацию. Имеет более 20 опубликованных статей по теме автоматизации конструирования параллельных программ в области численного моделирования. Является одним из основных разработчиков экспериментальной системы автоматизации конструирования численных параллельных программ для мультикомпьютеров LuNA (от Language for Numerical Algorithms). Область профессиональных интересов включает автоматизацию конструирования параллельных программ, языки и системы параллельного программирования, высокопроизводительные вычисления.

Perepelkin Vladislav Aleksandrovich — PhD in Computer Science. Graduated from Novosibirsk State University in 2008 with the master degree in engineering and technology in

computer science. Defended his PhD thesis in 2023. Nowadays has the senior researcher position at the Institute of Computational Mathematics and Mathematical Geophysics (Siberian Branch of Russian Academy of Sciences), and also has a part time senior professor position in the Novosibirsk State University. He is an author of more than 20 papers on automation of numerical parallel programs construction. He is one of main developers of system LuNA (Language for Numerical Algorithms) for automatic construction of numerical parallel programs. Professional interests include automation of parallel programs construction, languages and systems of parallel programming, high performance computing.



Нуштаев Юрий Юрьевич — студент 2-го курса магистратуры по направлению «Информатика и вычислительная техника» Новосибирского государственного университета. Область интересов: параллельное программмирование, высокопроизводительные вычисления. Почта: `y.nushtaev@g.nsu.ru`

Дата поступления — 02.06.2025