DOI: 10.15514/ISPRAS-2022-34(4)-7

Data distribution and parallel code generation for heterogeneous computational clusters

N.A. Kataev, ORCID: 0000-0002-7603-4026 <kataev_nik@mail.ru>
A.S. Kolganov, ORCID: 0000-0002-1384-7484 <alexander.k.s@mail.ru>
Keldysh Institute of Applied Mathematics of Russian Academy of Sciences,
4, Miusskaya sq., Moscow, 125047, Russia

Abstract. We present new techniques for compilation of sequential programs for almost affine accesses in loop nests for distributed-memory parallel architectures. Our approach is implemented as a source-to-source automatic parallelizing compiler that expresses parallelism with the DVMH directive-based programming model. Compared to all previous approaches ours addresses all three main sub-problems of the problem of distributed memory parallelization: data and computation distribution and communication optimization. Parallelization of sequential programs with structured grid computations is considered. In this paper, we use the NAS Parallel Benchmarks to evaluate the performance of generated programs and provide experimental results on up to 9 nodes of a computational cluster with two 8-core processors in a node.

Keywords: compilers; automatic parallelization; code generation; distributed memory systems

For citation: Kataev N.A., Kolganov A.S. Data distribution and parallel code generation for heterogenious computational clusters. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 89-100. DOI: 10.15514/ISPRAS-2022-34(4)-7

Построение распределения данных и генерация кода при распараллеливании на гетерогенный вычислительный кластер

Н.А. Катаев, ORCID: 0000-0002-7603-4026 <kataev_nik@mail.ru>
А.С. Колганов, ORCID: 0000-0002-1384-7484 <alexander.k.s@mail.ru>
ИПМ им. М.В. Келдыша РАН,
125047, Россия, Москва, Миусская пл., д.4

Аннотация. В данной статей рассматривается новый подход к компиляции последовательных программ для их последующего выполнения на вычислительных системах с распределенной памятью. Предложенный подход был реализован в виде автоматически распараллеливающего компилятора для программ на языках Си и Фортран. Для описания параллелизма, обнаруженного в программе, используется директивная модель параллельного программирования DVMH. Таким образом, реализованный компилятор выполняет преобразование программ на уровне исходного кода, добавляя в них высокоуровневые спецификации параллелизма в терминах DVMH модели. Распараллеливание основано на анализе гнезд циклов программы, содержащих обращения к многомерным массивам, для которых большинство индексных выражений линейно зависит от индуктивных переменных циклов гнезда. Основной областью применения предложенного подхода являются программы научно-технических расчетов, реализующие вычисления на структурированных сетках. В отличие от подходов к распараллеливанию программ, предложенных в других работах, наш подход охватывает решение всех трех основных задач, возникающих при распараллеливании для систем с распределенной памятью: распределение данных, распределение вычислений и оптимизация коммуникационных обменов между узлами вычислительной системы. Для оценки эффективности получаемых параллельных программ, мы использовали некоторых приложения из набора NAS Parallel Benchmarks. В статье приведены

результаты экспериментов, в которых были задействованы до 9 узлов вычислительного кластера, каждый из которых содержал два 8-ядерных процессора.

Ключевые слова: компилятор; автоматическое распараллеливание; генерация кода; системы с распределенной памятью.

Для цитирования: Катаев Н.А., Колганов А.С. Построение распределения данных и генерация кода при распараллеливании на гетерогенный вычислительный кластер. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 89-100. 10.15514/ISPRAS-2022-34(4)-7

1. Introduction

Most of current high performance computing (HPC) systems tend to be heterogeneous and may comprise diverse compute devices. However, from the prospective of obtaining greater processing power on the way to exascale computing, the common feature of these systems is the distributed memory allocation. Such systems consist of multiple compute nodes which are connected with a high performance interconnect and each node operates with its own data. The only way to distribute computations across different nodes is sending and receiving messages over the interconnect.

Distributed memory drastically complicates parallel programming. The programmer has to take into account not only distribution of computation but also distribution of data and cost of data movement. To minimize communication overhead the programmer has to ensure data locality. Another goal is even data distribution in order to balance computations.

Unlike incremental parallelization applicable for shared memory, distributed memory requires global decision making since individual parts of a program may impose conflicting requirements. These conflicts will ultimately lead to additional communications aimed at data redistribution.

One of the parallel programming models, widely used to develop compute-intensive applications on distributed-memory clusters, is the Message Passing Interface (MPI). However, its low-level forces the programmer to manage distribution of data and computation manually, as well as communications. This means that parallelization for distributed memory even of a simple program can be very error-prone and time-consuming. Therefore, automation of parallel programming for distributed-memory systems becomes very much desirable. Especially while the complexity and size of systems is growing from year to year.

In this paper, we propose a technique for automatic translation of sequential programs of scientific-technical calculations to parallel ones suitable for execution on heterogeneous computational clusters. Our technique aims at parallel execution of structured grid computations and allows processing sequences of arbitrary nested loops with almost affine accesses.

The problem of distributed memory parallelization requires a solution to three main sub-problems. An automation tool has to distribute data as well as computation and has to manage communications: typically accesses to remote data and data redistribution. The proposed technique overcomes all these problems. However, in the paper we pay the most attention to data distribution because yet we do not find any common solution to this sub-problem, which is suitable for large compute applications, in the current literature.

We implemented our technique as an automatic parallelizing compiler which generates a parallel version of a sequential C or Fortran program with parallelism specifications expressed with DVMH [1][2] directive-base programming model. To extract properties of an original program which are necessary for its parallelization the compiler relies on static and dynamic analysis techniques. We also follow an implicit parallel programming methodology [3][4] which implies that the sequential program must be well-formed. Thus, a preliminary sequential program transformation may be helpful. The user may also assert some high-level program properties which are essential for parallelization. Nevertheless, automatic parallelization ensures that the programmer does not write parallel code directly.

This paper makes the following contributions:

- a novel approach to automatic program parallelization for distributed memory systems which covers distribution of data and computations as well as access to remote data and data redistribution;
- a parallelizing compiler for code generation which produces parallel versions of C and Fortran programs suitable for execution on heterogeneous computational clusters;
- experimental evaluation of the presented approach on some programs from the NAS Parallel Benchmarks [5].

The rest of the paper is organized as follows. Section 2 presents our program execution model and proposes a solution to the problem of distributed memory parallelization focusing on the data distribution sub-problem. Section 3 summarizes implementation details and briefly describes frameworks that we used to analyze sequential programs and to implement code generation. Section 4 provides experimental results. Section 5 discusses the related work and, finally, section 6 concludes this paper.

2. Distributed memory parallelization

2.1 Execution model

In this section, we highlight the main details of our abstract execution model representing a distributed memory system.

Our parallelization technique aims at processing sequential programs with structured grid computations. Hence we assume that a distributed memory system is formed by a multidimensional grid of a virtual nodes and each node executes operations on a part of the computational grid. The owner-computes rule [6] is applied to determine a node to execute each assignment statement. Thus, each node has its own (i.e. local) data which are allocated in its own memory space, which is not visible to other nodes. It also may access remote data which are allocated on other nodes by sending and receiving messages over the interconnect.

We also assume that the main source of parallelism is nested loops and that entire iteration of a loop can be executed by a single node only. These loops produce computations on multidimensional arrays which are the main source of data to be partitioned between virtual compute nodes. A data distribution rule divides array dimensions into almost equal blocks and implies that each node has its own sub-array with the same number of dimensions but of smaller sizes.

If the number of dimensions of a grid of nodes is less than the number of array dimensions, some array dimensions are not partitioned. Otherwise, if the number of grid dimensions is greater than the number of array dimensions, some array dimensions are replicated between nodes.

If we consider two arrays accessed in the same loop there is some kind of intuitive relation between elements of different arrays. To reduce the communication overhead, we have to distribute elements accessed at the same loop iteration to the same node. Thus, an alignment of an array A with a distributed array B is an accordance between an element of the array A and an element or a sub-array of the array B. This accordance aims to reduce the cost of data movement. To specify alignment we use affine expressions in a form $a * i + b$. Therefore, a distribution rule for the array B defines the distribution rule for the array A, i.e. if an element i of B is allocated on a compute node, the corresponding element $a * i + b$ of A is allocated on the same node.

If there is no distribution or alignment rule for a variable, we replicate it between all compute nodes. The replicated variable must have the same value in each node except reduction and private variables.

Different array accesses may produce different alignments, however, we choose only one of them to generate parallel version of a program. Other violated alignments lead to data movement. Moreover, non-affine alignments cannot be established and always lead to data movement. Our execution model supports two kinds of communications. Firstly, violated affine alignments allow us

to implement shadow edges [7]. For structured grid applications, it is useful to extend each sub-array with a shadow area to overlap with its neighbors. Shadow edges have to be updated only if their values have been altered on the owner node. Secondly, all other communications are implemented as an immediate access to remote data when corresponding data are required.

2.2 Data Distribution

In this section, we present our technique aimed to solve the data distribution sub-problem. This technique relies on a graph-based representation of an array alignment. We introduce a graph of arrays which depicts arrays accessed in loop nests. A vertex of the graph is a dimension of an array and an edge connects dimensions that should be aligned. Note that the alignment relation is transitive. Thus, two dimensions are aligned if there is a path between corresponding vertexes in a graph.

A pair of array accesses in a loop produces edges in a graph. Coefficients in affine subscript expressions which calculate offset from the beginning of array dimensions are attached to the edge and allow us to infer alignment expression. Only affine subscript expressions, which depend on the same single counter of a loop, produce edges. Note, that values of distinct induction variables of the same loop can be computed through the loop counter. Therefore, corresponding subscript expressions also produce edges in a graph.

If a pair of array accesses produces an edge in a graph, a kind and a weight of alignment are also attached to the edge. Three kinds of alignments are possible:

- WW if both accesses write into memory,
- WR if one of accesses writes into memory and another access reads from memory,
- RR if both accesses read from memory.

A weight of alignment equals to $Loop_w * Transfer_w$. $Transfe_w$ estimates a number of bytes to be send or received if the alignment is violated. Computation of $Transfer_w$ is based on sizes of arrays which are known from static and dynamic analysis. $Loop_w$ estimates a corresponding loop weight and shows how often the loop is executed. Static and dynamic analysis techniques allow us to compute loop weights. If an edge with the same subscript coefficients and the same kind already exists in a graph its weight is updated.

It is necessary to know a kind of alignment to compare different edges according to memory access types. A priority of WW edge is higher than a priority of WR edge which, in its turn, is higher than a priority of RR. The owner-computes rule establishes this priority because it is forbidden to write to non-local memory.

In the first step, loops in a sequential program are analyzed to determine all possible alignments which depend on array accesses at the same loop iteration. Procedure $buildGraph$, shown in Listing 1, builds a corresponding graph of arrays. It calls the following procedures when necessary:

- $dimension(Expr)$ to determine a dimension of an array which this subscript expression corresponds to,
- $kind(Acc_1, Acc_2)$ to determine a kind of alignment which these array accesses produce,
- $array(Acc)$ to determine a top level declaration of an accessed array, i.e. it analyzes a call graph and establishes correspondence between formal and actual parameters which have an array type.

This procedure produces a graph of arrays which may be ambiguous or may have conflicting edges. Two cases are possible:

- there is a cycle in a graph that implies two different affine expressions to specify an alignment of the same array dimension,
- two different dimensions of an array are aligned on each other, i.e. two different dimensions of

the same array should be mapped to the same node..

```
procedure buildGraph
  for $L \in Loops\ of\ a\ program$
   for $Acc_1 \in Array\ accesses\ in\ L$
    for $Expr_1 \in Subscripts\ in\ Acc_1$
     if $Expr_1\ is\ a_1 * i + b_1$ and $i\ is\ loop\ counter\ of\ L$
      for $Acc_2 \in Array\ accesses\ in\ L$
       for $Expr_2 \in Subscripts\ in\ Acc_2$
        if $(Acc_1 \neq Acc_2\ or\ dimension(Expr_1) \neq dimension(Expr_2))$ and
          $Expr_2\ is\ a_2 * i + b_2$ and $i\ is\ loop\ counter\ of\ L$
         Add vertexes $V_{array(Acc_1)}^{dimension(Expr_1)}, V_{array(Acc_2)}^{dimension(Expr_2)}$, if they do not exist.
         Add an edge $E_{a_1,b_1,a_2,b_2}^{kind(Acc_1,Acc_2)}(V_{array(Acc_1).}^{dimension(Expr_1)}, V_{array(Acc_2).}^{dimension(Expr_2)})$, if it does not exist.
        Update weight of the edge.
```

Listing 1. Algorithm to build a graph of arrays

In the second step, an original graph of arrays is reduced to disambiguate these conflicts. The goal is to minimize total weight of edges to be removed from the graph. Fig. 1 shows a graph of arrays for a fragment of a source code in Listing 2.

```
integer A(40,50), B(40,50)
do i = 1, 30
  z = A(i, 1)
  do k = 5, 30
    a(i, k) = b(i, k) + b(k, k) / z
  enddo
enddo
do i = 1, 30
  z = a(i, 2)
  do k = 5, 30
    a(i, k) = b(i, k + 1) + z
  enddo
enddo
```

Listing 2. Example of a Fortran program which is used to build a graph of arrays in Fig. 1

Different weight models can be used to calculate w_1, w_2, w_3, w_4 weights, so we do not specify exact weights in the example. Assuming $w_1 > w_2$, $w_3 > w_2$ and $w_3 > w_4$, the solid lines determine a possible graph after all conflicts are disambiguated.

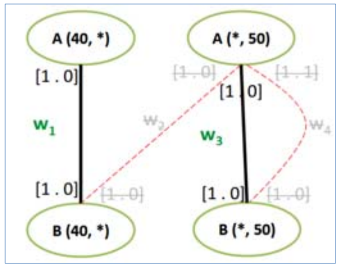

Fig. 1. Graph of arrays for the program in Listing 1

We adapted Prim's algorithm for finding a minimum spanning tree (MST). A minimum spanning tree is a subset of the edges of a connected edge-weighted undirected graph that connects all the vertices together. It has the minimum possible total edge weight and does not have cycles. If the

source graph is connected, then a spanning tree is built, but if there are several disconnected components in the source graph, then the result is a spanning forest. Our goal is to find the set of edges with the highest weight. Thus, we can use an algorithm similar to Prim's algorithm to find a maximum spanning tree, i.e. a tree which has the maximum possible total edge weight.

To resolve conflicts of a second type, we add temporary edges between all different dimensions of the same array. A weight of these edges must be greater than the sum of all weights in the graph. Hence, these temporary edges are never removed and a spanning tree does not contain edges that imply alignment of two dimensions of the same array on each other.

This algorithm has linear complexity depending on the number of vertices in the graph. However, the found solution is not always optimal, but the search of optimal solution has exponential complexity and it is not applicable in practice if the total number of dimensions of arrays in a program is significantly greater than 10. On the other hand, the algorithm for finding the maximum spanning tree has not only linear complexity but can be also performed in parallel, so it is applicable to any program with any number of arrays.

3. Code generation

We implemented our technique as automatic parallelizing Fortran and C compiler. We adapted the compiler from the System FOR Automated Parallelization (SAPFOR) [8], which is a software development suit that is focused on cost reduction of manual program parallelization, to implement our parallelization technique. To express parallelism specifications in a code the DVMH [1][2] directive-based programming model is used.

2.3 The DVMH programming model

DVMH was designed to create parallel programs of scientific technical calculations for heterogeneous computational clusters. This model provides the programmer with high-level specifications of parallelism. Thus, it introduces directives to specify data and computation distribution, to manage accesses to remote data and to highlight the computational regions which can be executed on accelerators and multiprocessors.

We choose the DVMH model because it satisfies constraints which our execution model exposes. The DVMH languages allow us to generate distribution and alignment rules in a simple way, as well as to implement shadow edges and other accesses to remote data. Moreover, the DVMH compilers and run-time system implement various optimizations: data transformation at run-time to choose the right memory access pattern, dynamic CUDA handler compilation during the program run-time, parallel execution of loops with regular loop carried dependencies on GPU and others. Therefore, the presence of these optimizations significantly simplifies the implementation of the automatic compilers.

Listing 3 shows the distribution (the *distribute* specification) and alignment (the *align* specification) rules for some arrays in the BT application from the NAS Parallel Benchmarks. The *shadow* specification determines sizes of shadow edges for the array *u*.

```
#pragma dvm array distribute [block][block][block]
double us[KMAX/2*2+1][JMAX/2*2+1][IMAX/2*2+1];

#pragma dvm array align ([k][j][i][] with us[k][j][i]),\
                  shadow[2:2][2:2][2:2][2:2]
double u[(KMAX+1)/2*2+1][(JMAX+1)/2*2+1][(IMAX+1)/2*2+1][5];
```

Listing 3. Example of data distribution in the CDVMH language.

Listing 4 shows the computation decomposition (the *parallel* specification) for a loop nest in the BT application. The computation decomposition depends on the data distribution. Hence the definition of a parallel loop must include the *on* specification which associates loops in the perfect loop nest with dimensions of a distributed array.

The mentioned specifications imply a relation between loop iterations and array elements that enables compile-time and runtime optimizations. For example, knowing how arrays are associated with each other and knowing the loop mapping rules, the DVMH runtime system determines the optimal representation of arrays in the device memory for a given parallel loop, and subsequently, it performs the dynamic transformation of arrays before the loop execution. As a result, on GPUs all accesses to global memory, performed by CUDA threads of each warp, will be combined. Adjacent threads of the CUDA-block will access neighboring cells of the GPU's global memory and the loop can be performed up to 10 times faster [9].

The hyperplane method (the *across* specification) is implemented in the DVMH model to execute loops with regular loop carried dependencies (some kind of flow dependencies with distances limited by a constant) in parallel. The per-diagonal transformation is implemented in the DVMH runtime system to efficiently execute these kinds of loops on GPUs [9].

To update shadow edges the *shadow_renew* specification is used. To specify all other communications the DVMH model provides the *remote_access* specification.

```
#pragma dvm region
{
#pragma dvm parallel ([k][j][i] on u[k][j][i][])\
            reduction(sum(r1),sum(r2),sum(r3),sum(r4),sum(r5)),\
            private(u_exact,xi,eta,zeta,m,add)
  for(k = 0; k <= PROBLEM_SIZE - 1; ++k)
    for(j = 0; j <= PROBLEM_SIZE - 1; ++j)
      for(i = 0; i <= PROBLEM_SIZE - 1; ++i) {
      ...
        for (m = 1; m <=5; ++m)
          u_exact[m-1] = ...
               add = u[k][j][i][0] - u_exact[0];
               r1 = r1 + add * add;
               ...
      }
}
```

Listing 4. Example of computation distribution in the CDVMH language

The *region* specification forms the computational region, which can be executed on different computational devices (multi-core CPU and accelerators). The region is a fragment of a source code with one entrance and one exit. Each region may enclose one or more parallel loops. The iterations of parallel loops inside the region are partitioned between the devices selected for the region execution according to the *parallel* specification.

Code fragments outside the regions are always executed on CPU. The DVMH model introduces actualization directives (the *actual* and *get_actual* specifications) to manage data transfer between random access memory of CPU and memories of accelerators. These specifications must be placed outside computational regions. The deferred semantic of actualization directives allows DVMH runtime system to avoid redundant data transfer. Moreover, there is no difference whether all regions are executed on GPU or some part of them is targeted to the CPU-only execution. Actualization directives affect only transfer between regions and code fragments outside them. The DVMH runtime system will manage the necessary data transfer in an automatic way.

2.4 SAPFOR

SAPFOR includes an automatic parallelizing compiler that relies on static [10] and dynamic [11] analysis techniques. However, unlike conventional automatic parallelizing compilers, which may

suffer from a lack of user participation, SAPFOR relies on an implicitly parallel programming model and involves the user in the parallelization process [12].

SAPFOR has already implemented semi-automatic parallelization approach to exploit loop-level parallelism for multi-core processors and accelerators. It uses the DVMH model to express parallelism specifications. SAPFOR inserts three kinds of annotations: specification of parallel loops, specification of compute regions and specification of data transfer between a memory of CPU and memory of accelerator. It searches for the outermost perfect loop nests to execute them in parallel and then it examines whether some properties of the loop nest prevent its parallel execution (safety of control flow and memory accesses, canonical loop form, etc.). To insert actualization directives SAPFOR examines the direction of data usage in every loop. It also applies alias analysis to determine indirect memory accesses in C programs. SAPFOR relies on the optimization of data transfer at run-time that DVMH run-time library makes. So, it marks each assignment to the variable outside a compute region with the *actual* specification and places the *get_actual* specification before each statement which uses the variable changed in any DVMH region.

In our work we extend SAPFOR capabilities to produce parallel versions of C and Fortran programs suitable for execution on distributed memory systems. Our implementation allows the compiler to insert:

- *distritbute* and *align* specifications to partition array elements between compute nodes,
- *parallel* and *on* specifications to partition computations according data distribution,
- *shadow, shadow_renew* and *remote_access* specifications to express necessary data transfer between compute nodes.

We implemented our parallelization technique as a separate library. This library provides interface to build the graph of arrays and to get data and computation distribution in the language independent way. It automatically resolves possible conflicts in the graph and rejects arrays that cannot be partitioned across multiple computational nodes.

The core of the SAPFOR compiler architecture is the SAPFOR pass framework. Passes perform analysis and transformation of the program. New passes can be constructed to implement new capabilities. We added new SAPFOR passes to collect and fill data necessary to build the graph of arrays. To place necessary DVMH specifications in a source code we also extended the code generation passes for both C and Fortran programming languages. We use passes available in SAPFOR to determine loop-carried and spurious data dependencies, to find parallelizable loops and to find computational regions suitable for execution on accelerators and multiprocessors.

4. Experimental results

This section presents performance results from 3 applications from the NAS Parallel Benchmarks [5]: BT (Block Tri-diagonal solver), CG (Conjugate Gradient), EP (Embarrassingly Parallel) that we parallelized using out technique. Fig. 2 and fig. 3 show the execution time of the generated DVMH programs in comparison with the origin MPI programs written by the developers of the NAS Parallel Benchmarks.

We conducted experiments on the K10 [11] cluster of NUMA nodes. Each node contains two 8-core Intel Xeon E5-2660 processors and three GPUs NVIDIA Tesla M2090. All codes were compiled with Intel C/C++ and Fortran compilers version 14.0.1 with option -O2. DVMH compiler was preliminary used to translate programs with DVMH specifications to MPI+OpenMP+CUDA code. For all experiments, we use the total power of one, four and nine nodes. We did not use GPUs to achieve the fair comparison of our solution to distributed-memory parallelization problem with a manual parallelization approach using MPI.

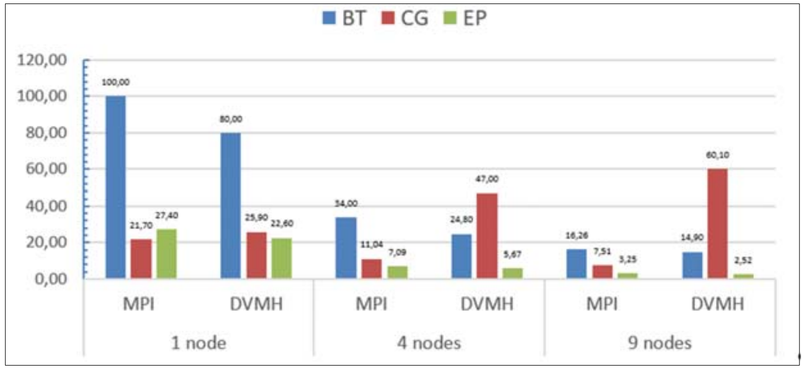

Fig. 2. Times in seconds of Fortran programs, NPB 3.3 class C

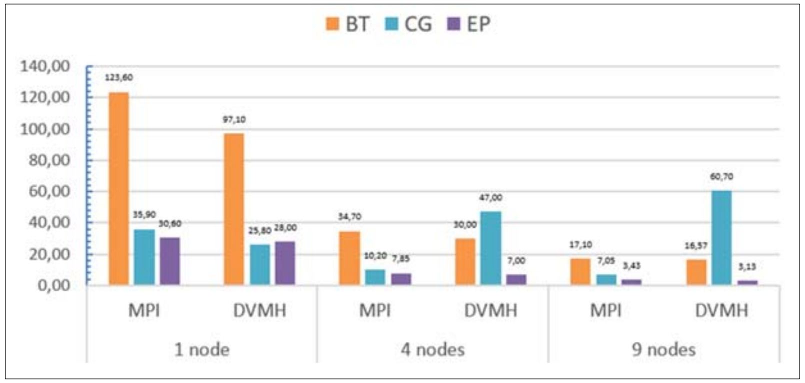

Fig. 3. Times in seconds of C programs, NPB 3.3 class C

Unfortunately, in the suite, there are no built-in C versions of considered applications, so we translated Fortran to C manually. To obtain well-formed versions of original programs, which can be converted to DVMH programs fully automatically, we applied some source-to-source transformations implemented in SAPFOR (functions inlining, loop fusion, loop distribution, and etc.). Also, we manually did some transformations which have not been implemented in SAPFOR yet.

Table 1 and Table 2 show speedup of parallel DVMH programs relative to their well-formed sequential versions. The first group of columns represents execution time of sequential programs before and after source-to-source transformations.

Number of transformations implemented in SAPFOR must be extended to obtain well-formed versions of other applications from the suite. We plan to review these transformations in future works and figure out basic source-to-source transformations that can be done automatically. Their implementation as separate passes in the SAPFOR pass framework will allow us to parallelize other applications.

Table 1. Speedup of Fortran-DVMH programs relative to well-formed sequential versions

	Serial, time (s.)		1 node (16 MPI)		4 node (64 MPI)		9 node (144 MPI)	
	Origin	Well-formed	Time (s.)	Speedup	Time (s.)	Speedup	Time (s.)	Speedup
BT	925,77	934,14	80,00	11,7	24,80	37,7	14,90	62,7
CG	282,99	297,67	25,90	11,5	47,00	6,3	60,10	4,95
EP	341,53	381,73	22,60	16,9	5,67	67,3	2,52	151,5

Table 2. Speedup of CDVMH programs relative to well-formed sequential versions

	Serial, time (s.)		1 node (16 MPI)		4 node (64 MPI)		9 node (144 MPI)	
	Origin	Well-formed	Time (s.)	Speedup	Time (s.)	Speedup	Time (s.)	Speedup
BT	955,04	816,38	97,10	8,4	30,00	27,2	16,57	49,3
CG	314,43	295,74	25,80	11,5	47,00	6,3	60,70	4,9
EP	431,95	408,89	28,00	14,6	7,00	58,4	3,13	130,6

The experiments show that automatically generated versions of BT and EP applications have the similar performance to the manually parallelized ones.

However, the MPI version of CG program running on more than one node significantly outperforms the automatically generated DVMH version. Detailed examination of the benchmark shows that most of time is spent in the multiplication of a sparse matrix by a vector. This operation leads to indirect array accesses which cannot be efficiently processed by our parallelization technique that aims at processing sequential programs with structured grid computations. The current implementation produces at each iteration a collective operation to broadcast remote data between all processes. The cost of this data movement degrades parallel program performance if data are transferred through the interconnect between different nodes. If GPUs are used, then 16 processes (1 node) will be enough to achieve high performance.

The available extension of the DVMH model for irregular grids may be helpful to increase parallelization performance, so we plan to address this problem in future works and to improve our parallelization technique.

5. Related works

Various approaches exist to simplify parallelization for distributed-memory systems. However, most of approaches do not overcome all three main sub-problems.

For example, [6] proposes an approach to derive the computation decomposition from predetermined data distribution. The polyhedral approach is used to develop a mathematical model for code generation and to optimize the communications. These optimizations include eliminating redundant messages, aggregating messages, and hiding the communication latency by overlapping the communication with computation [6].

The predefined data distribution was also used in the [14] tool. This tool provided the user with an interactive subsystem to manage the parallelization process, to define data distribution and to choose program transformations. In both works the owner-computes rule is applied to derive the computation decomposition. However, [6] makes it possible to relax owner-computes rule. Thus, locations written to can be replicated or mapped to different nodes.

The Molly [15] tool extends the capabilities of the Polly [16] compiler for shared memory systems, built on top of LLVM [17]. Molly also relies on the polyhedral model. It introduces a special data type to define distributed arrays which allows the compiler to control the absence of pointer arithmetic operations applied to distributed data. The tool uses a fixed block-distribution while communications are generated in an automatic way. The sizes of blocks are equal for every node. There is no way for the programmer to specify alignment of data on each other. To derive computation distribution the own-computes rule is applied. It is assumed that in the future the user will be able to define an arbitrary mapping of data and computation by putting appropriate directives in a source code. Molly also imposes additional restrictions on the well-structured code fragments (SCoPs), does not deal with reduction operations and supports distribution of global data only. Hence an accurate interprocedural analysis is not required.

An automatic parallelizing compiler presented in [18] also uses the polyhedral compiler framework to translate sequential C programs to parallel ones which are suitable for execution on distributed-memory systems. This work extends the capabilities of the Pluto [19] polyhedral compiler for shared memory systems. The data distribution is not required, and the computation partitioning and data dependencies determine the owner of data at a particular moment. Therefore, this approach does not support the global decision-making in data distribution and finally may lead to an increase in the frequency and the volume of communications.

The problem of data distribution was addressed in the Paradigm [20] tool. The research aims at parallelizing sequential programs in the Fortran 77 language. It also addresses other parallelization problems along with data distribution: communication optimization, support for irregular computations, multi-threaded execution and pipeline execution of loops with cross-iteration dependencies. However, only simple inter-dimensional alignment is performed and nor offset, nor stride within a given pair of dimensions cannot be used. The practical applicability of the tool is not clear because the experimental results are only given for small computing kernels.

6. Conclusion

In this paper, we introduce the technique for automatic translation of sequential programs of scientific-technical calculations to parallel ones suitable for execution on heterogeneous computational clusters. We emphasize in the paper that a problem of distributed memory parallelization requires a solution to three main sub-problems. These sub-problems include data and computation distribution as well as communication optimization and our technique addresses all of them. In the paper, we focus more on data distribution sub-problem because we do not find yet any common solution to this sub-problem, which is suitable for a large compute applications, in the current literature.

We present a novel data structure, called the graph of arrays, and use it to derive data decomposition from affine array accesses in loop nests. We also determine an alignment of arrays with each other to reduce the frequency and the volume of data movement. To choose between possible alignments, we estimate communication costs if one of them is violated.

We implemented our technique as an automatic parallelizing compiler which generates a parallel version of a sequential C or Fortran program with parallelism specifications expressed with DVMH directive-base programming model. Conducted experiments show that in conjunction with implicit parallel programming methodology the generated code is capable of matching hand-coded MPI versions of programs with structured grid computations.

References / Список литературы

[1]. Коновалов Н.А., Крюков В.А. и др. Fortran DVM - язык для разработки мобильных параллельных программ. Программирование, том: 21, вып. 1, 1995 г., стр. 49-54 / Konovalov N.A., Krukov V.A. et al. Fortan DVM: a Language for Portable Parallel Program Development. Programming and Computer Software, vol. 21, no. 1, 1995, pp. 35-38.

[2]. Bakhtin V.A., Klinov M.S. et al. Extension of the DVM-model of parallel programming for clusters with heterogeneous nodes. Bulletin of South Ural State University, series: Mathematical Modeling, Programming & Computer Software, no. 18 (277), issue 12, 2012, pp. 82-92 (in Russian) / Бахтин В.А., Клинов М.С. и др. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами. Вестник ЮУрГУ. Серия: Математическое моделирование и программирование, № 18 (277), вып. 12, 2012 г., стр. 82-92.

[3]. Hwu W.-m., Ryoo S. et al. Implicitly parallel programming models for thousand-core microprocessor, In Proc. of the 44th annual Design Automation Conference (DAC '07), 2007, pp. 754-759.

[4]. Vandierendonck H., Rul S., De Bosschere K. The Paralax infrastructure: automatic parallelization with a helping hand. In Proc. of 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 389-400.

[5]. NAS Parallel Benchmarks. Available at https://www.nas.nasa.gov/publications/npb.html, accessed 16.09.2022.

[6]. Amarasingh S.P., Lam M.S. Communication Optimization and Code Generation for Distributed Memory Machines. In Proc. of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, 1993, pp. 126-138.

[7]. Thomas B. HPF Library, Language and Compiler Support for Shadow Edges in Data Parallel Irregular Computations. In Proc. of the 8th International Workshop on Compilers for Parallel Computers, 2000, pp. 21-34.

[8]. Klinov M.S., Krukov V.A. Automatic parallelization of Fortran programs. Mapping to cluster. Vestnik of Lobachevsky University of Nizhni Novgorod, no. 3, 2009, pp. 128-134 (in Russian) / Клинов М.С., Крюков В.А. Автоматическое распараллеливание Фортран-программ. Отображение на кластер. Вестник ННГУ, no. 2, 2009 г., стр. 128-134.

[9]. Bakhtin V.A., Kolganov A.S. et al. Methods of dynamic tuning of DVMH programs on clusters with accelerators. In Proc. of the International conference «Russian Supercomputing Days», 2015, pp. 257–268 (in Russian) / Бахтин В.А., Колганов А.С. и др. Методы динамической настройки DVMH-программ на кластеры с ускорителями. Труды международной конференции «Суперкомпьютерные дни в России», 2015, стр. 257-268.

[10]. Kataev N.A. Application of the LLVM compiler infrastructure to the program analysis in SAPFOR. Communications in Computer and Information Science, vol. 965, 2018, pp. 487-499.

[11]. Kataev N. A., Smirnov A.A., Zhukov A.D. Dynamic data-dependence analysis in SAPFOR. CEUR Workshop Proceedings, vol. 2543, 2020, pp. 199-208.

[12]. Kataev N.A., Kolganov A.S. Additional parallelization of existing MPI programs using SAPFOR. Lecture Notes in Computer Science, vol. 12942, 2021, pp. 41-52.

[13]. K10 Heterogeneous cluster K10. Available at https://www.kiam.ru/MVS/resourses/k10.html, accessed 16.09.2022.

[14]. Zima H., Bast H., Gerndt M. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. Parallel Computing, vol. 6, issue 1, 1988, pp. 1-18.

[15]. Kruse M. Introducing Molly: distributed memory parallelization with LLVM. arXiv.1409.2088, 2014, 9 p.

[16]. Grosser T., Groesslinger A., Lengauer C. Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters, vol. 22, issue 04, 2012, 27 p.

[17]. Lattner C., Adve V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04). 2004, pp. 75-86.

[18]. Bondhugula U. Compiling affine loop nests for distributed-memory parallel architectures. In Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1-12.

[19]. Bondhugula U., Hartono A. et al. A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Notices, 43, issue 6, 2008, pp. 101-113.

[20]. Banerjee P., Chandy J.A. et al. The Paradigm compiler for distributed-memory multicomputers. Computer, vol. 28, issue 10, 1995, p. 37-47.

Information about authors / Информация об авторах

Никита Андреевич КАТАЕВ – научный сотрудник. Сфера научных интересов: компиляторные технологии, параллельные вычисления, оптимизация кода.

Nikita Andreevich KATAEV – research scientist. Research interests: compilers, parallel programming, program optimization.

Александр Сергеевич КОЛГАНОВ – кандидат физико-математических наук, научный сотрудник. Сфера научных интересов: параллельные вычисления, оптимизация кода, графические процессоры.

Alexander Sergeevich KOLGANOV– PhD, research scientist. Research interests: parallel programming, program optimization, GPUs.