On the Possibility of Parallel Programs Synthesis from Algorithm Graph Specification

Ark.V. Klimov[0000-0002-7030-1517]

Institute of Design Problems in Microelectronics (IPPM) of RAS
arkady.klimov@gmail.com

Abstract. To solve the actual problem of automated parallel programming, the UPL language is proposed for specifying the M-graph of the algorithm and a roadmap for generating programs on its basis for various parallel platforms in automatic mode is presented. For this purpose, in addition, the user must specify the distribution functions that map calculations to the platform resources. The algorithm for generating a loop nest by the M-graph of the algorithm and the combined distribution function is described, and its operation is demonstrated by a simple example.

Keywords: Parallel Programming, Dataflow Computation Model, Algorithm Graph, Graph Representation Language, Distribution Function, Automated Program Synthesis.

1 Introduction

At present, program efficiency is impossible without parallelism. But, unlike the classical Von Neumann type sequential computer, there is still no common computation model and a corresponding programming language that could automatically be translated with acceptable efficiency (like C language) to all existing platforms of parallel and distributed computations. Worse, there are many different types of platforms (OpenMP, MPI, CUDA, TBB, ..., FPGA), for each of which one has to radically rebuild the entire algorithm.

As an answer to this situation, appeared the AlgoWiki project [1]. Its goal was to present all known fundamental computational algorithms in a way as to first describe the algorithm in some abstract form independently of the platform, and then, in the second part, to describe the peculiarities of various implementations for different specific platforms. Now, more than a hundred different algorithms are so described.

For the abstract representation of algorithms, the concept of an algorithm graph is introduced, in which information links between computing nodes are explicitly expressed: nodes represent calculations, arcs represent connections between outputs and inputs of various nodes. Unfortunately, the AlgoWiki project offers no formal language for describing the abstract algorithm graph, regarding which it was possible to raise questions about verification and further transformations of algorithms, including the generation of code for various platforms. Here, we try to fill this gap.

To this end, having specified the concept of an algorithm graph (Section 2), we propose an algorithm graph representation language UPL (Section 3). This language can be considered as a programming language with full-fledged semantics that allows it to be executed as a program on some abstract machine (Section 4). To turn a graph into an efficiently working code, we first need to distribute the computations by space (placement, section 5) and time (scheduling, section 6). Section 7 introduces the concept of a unified distribution function, section 8 describes a general method for constructing it, and section 9 presents a universal method for generating code from an algorithm graph and a distribution function. The method is illustrated with a simple example. The novelty of our approach is that, unlike many systems for automatic parallelization, where these distributions are automatically generated, we consider them as a tool for manual programming and compilation control: by setting these functions, the user controls the code generation, whose efficiency depends from the user selection. In the paper we try to show that the concept of distribution function not only allows to automatically generate the code according to it, but also is a convenient and adequate means to control the properties of the generated code.

2 The Concept of Algorithm Graph

First of all, let us pay attention to the ambiguity associated with the concept of an algorithm graph. We ought to distinguish between a *computation graph* in which the nodes are individual computational acts and the *algorithm graph* properly, which in a compact form describes all kinds of computation graphs that arise when this algorithm works with specific input data. The first can be huge, while the second can usually fit in one or more pages. For brevity and definiteness, we will call it the R-graph (running graph, and also - *lattice graph* due to the Rostov-on-Don school [3]). And the actual graph of the algorithm will be called the M-graph (macro-graph, meta-graph, etc.). To avoid ambiguity, the nodes of the R-graph will also be called virtual[1] nodes. In the literature, the terms graph of information dependencies or information graph are also found. These can be understood as both an M-graph and an R-graph, depending on the context.

An M-graph can be build automatically from a fragment of code in C or Fortran if the fragment belongs to a so-called *affine* class. Such a fragment can contain only loops and assignment statements, where all accesses to array elements and loop boundaries are affine expressions in enclosing loops variables and fixed structural parameters (for example, `2*i+j`, `N-1`). Conditional statements with affine conditions (like `i<j`, `i==N`) are also allowed. In affine expressions, whole division by an integer constant can be used as well (in which case expressions are called quasi-affine).This kind of fragment is usually called Static Control Program (SCOP) and the corresponding M-graph is its polyhedral model [12].

[1] As they are all considered to virtually exist a priori, while only some of them will actually appear in the computation.

Each node X of the polyhedral model (or M-graph) stems from some assignment statement (to variable or array element) and has the following properties:

- a unique name (reflecting the name of the target array),
- a tag, or a set of indices $I=(i_1,i_2,...,i_k)$ comprised of enclosing loops variables,
- a domain D_X (the set of tag values) defined as a piecewise quasi-affine polyhedron [4], depending (linearly) on the structural parameters,
- a set of input ports $V = (v_1, v_2, ..., v_p)$, corresponding to array accesses in the right hand side,
  a code evaluating the right hand side.

From each node X in the M-graph (except output nodes) exits one or more edges, each going to an input port of a node Y. The edge properties include:

- the transmitted value calculated by the node X program,
- the target node tag, calculated by the sender according to the some formula $\varphi(I,V)$, depending on the sender' tag I and inputs V,
- emit condition (generally, the generator of output values).

Many parallelization systems for C or Fortran are based on the extraction of the M-graph from a sequential program (OPC [2], PLUTO [11]). We believe, however, that using the unified universal language for representing the M-graph itself as a starting point will be more fruitful, in particular, as it is more convenient to specify mappings in terms of nodes and their indices, rather than indices of loops and arrays.

The R-graph can be constructed from the M-graph, if you provide the necessary input data. Usually, only structural parameters do suffice, such as N and M in Fig. 2 (as it is in the case when target tag on the edge depends on sender's own indices I only, rather than on input values V as well, which is valid for polyhedral models). Normally, each node X in the M-graph generates a family of virtual nodes $X\langle I\rangle$ in the R-graph, where $I \in D_X$. Each edge in the M-graph (going from node X to port P of node Y) will generate a family of edges in the R-graph, each connecting some node instance $X\langle I\rangle$ with the port P of an instance $Y\langle J\rangle$, where $J=\varphi(I,V(X\langle I\rangle))$.

Note that there are no cycles in the R-graph; otherwise, some node would receive input depending on its result. But the M-graph usually contains cycles, on which indices change to prevent cycles in the produced R-graph (as, for example, t increments on an edge with the condition $t<M$ in Fig. 2). This requirement implies that each node of the R-graph is activated (see below) only once[2].

[2] It is not a ban however. In practice, there are programs (M-graphs) in which one node of the original M-graph is activated repeatedly with the same tag value. We consider such activations as different nodes of the R-graph.

3 The M-graph Representation Language UPL

For M-graph specification we develop the graphic language UPL (Universal Parallel Language) [5,6]. It can be treated as a generalization of the polyhedral model [12], in which tag expressions on edges are not required to be (quasi-)affine.

As an illustration and a working example, consider the problem of solving the heat equation in the one-dimensional case by the classical method with a simple three-point pattern. The corresponding C code is shown in Fig. 1.

```
for(int t=1; t<=M; t++){
  for(int i=1; i<N; i++)
    B[i] = 0.33333*(A[i-1]+A[i]+A[i+1]);
  for(int i=1; i<N; i++)
    A[i] = B[i];
};
```

Fig. 1. Program Heat1 in C

From this code, the M-graph shown in Fig. 2 (left) was automatically constructed. It was produced with the assumption that $M>1$ and $N>2$, and that the whole array A [0:N] is the input as well as the output. This M-graph exactly mimics the Heat1 program up to transition nodes, such as node A (which have a single input that is transmitted to output as it is).

In Fig. 2 (right) is shown the corresponding R-graph for $N = 5$ and $M = 3$. Each nodes have an index with one or two components, according to the number of cycles enclosing the corresponding statement. Within the text, indices are usually written in angle brackets, and in hexagons on the graph. A regular node is drawn as a block with one or more named input ports, shown as truncated yellow circle attached to the block. A transition node is depicted as a yellow oval. Node tag is declared on a pink hexagon. Each arc (edge) has a label with a list of index expressions in a blue hexagon: this is the tag of the target node. Near the beginning of the arc, the emission condition may be written. An expression that defines a transmitted value normally is written in a small box on the arc; otherwise it is either the formula in the sender node or the value of its first input port.

All index expressions standing on arcs calculate the recipient index based on the sender context. We say that the M-graph works in the *scatter paradigm*, which means that a producer knows where and to whom the produced value should be transmitted. If you revert all arcs, making index expressions to calculate the sender index in the context of the receiver, you get a graph in the *gather paradigm*, where a consumer knows where to get the input data from, but knows nothing about where to transfer the results - it just puts them into their output data storage, and whoever needs them will pick them up from there. For polyhedral models, such a transformation is always possible, but not always in the general case. The scatter paradigm is, in principle, better suited for parallel and distributed setting, since it provides just one-way data transfer along the arcs, while in the gather paradigm the data is first saved and then transmitted in response to a request.

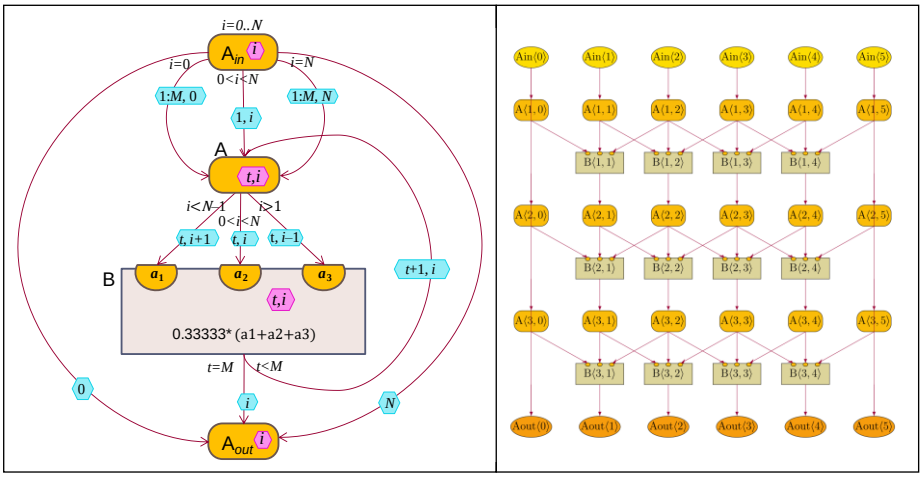

Fig. 2. Algorithm graph for Heat1: left – algorithm graph itself, or M-graph, right – computation R-graph for N=5, M=3

Note that in the M-graph the information about where the data was allocated (here arrays A and B) is lost. Instead, collections of nodes and their associated collections of inputs and outputs appeared. If you add information about ordering nodes in time and mapping data to addressable memory, the source code very close to the original one can be recovered.

4 Dataflow Computation Model

An M-graph can be interpreted as a self-contained program that can be executed in a dataflow computation model defined as follows. Tokens with data are supplied to the inputs of the nodes. Each virtual node (with a specific tag) is activated when there are tokens on all its inputs (we call such set of tokens *active combination*). Activated node makes its program to execute, which results in sending new tokens to the inputs of other virtual nodes.

An abstract machine that performs the described work is shown in Fig. 3. It contains a storage device called the Workspace (WS), which receives tokens. According to certain rules, it searches for active combinations (AC) in it – these are sets of tokens directed to the inputs of a unique virtual node, one for each input. The operation step consists in selecting one AC in the WS and activating the corresponding virtual node. At the same time, all used tokens are instantly (atomically) deleted from the WS (or, if a token had a multiplicity greater than 1, it remains in the WS with decremented multiplicity), after which a *packet* (which is a tasks to execute the node program) with token values and common tag is transmitted to the Executing Unit (EU). As a result of execution, new tokens are generated and injected into the Input Buffer (IB), from where they are transmitted with some delay one at a time to the WS. The ma-

chine stops when the IB is empty and no more ACs can be found in WS. The tokens remaining in the WS constitute the result of the work.

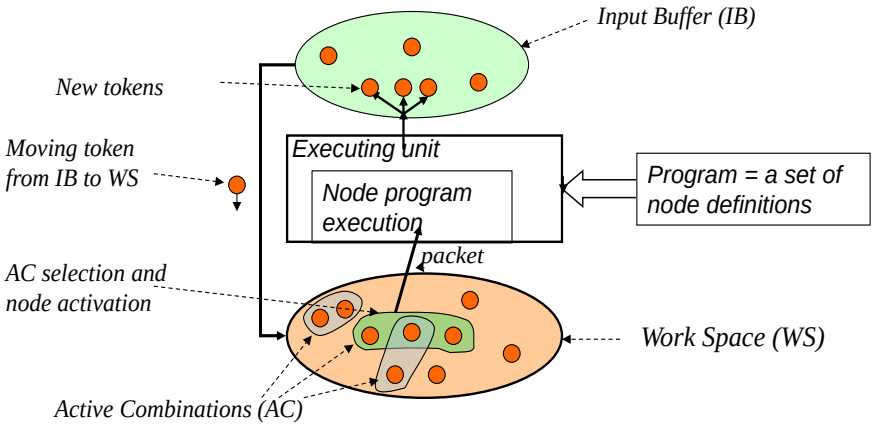

Fig. 3. The Virtual Abstract Computing Machine

The described machine can be implemented either in hardware [8] or in emulation mode on a single processor or multi-processor system [9]. This is convenient for debugging and investigating the algorithm, but it may be not efficient due to high overhead (for transferring a token to WS, searching for AC, activation, transferring a packet to a EU, etc.). Therefore, it would be useful to have alternative mechanisms for mapping the M-graph into efficient program for existing (super-)computers. Additional information may be required for this. Below we consider a method of such a mapping based on information about distribution of virtual nodes in space and time.

5 Distributing Computations in Space

The UPL abstract machine, informally described above, has a remarkable property: it can be distributed.

Suppose we have P machines (cores) connected by a communication network. Suppose that for each node X a distribution function Π_X is specified, that maps each tag $I \in D_X$ to a core number $p = \Pi_X(I) \in P$. After emitting each new token targeted to node X⟨I⟩ the core calculates $\Pi_X(I)$ and sends the token to the core number p. Clearly, all tokens directed to a single virtual node X⟨I⟩ will fall into the same core $p = \Pi_X(I)$ and activate the node X⟨I⟩ there in. Thus, everything will work as if it were a single core.

Apart from some limitations, we can assume that the function Π can be selected arbitrarily: the program (M-graph) will work correctly in any case, except that the cost of transferring tokens and the amount of idle time due to load imbalance may vary. Hence, we need to choose the distribution function Π so as to minimize the vol-

ume of communication between cores while preserving good load balance. Also, it is important to minimize the number of inter-core transmissions of higher latency on critical paths (the longest chains of virtual nodes transmitting data to each other).

If there is a hierarchy in the structure of the network, it will be necessary to optimize at all levels. All three factors: transmission volume, delays on critical paths and load balance are closely interrelated and should be optimized together when choosing a function Π.

6 Scheduling or Distribution in Time

According to the dataflow computation model, each virtual node can be executed when all its arguments are delivered. We say that the virtual node v depends on the virtual node *u*, denoting it as $u \to v$ if *u* creates a token used in the activation of *v*. The transitive closure of this relation (denoted by $\twoheadrightarrow$) creates a base (strict) partial order-dock on the set of all virtual nodes that arise in the task. A linear or partial order (<) defined on a set of virtual nodes is called *valid* if it strengthens the base order, that is

$$u \to v \Rightarrow u < v.$$

If the order is linear, then it defines a strictly sequential traversal (execution) of the R-graph. If the order is partial, then it leaves a room for parallelism.

Just as the distribution over the cores, that is, *in space*, can be specified by the function Π, the computation order can be defined by a distribution *in time*, or a *schedule* Θ, which also depends on the node and its tag and yields the logical time at which the node instance must be performed.

Composing a code (sequential or parallel) that traverse the whole R-graph is also a way to define the order of computation. But it seems to be much more difficult for a human than to just write down a distribution function. That is why we consider it important to be able to automatically generate a program code from the given space and time distribution functions.

7 A Combined Distribution Function

Often, the multidimensional space of processors or threads is considered when the processors are located in the nodes of multidimensional lattice, and a processor number is a vector of integers. In this case, the function Π should produce vector values. Similarly, time can be multidimensional, requiring the function Θ be vector. In this case, the lexicographical order is implied. We introduce a combined multidimensional function Ψ, all of whose components are either temporal or spatial, the latter will be underlined.

On the value set of the function Ψ we define a partial lexicographic order $\succ$ as follows. To compare the two vectors, we find the first (from the left) component in which they differ. If this component is temporal, the values are comparable and the one with larger component is larger. Otherwise, the values are incomparable (equal ones are also incomparable).

The function Ψ (as well as Θ) must be monotonic. Formally, we call Ψ *valid* if for each u and v:

$$u \to v \Rightarrow \Psi(v) \succ \Psi(u) \tag{7.1}$$

Compliance with this requirement must be verified before using the Ψ function to generate code.

8 How to Write Good Distribution Function

In principle, you can write the function Ψ "from your head" if you are well aware of the structure of the R-graph of the problem. And this is still easier than implementing the same idea in the form of program code traversing the R-graph. Besides, it is possible to show a general search direction, applicable to a wide class of problems, primarily to problems with homogeneous graphs. This technique is based on the ideas of [13], with the addition that they are used to write out a multidimensional space-time distribution function, which is then fed as an additional input to an automatic synthesizer of program code.

Let us demonstrate this technique using the same Heat1 problem as an example. It is important that the nodes of the R-graph be embedded in a single integer space I^n. This automatically takes place here, since there is only one essential node in the M-graph, $B\langle t,x\rangle$. For brevity, we will identify the virtual nodes with the corresponding vectors and denote them in bold letters with lowercase letters: $\boldsymbol{u}$, $\boldsymbol{v}$, ... We start with the definition of the set *Dist* of all dependency vectors:

$$Dist = \{\boldsymbol{u} - \boldsymbol{v} \mid \boldsymbol{v} \to \boldsymbol{u}\}.$$

Here it consists of three vectors: ⟨1, –1⟩, ⟨1.0⟩, ⟨1.1⟩. We draw them applied to some common point, as in Fig. 4. They produce a *dependence cone*, the set of all their linear combinations with non-negative coefficients. Its main property is that only its points can be reached through the relation ↠ from the vertex of the cone.

Consider the hyperplane $\varphi_{\boldsymbol{h}}$ defined by the equation $\boldsymbol{h}\cdot\boldsymbol{v}$ = const (with the normal vector $\boldsymbol{h}$ and offset c) such that the vertex of the dependence cone lies on it and the whole cone lies in its positive side. We will call it a *tiling hyperplane*. It divide the whole space into a conditional *past* and a conditional *future*. No dependence can cross it from future to past.

A family of parallel tiling hyperplanes $\boldsymbol{h}\cdot\boldsymbol{v}=km$, where m is a constant (step), and k is any integer, will divide the entire graph into “hyperbands” that can be executed one after another. If we take n such linearly independent hyperplanes, then their families $\boldsymbol{h}_i\cdot\boldsymbol{v} = km_i$ will divide the entire space into parallelepiped blocks with the block coordinates $k_i = \boldsymbol{h}_i\cdot\boldsymbol{v}/m_i$. (Note that these are functions of the original virtual coordinates $\boldsymbol{v}=\langle v_1,...,v_n\rangle$). Moreover, the block K must be executed before the block L if (and only if) $k_i \leqslant l_i$ for all i. For example, it may be a lexicographic order. And if the blocks are coordinate-incomparable, then they are independent and can be executed in parallel. In particular, all the blocks lying on the block hyperplane $\Sigma k_i = \tau = const$ can be executed in parallel, and the hyperplanes themselves can be executed in the increasing order of τ.

Thus, we can put $\tau=\Sigma k_i$, which shows the distribution in time, as the senior element of the distribution function. The next $n-1$ elements are other linear combinations of k_i, which together with the first element form a basis in the space of blocks (for example, k_n-k_1, k_2, k_3,..., k_{n-1}). They must be underlined to indicate parallelism between those blocks, and their proper values can be used as locations, that is, the distribution over space. And finally, we must define the order of nodes within blocks, for example, by writing out their original indices $\boldsymbol{v}$ (or any functions from them) in a certain order, such that the final combined distribution function

$$\Psi(\boldsymbol{v}) = \langle \Sigma k_i , \underline{k_n-k_1}, \underline{k_2}, \underline{k_3,\ldots, k_{n-1}}, v_1,\ldots,v_n\rangle$$

be valid (see (7.1)).

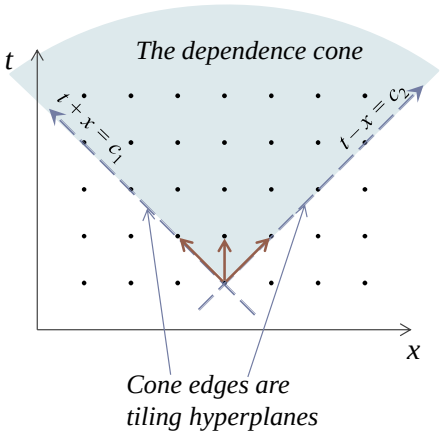

Fig. 4. The set *Dist* and dependence cone for Heat1

Back to Heat1. Tiling hyperplanes can be selected from the faces of a dependence cone. In Fig. 4 they are lines $t+x=c_1$ and $t-x=c_2$. The parallelepiped blocks formed by them are shown in Fig.5a. Here a and b are block coordinates, d is the block size, $\Pi=p(t,x)$ is the distribution in space, $\Theta=s(t,x)$ is the distribution in time. And the whole combined distribution function (for node B) looks like

$$\Psi_B(t,x) = \langle\Theta,\underline{\Pi},t,x\rangle.$$

9 Code Generation from Distribution Function

Consider the work of the proposed code generator for problem Heat1. In Fig. 5 the R-graph of the problem is shown labeled by the values of the functions $\Pi=p(t,x)$ and $\Theta=s(t,x)$. The functions p and s are chosen such that the inner loop goes within the blocks shown. The values p,s are written on the blocks. An example is taken from our work [7], where the generation result for variant (a) is given. With this traversal method (we call it Rhomb), performance has greatly increased (compared to the original iterative traversal) due to more efficient use of the cache.

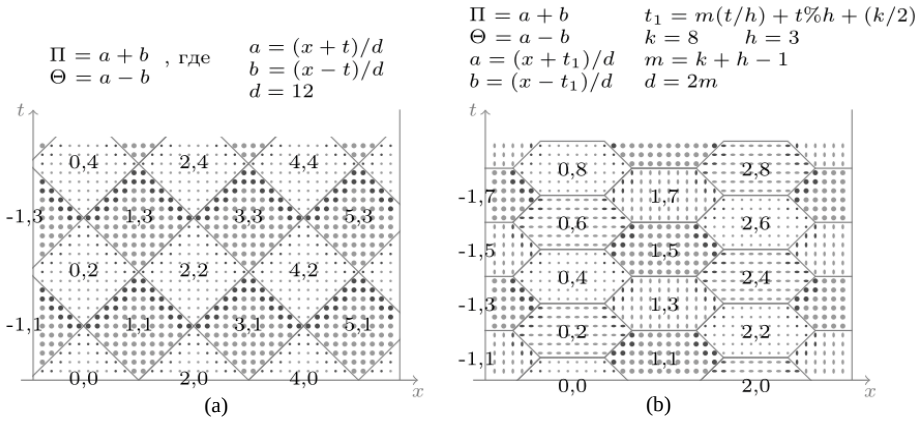

Fig. 5. Variants of the distribution of grid nodes in space and time for problem Heat1

Here we consider another option (b), in which the blocks are composed of trapezoids (named Trap). It can be specified by the following functions:

$$\Pi(t,x) = p(t,x) = (x+t_1)/d + (x-t_1)/d, \qquad (9.1)$$
$$\Theta(t,x) = s(t,x) = (x+t_1)/d - (x-t_1)/d,$$

where

$$t_1 = m(t/h) + t\%h + k/2,\ d = m,\ m = k + h - 1,\ k = 8,\ h = 3$$

Here / and % – are whole division and remainder respectively, the remainder has the sign of the divisor. It is obtained from option (a) by time replacement $t \leftarrow t_1$. Essentially it is derived by erasing some horizontal lines. The resulting distribution is beyond the above scheme. It was motivated by the hope that it will be faster as it does not have short cycles in x.

As before, the combined distribution function has the form

$$\Psi_B(t,x) = \langle \Theta, \underline{\Pi}, t, x \rangle.$$

It will produce the loop nest of the form:

```
int s,p,t,x;
for (s=□;s<□;s++)
  #pragma omp parallel for
  for (p=□;p<□;p++)
    if (exist(s,p))
      for(t=□;t<□;t++)
        for(x=□;x<□;x++)
         B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
```

Here, □-s mark places of loop boundaries to be determined. In front of the body of each cycle, there may appear a test with predicate `exist` that checks for the non-

emptiness. Here it appeared only for loop over p: a block for indices (*s*,*p*) exists only when *s* and *p* have the same parity.

To determine the boundaries of the loops, we consider the iteration space

$$D = \{ (t,x) \mid 1 \leqslant t \leqslant M,\ 1 \leqslant x < N \}$$

and its image under the action of the function Ψ:

$$\Psi(D) = \{ (s,\underline{p},t,x) \mid 1 \leqslant t \leqslant M,\ 1 \leqslant x < N,\ p = p(t,x),\ s = s(t,x) \}. \qquad (10.2)$$

This is a quasi-affine polyhedron in I^4.

Our algorithm for constructing a loop nest is organized in the spirit of [10], but with some modifications. Components of Ψ(D) are used as loop variables: here they are *s*,*p*,*t*,*x*. Loops will be nested in this order (except that underlined components may be interchanged).

Since the polyhedron D is defined by a system of quasi-affine inequalities, to determine the boundary of a cycle for a variable, for example, *t*, it suffices to choose an inequality with the desired direction, which includes only this and the left variables, in this case, *s*, *p*, *t*. To obtain such inequalities, it is necessary to project the polyhedron *D* onto the subspace over these variables. We do it by the Fourier-Motzkin algorithm for the step by step elimination of "extra" variables. Denote it as

D_{vars} = Fourier(*D*,*vars*)

where *vars* is a list of remaining variables.

The algorithm also uses the function ElimRedund (*CTX*, *D*), which removes redundant constraints in *D* under the context condition *CTX* (those which follow from CTX together with other conditions from D).

Algorithm of making loop nest boundaries

Given:

pars – static parameters list;

vars = $v_1,v_2,...,v_k$;

D (a set of quasi-affine inequalities depending on *pars*+*vars*);

Result:

A split $D = D_1+D_2+...+D_k$, where each D_i depends only on variables v_1, ..., v_i (and static parameters *pars*). Each D_i contains at least one inequality restricting the variable v_i. from below/above.

begin

CTX := list of restrictions on *pars*.

for all *i* **from** 1 **to** *k*

vs = start(*vars*, *i*) ; //– first i variables from *vars*

D_{vs} := Fourier (*D*, *pars*+*vs*) ;

D_i := ElimRedund (*CTX*, D_{vs}) ;

CTX := *CTX*+ D_i ;

D := ElimRedund (*CTX*, *D*) ;

end

Now we find the boundaries of the cycle in the variable v_i. For the lower / upper boundary, we choose one or several inequalities from D_i that bound v_i from below / above and get the restriction in terms of the remaining variables v_1, ..., v_{i-1} (combining the results from several inequalities by max/min).

As applied to function (9.2), the algorithm failed to work well: the result is complex and inefficient. The block boundaries are complicated as they involve two inequalities. To get around this, we break each hexagonal block with the index (p, s) into two halves - upper and lower. Then in each half there will be only one restriction from left and from right. In terms of the distribution function Ψ, we introduce an additional variable, $q = t/h–s$, h=3, and include it in Ψ between p and t:

$$\Psi(D) = \{ (s,\underline{p},[q],t,x) \mid 1 \le t \le M,\ 1 \le x < N,\ p=p(t,x),\ s=s(t,x),\ q=t/h–s \}. \qquad (10.3)$$

To get the desired effect, we enclosed q with square brackets, which is a requirement to statically unroll the respective cycle. The resulting C code (with OpenMP directive) is shown in Fig. 6.

```
#define B(t,x) BB[t%2,x]
double A[N+1],BB[2,N+1];
int s,p,t,x;
for (int x=0; x<=N; x++)
  B(0,x)=A[x];
for (int s=0; s<=(M+3)/3; s++)
  #pragma omp parallel for
  for (int p=-1; p<=(N-4)/10; p++)
    if ((p+s)%2 == 0) {
      // q=-1
      for(int t=max(1,3*s-3); t<=min(M,3*s-1); t++)
        for(int x=max(1,10*p-t+3*s+3);
                x<=min(N-1,10*p+t-3*s+16); x++)
         B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
      // q=0
      for(int t=max(1,3*s); t<=min(M,3*s+2); t++)
        for(int x=max(1,10*p+t-3*s+4);
                x<=min(N-1,10*p-t+3*s+15); x++)
         B(t,x)=0.33333*(B(t-1,x-1)+B(t-1,x)+B(t-1,x+1));
      };
for (int x=0; x<=N; x++)
  A[x]=B(M,x);
```

Fig. 6. An OpenMP code generated for Heat1 according to function (10.3)

Initially, the constructed code used the two-dimensional array `B(t,x)` representing the output of the virtual node B⟨t, x⟩. Its size is extremely large, since each element is assigned just once. So we need to map it to a small array so that its elements are used repeatedly, but at the same time the necessary elements were not overwritten.

In this example, we use the observation[3] that when writing the element `B(t+2,x)`, the element `B(t,x)` is obviously not needed (since all uses of the latter are among the elements on which the former depends). Therefore, nothing prevents the virtual elements `B(t,x)` from being mapped as `BB[t%2,x]`.

We see that even for such a simple example, the complexity of writing code significantly exceeds the complexity of writing the corresponding distribution function (10.3), not to mention the high probability of errors. And for similar task of dimension 2 or 3 it is practically impossible to write such code manually. Note that errors in the distribution function (that pass the check for the validity) will not lead to an incorrect program (provided that the generator is correct).

Such intricate distributions are justified if they give a gain. Fig. 7 shows the results of performance measurements for the following options:

- Base1 – the source code from Fig.1 with both inner loops parallelized,
- Base2 – the same, but the copy loop is replaced with computation step with A and B exchanged,
- Rhomb – according to Fig. 5(a),
- Trap – according to Fig. 5(b).

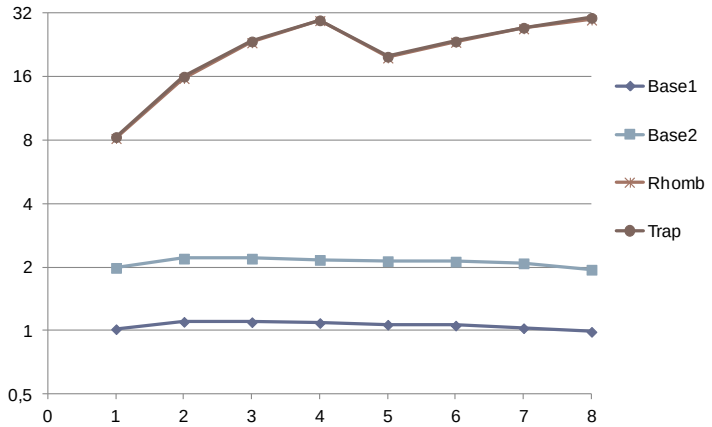

Fig. 7. Performance in GFlops dending on number of threads

The measurements were carried out on a PC with a 4-core Intel Core i7-2600 3.4 GHz processor using the compiler from Microsoft Visual Studio Community-2019 with 1-8 threads with parameters: N = 2,000,000, M = 5000, d = 300, h = 125. We see that the base options are not parallelized at all, since they work at the limit of memory access speed. It is important here that the task data does not fit in the entire

[3]Automation of this task is a subject of future research.

L3 cache. At the same time, Base2 works twice as fast, since it avoids plain copying. The Rhomb and Trap options give the same acceleration, in which 4 times (compared to Base2) gives the block distribution itself, and another 4 times gives parallelization over cores and threads. It turned out that the Trap option is no better than Rhomb, but to get it clear, we had to make code and test it. We would never do it without automatic code generation.

10 Conclusion

The work demonstrated the possibility and method of automatic synthesis of parallel programs (here for the OpenMP platform) by its specification in the form of a flow graph of the algorithm. To do this, the user must additionally provide a mapping to machine resources (primarily space and time) in the form of distribution functions of a quasi-affine class. It is important that the choice of the function cannot violate the correctness of the program, unless, of course, the compiler is not correct. In order to construct the correct code, the compiler must be able to efficiently manipulate with quasi-affine expressions.

There are many works on the automatic synthesis of parallel programs, in which the distribution functions are selected also automatically. However, the class of such functions, as a rule, is limited to purely linear ones, with possible division only as a last step. We work with a wider class of functions, among which there are, possibly, giving higher results in performance. At the same time, we suggest to specify them manually as a way of programming, although do not reject some automation of this matter. In paper [6], the author gives an overview of similar works and compares his own approach with them.

The following problems require further work:

- Making resulting code in C (at present we just generate loop boundaries and insert them manually).
- Mapping virtual arrays to real ones.
- Preserving nonlinear symbolic parameters (here d and h).
- Doing the same for dimension 2 and 3 and for other tasks.
- Comparing the obtained performance with existing results.

The work is performed under the partial support of Russian Foundation for Basic Research, projects 17-07-00324 and 17-07-00478.

References

1. AlgoWiki Project, http://algowiki-project.org, last accessed 2019/12/10.
2. Stejnberg, B.Ya: Otkrytaja rasparallelivajushchaja sistema. In: PACO 2001, Trudy mezhdunarodnoj konferencii “Parallelnye vychislenija i zadachi upravlenija”, pp. 214–220, IPU RAS, Moscow (2001).

3. Shulzhenko, A.M.: Reshetchatyj graf i ispolzujushchie ego preobrazovanija v Otkrytoj rasparallelivajushchej sisteme. In: Nauchnyi servis v seti Internet: trudy Vserossiiskoi nauchnoi konferentsii (19–24 sentiabria 2005 g.), pp. 82–85. Novorossiisk (2005).
4. Guda, S.A.: Operations on the tree representations of piecewise quasi-affine functions. Informatics and applications, 7(1), pp. 58–69 (2013) (In Russian).
5. Klimov, Ark. V.: O paradigme universal'nogo yazyka parallel'nogo programmirovaniya. In: Yazyki programmirovanija i kompilyatory (Programming Languages and Compilers, PLC–2017), trudy konferencii, pp. 141–146. YuFU, Rostov-na-Donu (2017).
6. Klimov, Ark.V.: Obzor podkhodov k sozdaniyu multi-platformennoj sredy parallelnogo programmirovaniya. In: Nauchnyi servis v seti Internet: trudy XIX Vserossiiskoi nauchnoi konferentsii (18–23 sentiabria 2017 g.), pp. 260–275. KIAM, Moscow (2017), http://keldysh.ru/abrau/2017/19.pdf, last accessed 2019/12/10.
7. Klimov, Ark. V.: Towards automatic generation of stencil programs with enhanced temporal locality. Program Systems: Theory and Applications, 9:4(39), pp. 493–508 (2018).
8. Stempkovskij, A.L., Levchenko, N.N., Okunev, A.S., Cvetkov, V.V.: Parallel Dataflow Computing System: Further Development of Architecture & Structural Organization of the Computing System with Automatic Distribution of Resources. Informacionnye tehnologii, (10), pp. 2–7 (2008) (In Russian).
9. Zmeev, D.N., Klimov, A.V., Levchenko, N.N., Okunev, A.S., Stempkovskij, A.L.: Emulation on Hardware and Software of the Parallel Dataflow Computing System "Buran". Informacionnye tehnologii, 21 (15), pp. 757–762 (2015) (In Russian).
10. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: Principles and Practice of Parallel Programming, PPoPP'91. ACM SIGPLAN Notices, 26 (7), pp. 39–50 (1991), https://doi.org/10.1145/109626.109631.
11. Bondhugula, U., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08), pp. 101–113. ACM, New York, NY, USA (2008).
12. Feautrier, P., Lengauer, C.: Polyhedron Model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer, Heidelberg (2011).
13. Irigoin, F., Triolet, R.: Supernode partitioning. 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'88), pp. 319–329. ACM, New York, NY, USA (1988), https://doi.org/10.1145/73560.73588.