Derivation of Parallel Programs of Recursive Doubling Type by Supercompilation with Neighborhood Embedding*

Arkady V. Klimov

Institute of Design Problems in Microelectronics, Russian Academy of Sciences, Moscow, Russia
arkady.klimov@gmail.com

Abstract. Recursive doubling (RD) is a well-known family of parallel algorithms that have $O(Log(N))$ parallel time when applied to vectors of length N. They are used to solve problems like Reduction, Scan (partial sums), computing some recurrences, solving tridiagonal linear system etc. Typically, in textbooks, these algorithms are explained and derived each in their own way, ad-hoc, based on the author's ingenuity. The paper suggests a general way to derive each such RD algorithm from a raw specification by an equation of the form $Y = F(Y)$, in which no idea of the RD principle is pre-contained. First, the base process P is defined, in which the general RD method is applied to the computation of the fixpoint of the specifier function F. Then, the supercompilation is applied to the base process P, eliminating all redundancies and producing a particular efficient RD code.
All necessary definitions in the area of supercompilation are presented. The supercompiled graph in all our examples has just a single loop that is built by looping strategy based on neighborhood embedding. The supercompilation process is carried out manually, all the steps being presented in the paper. The application of the method to three well-known problem examples is demonstrated.

Keywords: recursive doubling, parallel algorithms, SIMD, specification, fixpoint, supercompilation, neighborhood analysis, neighborhood embedding, program synthesis.

1 Introduction

Recursive doubling (RD) is a well-known family of parallel algorithms that have $O(Log(N))$ parallel time when applied to vectors of length N. They are used for problems like Reduction, Scan (partial sums), computing some recurrences, solving tridiagonal linear system etc. This method is also known as *cascade*, or *pairwise*, *summation* (usually for partial sums) or *cyclic reduction* (for linear

* Supported in part by the budget funding and the RAS Presidium Program № I.33P on strategic directions of science development in 2016.

recurrences and tridiagonal systems) [2,11]. With our approach they all appear to be the same, though in [2] they are presented as different methods.

Usually the RD method requires to reveal an associative binary operation, using which the problem could be reformulated and reduced to calculation of a convolution (or partial sums) of some vector with this operation. However, this operation and the respective reformulation of the problem are often difficult to guess.

We claim that these RD algorithms can be systematically derived from a problem specifications given in the form of equations like $Y = F_X(Y)$, where X is an input and Y the result. First, a general but inefficient method of computing the fixpoint solution is considered. It is the process P of symbolic evaluation of $(F_X)^n$, which leads to solution of the form $Y = (F_X)^n(\perp)$, where $\perp$ means "unknown", and n is large enough for the right hand side to compute to a result vector. To obtain $(F_X)^n$ for sufficiently large n we apply recursive doubling scheme which requires $Log(n)$ doubling steps.

However, computing this way immediately is rather expensive. To improve the efficiency, it is a good idea to apply supercompilation to the process P which would result in an efficient parallel SIMD-like program solving the original problem. We show the work of this idea for three classic examples of problems for which algorithms operating on RD principles are known. We perform the supercompilation process manually presenting all its steps in the paper. The technique has not been implemented in the computer yet.

The supercompiled program graph in all our examples has just a single loop that is built by looping strategy based on the so-called *neighborhood* embedding. The uniqueness of this strategy is that it relies on the computing process as a whole rather than just on the states the process produces.

Though all our result algorithms are textbook ones, the contribution of the paper is a general method to derive them systematically from very simple and evident problem specifications. In addition, the paper fills the gap in publications on the usage of neighborhood analysis in supercompilation, as it was proposed by V.Turchin [9,10].

The paper is organized as follows. In Section 2 a general idea of supercompilation is briefly presented. The concepts of *driving*, *computation tree*, *configuration graph*, *generalization*, *looping*, *whistle*, and *neighborhood strategy* are introduced, which will be needed below. In Section 3, the common pattern of RD code derivation from a tasks specification is presented. In Section 4, this pattern is applied to 3 distinct problem examples. Section 5 presents evaluation of the topic, identifying our contribution, and the last section concludes the paper.

2 Basics of the Supercompilation

For the purpose of this paper below we describe supercompilation briefly. For more systematic presentation the reader may refer to [3,5,6,8–10].

The word supercompilation is made of two parts: *super* and *compilation*. The name reflects the basic idea that a program is compiled by a supervision over

the process of its generalized computation. Such computation is performed over not a concrete data but rather over generalized data or states (both input and intermediate) called *configurations* and represented with symbolic configuration variables (*c-variables*) indicating unknown parts of state information. The dual nature of a configuration is that, on the one hand, it is a syntactic objects composed of c-variables, constants and constructors and, on the other hand, it can be regarded as a set of concrete states. Therefore, the configurations can be compared for set-theoretic inclusion, which usually corresponds to a syntactic embedding by means of a substitution.

A generalized computation is called *driving* and involves the following three types of actions:

1. If normal computational process assumes a choice based on current values of some c-variables, the driving process splits into two *branches* with the indication of the predicate, generating the splitting. Accordingly, for each of the branches the set of possible values for these variables is narrowed. A variable narrowed to a constant can be replaced with the constant. A variable narrowed to a structure can be split into several variables corresponding to structure fields.
2. If the process executes an operation forming a new data item from old ones, then an updated configuration with a new c-variable is created, and the arc leading to it from the old one is labeled with the substitution for this variable of an expression specifying the method of calculating its value from existing c-variables.
3. The process can perform a state reconstruction not linked with the emergence of a new data (for example, moving a structure field). Then a new node appears in the tree with a new configuration comprised of only old c-variables. None labels are placed on the arc to it. Such configuration (the source one) is called *transit*, and further it can be removed.

These three types of actions being applied iteratively to unfinished configurations produce a tree of generalized states whose branches represent generalized computation paths. These branches evolve independently of each other. Some of them may be (potentially) infinite. Now we are to fold the infinite tree into finite graph, representing the same set of paths.

To achieve this, each time a new configuration appears in the process of driving a *looping strategy* is applied which compares the new configuraton with all old ones (or their parts) seeking for their admissible generalizations into which the new one could be embedded. In case of successful embedding without extra generalization the branch evolution stops and an arc from the new configuration to the old one labeled with unifying substitution is drawn.

If a generalization is needed, the strategy decides whether to generalize and loop now or to continue driving. In the former case (generalization) the arc is drawn from the old configuration to the generalized one with the needed substitutions and all the subtree growing from it is rebuilt anew. If you are lucky, another new configuration would embed into the generalized old one, otherwise

we repeat the search. (The case of embedding into a part of an old configuration is not considered, as it is useless here. Just mention that a pruning is done and the sub-graph is built that forms a subroutine).

Choosing a strategy implies a compromise between the desire to have residual program smaller, and the desire to perform more work statically. For example, if you allow to make any possible generalization, the process will complete quickly with a small residual graph, but most of actions will remain in it. This is a *conservative* strategy. Yet if you hold from looping, more actions will be supercompiled out, but the process of supercompilation can go to infinity. Such strategy is called *aggressive*.

Often some rather aggressive strategy is used, and above it a so-called *whistle* is introduced, which occasionally generates a signal forcing a loop.

When and if the process is completed, the final graph is converted to a residual program. Wherein:

1. All elements except variables are erased in graph node configurations. The variables represent local data available at the current program point.
2. Substitution on the arc converts to assignment statements.
3. Branching converts to conditional statement.

The described process is usually carried out automatically by the supercompiler, as a program that directs the computation process and an initial configuration are given. In our case, the "program" is the general scheme of the recursive doubling method for computing a fixed point, the "initial configuration" is the task specification in the form of equation, and the process of "supercompilation" is carried out manually, imitating the work of the supercompiler.

For generalizations and looping the neighborhood strategy will be used. If a few driving steps were made from a configuration, then a *neighborhood* of this configuration is a set of configurations that includes this one together with all those from which the same steps would be made. The neighborhood (more precisely, its approximation from below) may be obtained by replacing unused parts of initial configuration with metavariables. Then, reducing metavariables to normal configuration variables, we obtain a generalization of the configuration, using which we keep all action performed earlier. A loop is made only when a new configuration (or rather its corresponding generalization) is embedded into a neighborhood of one of the previous configurations. This strategy does not guarantee the termination, but in our cases it is sufficient.

In essence, there are two major components in the supercompilation. The first is a formal inference rule, equivalent to the mathematical induction, which is provided by the mechanism of looping on condition of configurations embedding. The other component is a strategy, in our case the neighborhood one, which is nothing more than a heuristic for detecting inductive hypotheses. Together, these two components provide a fully algorithmic "intelligence" for algorithm transformations.

The idea of neighborhood strategy for supercompilation was proposed by V. Turchin [9,10]. The formalization of the concept of neighborhood and neighborhood analysis in the broader context was carried out by S. Abramov [1].

3 General Scheme of Algorithm Derivation

As an initial "language specification" we shall use the language of recursive equations. If the input is denoted by X and the output (including, possibly, interim results) by Y the specification takes the form

$$Y = \mathcal{R}(X, Y) \tag{1}$$

where $\mathcal{R}$ is an expression with variables X and Y. For a given X, equation (1) can be rewritten as:

$$Y = \mathcal{R}_X(Y) \tag{2}$$

Let us assume that the domains of X and Y have complete partial order structure and that all functions used in $\mathcal{R}$ are monotonic. Then the solution of equations of the form (2) can be written as the limit

$$Y = \lim_{n\to\infty} (\mathcal{R}_X)^n(\perp) \tag{3}$$

where the degree is the n-fold superposition of the function. In practice, one may take the value of $(\mathcal{R}_X)^n$ for some finite n, starting from which the sequence is constant. It will be the case if, e.g., expression $(\mathcal{R}_X)^n$ does not contain the argument variable Y.

For our purposes it will be enough to take the domain of vectors of length N with elements of a type T extended with value $\perp$, such that $a \succeq \perp$ for all $a \in T$. The element $\perp$ means *undefined* and relation $\succeq$ means *more (or equally) exact*. All operations ϕ over type T are extended by the rule $\phi(\dots, \perp, \dots) = \perp$ and on the vectors are defined element-wise. The vector $\perp = [\perp, \dots, \perp]$. In addition, we'll need the shift operator for vectors. Consider vector $X = [x_1, x_2, \dots, x_N]$. Then for $k \geq 0$ by definition we have

$$\begin{aligned} \mathsf{shift}(\ \ k, X) &= [0, \dots, 0, x_1, \dots, x_{N-k}], && (k \text{ zeros from left}) \\ \mathsf{shift}(-k, X) &= [x_{k+1}, \dots, x_N, 0, \dots, 0], && (k \text{ zeros from right}) \end{aligned}$$

As an example, consider a simple problem: calculation of the vector B of the partial sums of vector A. It can be defined by the following equation:

$$B = \mathcal{R}_A(B) = \mathsf{shift}(1, B) + A \tag{4}$$

Indeed, it is easy to see that

$$(\mathcal{R}_A)^k(\perp) = [a_1, a_1 + a_2, \dots, a_1 + a_2 + \dots + a_k, \perp, \dots, \perp] \tag{5}$$

and therefore for $k \geq N$ the formula (5) yields the desired result.

It is clear that to calculate $(\mathcal{R}_A)^N(\perp)$ immediately by N-fold applicaton of $\mathcal{R}_A$ is very expensive. Therefore, we first calculate the N-th degree of $\mathcal{R}_A$ using the doubling method:

$$\begin{aligned} \mathcal{F}_1 &= \mathcal{R}_A \\ \mathcal{F}_2 &= \mathcal{F}_1 \circ \mathcal{F}_1 \\ \mathcal{F}_4 &= \mathcal{F}_2 \circ \mathcal{F}_2 \\ &\dots \end{aligned} \tag{6}$$

The process (6) may stop if an expression $\mathcal{F}_{2^n}$ does not contain symbol Y (or it becomes clear that the limit is achieved). Then the desired result is $\mathcal{F}_{2^n}(\perp)$. This explains the type name of produced algorithms: *recursive doubling*.

If we perform the superposition operation $\circ$ formally, then no effect will take place, because the calculation of $F_{2^n}(\perp)$ will still have to make 2^n calls to initial function $\mathcal{R}_A$. However, making formula manipulation each step is also expensive. Thus, it is a good idea to supercompile the process (6) with unknown input A, such that only calculations dependent of A remain for dynamic execution.

As the supercompilation would only make sense for a particular problem $\mathcal{R}_A$, we give here only general comments on the scheme, and the full description of the supercompilation process will be set forth by specific examples in Section 4.

The initial configuration is the equation of the form $Y = \mathcal{R}_X(Y)$, in which X is a configuration variable indicating the input and Y is just a formal symbol. Performing the step is to move to the configuration $Y = \mathcal{R}_X(\mathcal{R}_X(Y))$, in which various simplifications are produced. Herein are used the usual properties of arithmetic operations and the following properties of the shift operator:

$$\mathsf{shift}(k, A \odot B) = \mathsf{shift}(k, A) \odot \mathsf{shift}(k, B) \tag{7}$$

$$\mathsf{shift}(k, \mathsf{shift}(k, A)) = \mathsf{shift}(k + k, A) \tag{8}$$

$$\mathsf{shift}(k, (\mathsf{shift}(k, A)) = \mathsf{shift}(k, \mathbf{1}) \cdot A, \textit{where } \mathbf{1} = [1, 1, \ldots, 1] \tag{9}$$

$$\mathsf{shift}(k, X) = \mathbf{0}, \textit{if } k \geq N \tag{10}$$

where $\odot$ is any operation on elements (such that $0 \odot 0 = 0$).

To provide the termination property of residual program the supercompiler must recognize formula that does not depend on Y. To this end, based on the property (10), we will act according to the following rule: when the formula has a sub-expression of the form $\mathsf{shift}(k, X)$, we introduce the splitting predicate $k \geq N$ and on the positive branch replace this sub-expression with $\mathbf{0}$. This will lead to possible exclusion of the argument Y from configurations on some branches, which then can become terminal.

The supercompilation will be effective if we find a common form for the steps. For doubling, it will be possible if we succeed to represent the configuration after the step in the same form as before the step, in which case the loop is possible. In case of success, the final residual graph is converted to the code in a programming language. The code will hold element-wise vector operations, which are easily parallelizable.

4 Examples

4.1 Linear Recurrence

This is a generalization of example (4). Given are two vectors A and B of length N. We are to solve the following equation

$$Y = A * \mathsf{shift}(1, Y) + B \tag{11}$$

In a more familiar mathematical language, this is equivalent to the calculation of the sequence $\{y_i\}$ defined by the following recurrence:

$$\begin{aligned} y_0 &= 0 \\ y_i &= a_i * y_{i-1} + b_i \ , \quad i > 0 \end{aligned} \tag{12}$$

Let us supercompile the computation process for solving equation (11) by doubling method. The initial configuration C0 is the original equation:

C0: $$Y = A * \mathsf{shift}(1, Y) + B$$

The doubling step is to substitute for the variable Y in the right-hand side with the right hand side itself and simplify:

C1: $$\begin{aligned} Y &= A * \mathsf{shift}(1, A * \mathsf{shift}(1, Y) + B) + B \\ &= [A * \mathsf{shift}(1, A)] * \mathsf{shift}(1 + 1, Y) + [A * \mathsf{shift}(1, B) + B] \end{aligned}$$

It is easy to notice the similarity of this configuration with the original one. But how can the machine "notice" this? Formally, it is necessary to represent both configurations as special cases (by substitutions) of a common configuration. But which generalizations should be considered valid? An answer may be given by neighborhood analysis. We carry out a step from configuration C0 to configuration C1, watching for what information from C0 representation we use. To this end, we cover the description of the configuration C0 with a translucent strip, and whenever a symbol is explicitly used in the process, we make it out of the strip. On step completion only the symbols A, B and 1 remain intact:

U0: $$Y = \boxed{A} * \mathsf{shift}(\boxed{1}, Y) + \boxed{B}$$

This means that no matter what could be in their place, the step would be carried out in the same way. Note that these parts can go without changes into new configurations after the step:

U1: $$Y = [\boxed{A} * \mathsf{shift}(\boxed{1}, \boxed{A})] * \mathsf{shift}(\boxed{1} + \boxed{1}, Y) + [\boxed{A} * \mathsf{shift}(\boxed{1}, \boxed{B}) + \boxed{B}]$$

The result of the driving step with neighborhood is shown in Fig. 1 (left). There appeared a split due to property (10) and new configurations on the arc ends. Intact parts of the original configuration are shaded.

Now we can replace all the unused parts with metavariables. Let it be $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{K}$. Considering all sorts of substitutions for them, we get a variety of configurations for which the driving step is identical to step from this configuration. In terms of metavariables it is a step from metaconfiguration

M0: $$Y = \mathcal{A} * \mathsf{shift}(\mathcal{K}, Y) + \mathcal{B}$$

to metaconfiguration

M1: $$Y = [\mathcal{A} * \mathsf{shift}(\mathcal{K}, \mathcal{A})] * \mathsf{shift}(\mathcal{K} + \mathcal{K}, Y) + [\mathcal{A} * \mathsf{shift}(\mathcal{K}, \mathcal{B}) + \mathcal{B}]$$

Thus, as a result of the neighborhood analysis a generalized configuration M0 has been built, which will be considered as a maximum permissible generalization of the initial configuration C0, to which the looping is feasible.

Recall that the rule (10) for M0 yields the predicate $k \geq N$ with the branch to the final metaconfiguration $Y = B$. Metaconfiguration M1 on the other branch should be driven further as it contains Y. But first we must check whether it falls into the processed one (M0). We see that it does, and therefore we may not proceed doing the next step but make a loop into the generalized old configuration M0. To this end, in the neighborhood M0 we replace metavariables with usual configuration variables such that both old and new configurations C0 and C1 can be obtained from it by substitutions. Those substitutions become labels on the arcs into the new generalized configuration. The result is shown in the right part of Fig. 1.

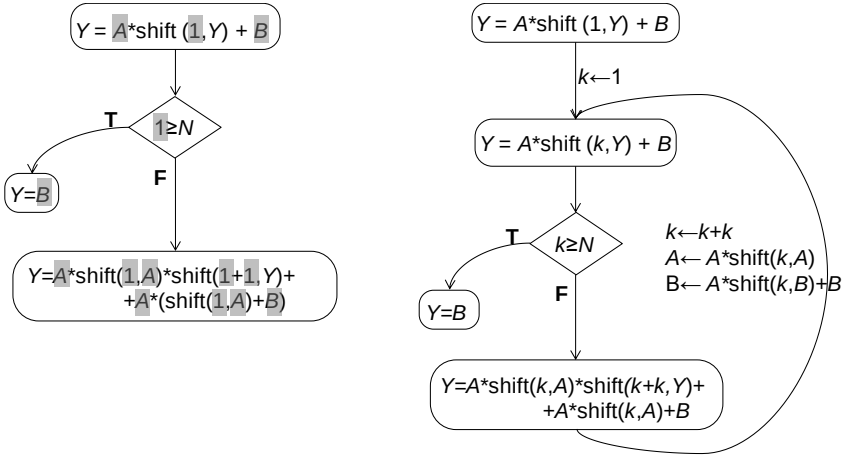


Fig. 1. Construction of the graph configurations in the Linear Recurrence. To the left is the first doubling step with its neighborhood. Untouched parts of the initial configuration are shaded. In their place, new variables are introduced (if they did not exist yet). To the right is the final graph after generalization and looping.

Now we convert the resulting graph into the program in a programming language: erase configurations (except the final) and replace the substitutions with assignment operators (changing their order and introducing interim variables if necessary). The result is shown in Fig. 2. It is a simple cyclic program built of parallel vector operations. It is easy to see that the loop performs Log (N) iterations, each taking an O(1) parallel time.

Interestingly, if the size of the vector N were a given constant, say 1000, the first driving step would lead to the lack of a lateral branch, as predicate $1 \geq 1000$ yields known **false**. So, the value of 1 would have been used, and therefore there

```
k:=1;
while k<N do
   B:=A*shift(k,B)+B;
   A:=A*shift(k,A);
   k:=k+k;
end;
Y:=B;
```

Fig. 2. The generated code for the Linear Recurrence

would be no embedding. The neighborhood strategy in this case would lead to a complete loop unfolding (here 10 rounds).

4.2 Binary Addition

Consider the task of adding two binary integers A and B. The desired result is a scheme with fast carry, which works for the $Log(n)$ clock cycles, where n is the bit size of summands.

We begin with the following system of recurrent equations defining the vectors carry C and sum S:

$$\begin{aligned} C &= A \wedge B \vee (A \vee B) \wedge \ shift(1, C) \\ S &= A \oplus B \oplus \mathsf{shift}(1, C) \end{aligned} \tag{13}$$

where $\wedge$, $\vee$, $\oplus$ are bitwise conjunction, disjunction and addition modulo 2 respectively ($\wedge$ has the highest priority).

Let us solve the equation for C by doubling. The initial configuration is

C0: $$C = A \wedge B \vee (A \vee B) \wedge \mathsf{shift}(1, C)$$

We do not show the step result (configuration C1) as it is bulky and not interesting. For configuration C0 we get the following neighborhood:

U0: $$C = A \wedge B \vee (A \vee B) \wedge \mathsf{shift}(1, C)$$

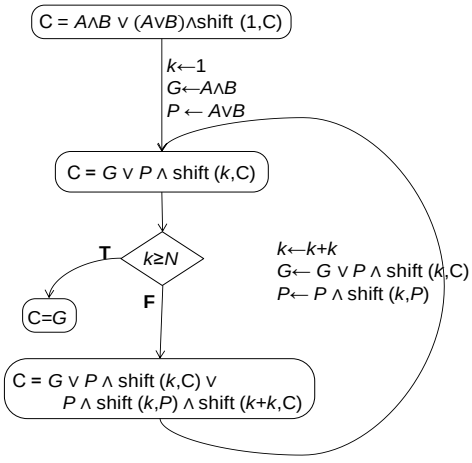
Replace unused parts (selected by gray background) with metavariables, let it be $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{K}$. Metaconfiguration

M0: $$C = \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, C)$$

goes by the driving step into metaconfiguration

M1: $$\begin{aligned} C &= \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, C)) = \\ &= \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{G}) \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{P}) \wedge \mathsf{shift}(\mathcal{K} + \mathcal{K}, C) \end{aligned}$$

One can see that metaconfiguration M1 embeds into the initial one M0. Making the needed actions for generalization and looping (structurally they are similar to the previous example, just operations are different), we get the graph shown in Fig. 3.




```
k:=1;
G:= A ∧ B;
P:= A ∨ B;
while k<N do
    G:=G ∨ P ∧ shift(k,G);
    P:=P ∧ shift(k,P);
    k:=k+k;
end;
C:=G;
S:= A ⊕ B ⊕ shift(1,C);
```

Fig. 3. To the left is the finished configuration graph for Binary Addition problem. To the right is the corresponding program "with fast carry".

4.3 Solving tridiagonal linear equations

We write the tridiagonal system of equations as

$$X = A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C \tag{14}$$

where A, B, C are input vectors and X is the vector of unknowns.

A distinctive feature of this problem is that the doubling step transforms not only the right hand side, but the entire equation. The initial configuration is:

C0: $$X = A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C$$

We perform the step and simplify the result with the use of the property (9):

C1:
$$\begin{aligned} X = {} & A \cdot \mathsf{shift}(1, A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C) + \\ & B \cdot \mathsf{shift}(-1, A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C) + C \\ = {} & A \cdot \mathsf{shift}(1, A) \cdot \mathsf{shift}(1 + 1, X) + \\ & B \cdot \mathsf{shift}(-1, B) \cdot \mathsf{shift}(-(1 + 1), X) + \\ & A \cdot \mathsf{shift}(1, C) + B \cdot \mathsf{shift}(-1, C) + C + \\ & A \cdot \mathsf{shift}(1, B) \cdot X + B \cdot \mathsf{shift}(-1, A) \cdot X \end{aligned}$$

By grouping linear on X terms to the left and dividing both parts by the coefficient of X, we reduce the configuration C1 to the following form:

C1′:
$$\begin{aligned} X = {} & (A \cdot \mathsf{shift}(1, A))/D \cdot \mathsf{shift}(1 + 1, X) + \\ & (B \cdot \mathsf{shift}(-1, B))/D \cdot \mathsf{shift}(-(1 + 1), X) + \\ & (A \cdot \mathsf{shift}(1, C) + B \cdot \mathsf{shift}(-1, C) + C)/D, \end{aligned}$$

where $D = 1 - A \cdot \mathsf{shift}(1, B) - B \cdot \mathsf{shift}(-1, A)$. Consider the neighborhood U0 of initial configuration, where all intact parts are shaded:

U0: $$X = \boxed{A} \cdot \mathsf{shift}(\boxed{1}, X) + \boxed{B} \cdot \mathsf{shift}(-\boxed{1}, X) + \boxed{C}$$

Note that the two occurrences of the number 1 were touched only by comparing them with each other (through the use of property (9)), so, in the neighborhood expression M0, they should be represented with the same metavariable $\mathcal{K}$.

M0: $$X = \mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, X) + \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, X) + C$$

After the step, at both corresponding locations in metaconfiguration M1 occurs the same sub-expression $(\mathcal{K} + \mathcal{K})$, which allows to embed.

M1: $$\begin{aligned} X =& (\mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{A}))/D \cdot \mathsf{shift}(+\mathcal{K}, X) + \\ & (\mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, \mathcal{B}))/D \cdot \mathsf{shift}(-(\mathcal{K} + \mathcal{K}), X) + \\ & (\mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{C}) + \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, \mathcal{C}) + \mathcal{C})/D, \end{aligned}$$

where $D = 1 - \mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{B}) - \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, A)$ Thus, the metaconfiguration M1 completely embeds into the neighborhood metaconfiguration M0, which results in a final graph on Fig. 4. After erasing all the excess, we obtain the residual program shown in Fig. 5.

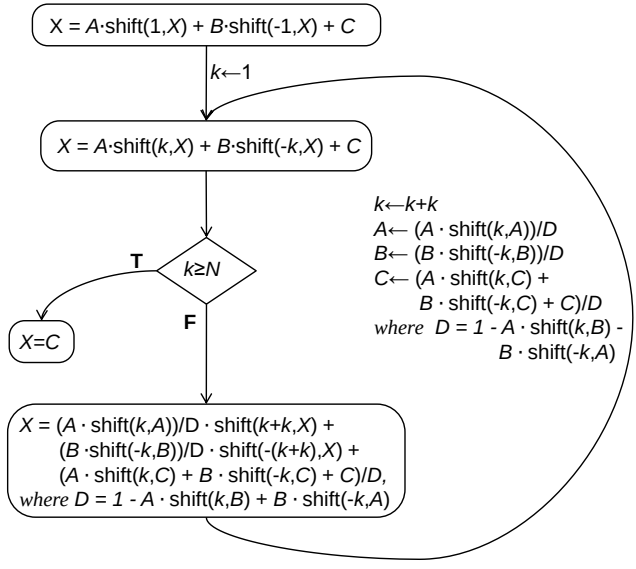


Fig. 4. Configuration graph for solving tridiagonal system of equations

```
k:=1;
while k<N do
   D:=1A*shift(k,B)B*shift(k,A);
   C:=(A*shift(k,C)+B*shift(k,B)+C)/D;
   A:=(A*shift(k,A))/D;
   B:=(B*shift(k,B))/D;
   k:=k+k;
end;
X:=C;
```

Fig. 5. The generated code for solving tridiagonal system of equations

5 Related Work

As was mentioned in the Introduction, all algorithms derived here are well known and described in textbooks [2, 11]. And though the rationale for the algorithm for the Linear Recurrence problem in [2] is meaningfully the same as ours, it is described there in the style of common reasoning, and the Fortran code (which is exactly the same as in Fig. 2) is given just as a hand-written implementation. On the contrary, in our paper we demonstrate the possibility of mechanical derivation using a supercompilation mechanism. To the best of our knowledge, no existing compiler is able to generate parallel code based on recursive doubling, unless an associative operation for functions Scan or Reduce is given explicitly.

An interesting and rather complicated case of algorithm based on doubling is considered by B. Steinberg [7]. He explores the possibility of calculating by doubling method the recursion of the form:

$$Y_i = if\ P_i(Y_{i-1})\ then\ A_i\ else\ F_i(Y_{i-1}),$$

where A is a given numerical vector, and P, F are functions, that depend on index i as well as on the main argument. The author presents conditions on functions P, F and vector A, in which it is possible, and the respective theorem is proved. All is presented in the style of traditional mathematics, relying on the author's ingenuity. We hope that, by means of a supercompiler with a well-developed system of formula transformations, a similar result including the necessary restrictions on input functions can be obtained automatically.

6 Conclusion

The algorithms derived here are well known and described in the textbooks [2, 11], but their explanation and/or derivation is typically based on conjectures, intuition and special considerations. Here we suggest a general method to systematically derive them from simple definitions presented in the form of recursive

equations like $Y = F(Y)$. Then, a general recursive doubling method for calculating the fixed point is applied. This process is subject to execution under the control of a supercompiler. As a result, an efficient parallel code is generated in a language with parallel vector operations. Its execution takes $O(log(N))$ parallel time, where N is the size of the problem. We do not give a detailed analysis of their performance, as there are a lot publications about that. The purpose of the paper is to present the very method of obtaining algorithms. The method was announced by the author in 1988 [4]. Now, this paper reveals the details.

In the paper the supercompilation process is carried out manually, but following certain rules. The paper shows the techniques needed in a compiler for automated parallelization of a kind of loops. Of particular importance is the tool for formula transformations, which should allow to convert a formula to a certain pattern that is itself created in the same process.

References

1. Sergei M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications).* Nauka, Moscow, 1995. (In Russian).
2. R.V. Hockney and C.R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Ltd., Bristol, UK, 1981.
3. Andrei V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In *First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008*, pages 43–53. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008.
4. Arkady V. Klimov. Ob odnom metode polucheniya bystrodejstvujuschih parallelnyh algoritmov (on a method to obtain high performance parallel algorithms). In *Semioticheskie aspekty formalizacii intellektualnoj deyatelnosti. All-union school-seminar Borzhomi-88. Tezisy dokladov i soobschenij*, pages 53–56, Moscow, 1988. URL: http://agora.guru.ru/abrau2009/pdf/238_NSSI_2009_Abrau-2009.pdf.
5. Andrei P. Nemytykh, Victoria Pinchik, and Valentin Turchin. A self-applicable supercompiler. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 233–252. Springer-Verlag, 1996.
6. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
7. B.J. Steinberg. Rasparallelivanie rekurrentnyh tsiklov s uslovnymi operatorami (parallelizing recurrency loops with conditionals). *Avtomatika i telemekhanika*, 56(9):176–184, 1995.
8. Valentin F. Turchin. Program transformation by supercompilation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 1985.
9. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
10. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
11. V.V. Voevodin. *"Matematicheskie modeli i metody v parallelnyh protsessah" (Mathematical models and methods in parallel processes).* Nauka, Moscow, 1986.