Execution Analysis of ARPC Programs in the Environment of the Recursive Parallel Programming*

A. G. Sedov
Yaroslavl State University
Yaroslavl, Russia
E-mail: agsedov@gmail.com

***Abstract*—This article examines the analysis of an execution of a program written in the recursive parallel RPC language and the RPC extension for algebraic calculation facilities.**

An RPC program execution model named a Trace Graph has also been examined, as well as tools for its construction and rendering.

The Trace Graph formal definition is given in terms of Recursive Parallel Program schemes described in [1].

Next, the conception of the RPC language extension named ARPC is offered, main additional statements are described.

Furthermore, the author made a review of changes both in the execution model and the execution model construction algorithm which could be caused by RPC extension.

I. Introduction

The recursive parallel programming is a fairly easy and somewhat elementary method of distributed system development. This method was first introduced in the second half of the XX century. It spares a programmer from the designing of a distributed system architecture, needed, for example, when using the MPI. Instead of it, a programmer breaks the calculation task into some parts — recursive parallel procedures, which will be distributed among calculation facilities as necessary by Recursive Parallel environment.

To develop this method, the conception of the Recursive Parallel C language (shortly RPC) was offered[2]. The language fulfills the requirements stated below:

- it is a recursive parallel language designed for a virtual multiprocessor computing system with dynamic parallelization, hereinafter referred to as a recursive parallel machine (RPM).
- a parallel RPC program can be translated by a standard C compiler both into a parallel code for RPM and a sequential code for a common computer.
- it should be a tool for the program concurrency analysis (and program models). The language should also support specified RPM program performance analysis (and program models).

To implement the RPC language, the RPM Shell environment was created [3].

* This work is supported by the RFBR grant # 11-07-00549-a.

It should be noted that a programmer needs some tools to check the effectiveness of the calculation distribution among processors. These are construction and rendering tools for the program execution model and a simulation tool. A Trace Graph serves as an execution model for the RPC programs.

The Trace Graph is a directed graph formed out of the vertices associated with events occurred during the program execution and edges which are associated with a control flow relation. The Graph is used for attaining a bunch of goals related to analysis of the recursive parallel program behavior and its debugging: simulation, examining of potential concurrency, collection and rendering the statistical data. Trace Graph construction is a process of gathering information about the stucture of recursive parallel program execution and its performance during the run. Supporting the Trace Graph construction and its rendering is an essential part of this environment.

All functionality applied to work with the Trace Graph was implemented in the old version of RPM environment. It contained a library for graph construction representing the Graph in a binary format, a Trace Graph rendering module, and an RPM simulator. These solutions became outdated because they are not compatible with modern operation systems. However, the recursive parallel programming conception is still relevant, and the key element of the environment — a library for concurrent execution of the RPC programs — keeps on beeing upgraded.

This paper describes a new version of the Trace Graph construction tool and the Trace Graph renderer. The new Trace Graph constructor and renderer use the old Trace Graph format to make it compatible with the RPM Shell environment.

There are two main future directions: the Trace Graph structure revision and the RPC language extension. Their overview is given in Sections IV and V. A new Trace Graph formal definition is given in terms of Recursive Parallel Program schemes (RPPS) examined in [1]. Being defined in that way, the Trace Graph can be easily associated, vertex to vertex, with the RPPS. This will make all execution analysis applications much more visual.

The RPC language extension is aimed at realizing the ideas stated in [2], making the RPC more powerful. The new language will be named ARPC (Algebraic RPC) because it is intended for simplifying the development of applications for algebraic calculations. The RPPS extraction should be a feature of ARPC-to-RPC compiler. The requirements for the RPC extension are stated in Section V.

II. RPC PROGRAM STRUCTURE

In this section we are going to give some basic information about the RPC. A recursive parallel program may be thought of as a set of mutually recursive procedures.

The computational process which is performed in a computational system supporting the recursive parallel programming, is a hierarchical concurrent process. The process consists of a number of system functions (such as functions for memory access) and activations of procedures. We will define an activation as a parallel procedure invocation with specified input parameters. An activation which makes a recursive call is located higher in the hierarchy than a called activation. Any component that corresponds to procedure activation can be a hierarchical concurrent process. There are several types of parallel process components:

- concurrently invoked activations of user recursive parallel procedures;
- sequentially invoked activations of user recursive parallel procedures;
- operators for the concurrent shared memory access;
- *Wait()* operator. Being called from parent activation, the operator makes it wait for the completion of all child activations.

The activation of a procedure will be concurrent if it is called by using a special parallel process call operator (*PCall(ProcedureName)*). After invoking the parallel procedure call, computations will proceed without stopping until the synchronization point is reached. The procedure can be also invoked in a sequential mode, so its execution will start eventually in the same process, from which it was invoked.

The exit from the child procedure to the parent one returns the control to synchronization point. Computations in the parent procedure and the child ones can be concurrently executed.

A hierarchical model of the concurrent computations describes a process interaction as follows: each activation can have control relations only with the parent and child activations. The activation can only be completed when all its child activations are completed. The parent activation can pass data to the child activation, when it is generated, and it gets data from the child activation, when it is completed.

Every activation executed at any moment has a unique number.

One of RPC characteristics is a block of parameters — a structure containing local data for their transmission from a calling procedure to a called one, and vice versa. The name of a local variable of a given type (the name of a parameter block) is declared in the calling procedure. The access to the elements of the parameter block in a child procedure can be done only by the command *P_(elem)*, where *elem* is the name of a structure element.

Example 1: The structure of a recursive parallel program (RPP). Let us consider a recursive parallel program for the explicit solution of the heat conduction equation using the finite difference method. Its alghorithm consists in repeated computation of a new heat distribution array using the previous array and the initial condition arrays (the main function). The NextLayer procedure either divides a given array part or gets an array from the shared memory and executes computations.

```
struct NLParam
{
   int begin,end,min;
   int branching;
   float tau,h;
}
parallel(NextLayer, NLParam)
{
 if (P_(end) - P_(begin) > P_(min))
 {//If the segmentation
  //limit is not reached.
   for(int i=0;i<P_(branching);i++)
   {
    struct NLParam pbl;
    //...Copying the parameter
    // block to pbl.
    //Calculation of the begin
    //and end of the vector.
    pbl.begin = P_(begin)
    + i*(P_(end)-P_(begin))
    /P_(branching);
    pbl.end = P_(begin)
    + (i+1)*(P_(end)-P_(begin))
    /P_(branching);
    PCall(NextLayer, &pbl);
   }
  Wait(); //Synchronization
 }
 else
 {
  //Loading the array segment
  //from the shared memory.
  //Calculations and writing the results
  //to a temporary array in the shared memory.
 }
}
int main(...)
{
 //...Heat distribution array creation.
 //Writing the array to the shared memory.
 //...Temporary array creation.
 for(int i=0;i<IterCount;i++)
 {
   PCall(NextLayer, &pbl);
   Wait();
```

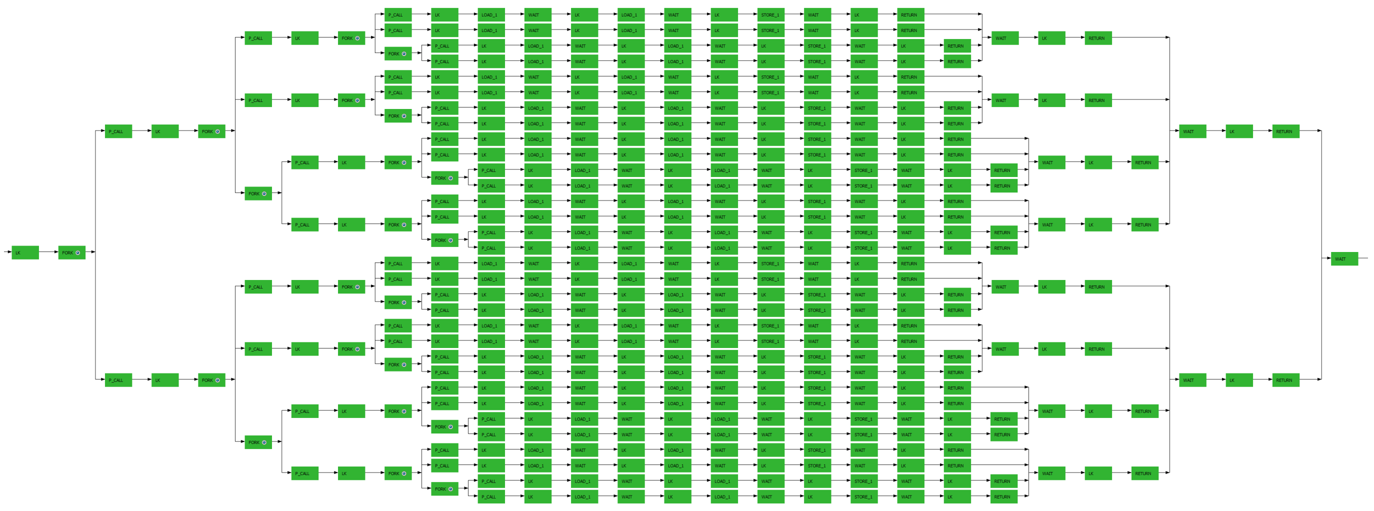
Figure 1. A fragment of the Trace Graph for the RPP from example 1 rendered in RPC Viewer (branching = 4).

```
    //Writing a new distribution
    //array to the file
    //Copying the temporary array
    //to the distribution array.
  }
}
```

III. Trace Graph construction and representation

The Trace Graph is a directed graph formed out of the vertices associated with the events occurred during the program execution, and the edges which are associated with transitions.

The following events are captured during RPC program execution, each event corresponding to a specific vertex type:

- forks — one or several calls to concurrent procedures; after call invocation, execution splits into a few parallel branches;
- synchronization operator calls — the parallel branches merge in a Wait vertex;
- calls to the shared and static memory management operations: allocation, deallocation, load, store;
- basic blocks — other sequential computations inside activations;

Specific data needed for modeling are saved as a file in addiction to vertices and relations between them. For example, a basic block vertex contains information about time taken to execute calculations. The vertices associated with the memory access operators contain statistics of the used memory volume.

A Trace Graph constructor is a library of functions for the sequential execution of an RPC program. It fully realizes the RPM functionality on a single computer. To compile an RPC program in a Trace Graph construction mode, the environment simply includes the library header file, instead of the header file for a standard mode library.

A Trace Graph is constructed by the GraphBuilder class. A GraphBuider instance is created before a call to the main program activation. As soon as any of the events from the list above occur, a new vertex is created and passed to the Graph builder.

The GraphBuilder handles an open-ended vertex list and a list of vertices that are ready to be written to a file. We will call a vertex open-ended, if new children may still appear. The open-ended vertex list contains one vertex from every process executed at the moment. We can consider it as a slice of the Trace Graph. After removing them from the open-ended vertex list, vertices are inserted into the ready-to-be-written list.

For each new vertex to be inserted into the open-ended list, its predecessor from the same activation is searched for. If a predecessor has not been found, the new vertex is recognized as a new activation beginning, and it is just inserted into the list. Else predecessor is removed from list.

To make a clear Trace Graph representation, the RPC-Viewer application was developed. It renders the Trace Graph as a hierarchical structure, allowing us to collapse or expand the chosen activations, and to view information mapped with the vertices. A format description for the Trace Graph file is passed to the RPC-Viewer in a special XML configuration file.

A Trace Graph rendering example can be seen in Figure 1.

IV. RPPS model

As we mentionted above, maintaining a Trace Graph construction in a format that is compatible with the old analysis tools was a goal of the first stage of work. However, the compatibility requirement appreciably limits the library facilities. For example, the information about associations between the Trace Graph structure and the code is not collected, so it is impossible to associate concrete recursive parallel procedures and their activations in the renderer.

Let us consider an RPPS (Recursive Parallel Program Scheme) model for describing the recursive parallel program structure and its execution in common terms.

A program scheme is a finite graph $G = (Q_G, q_0, \mapsto_G, L_G)$, where

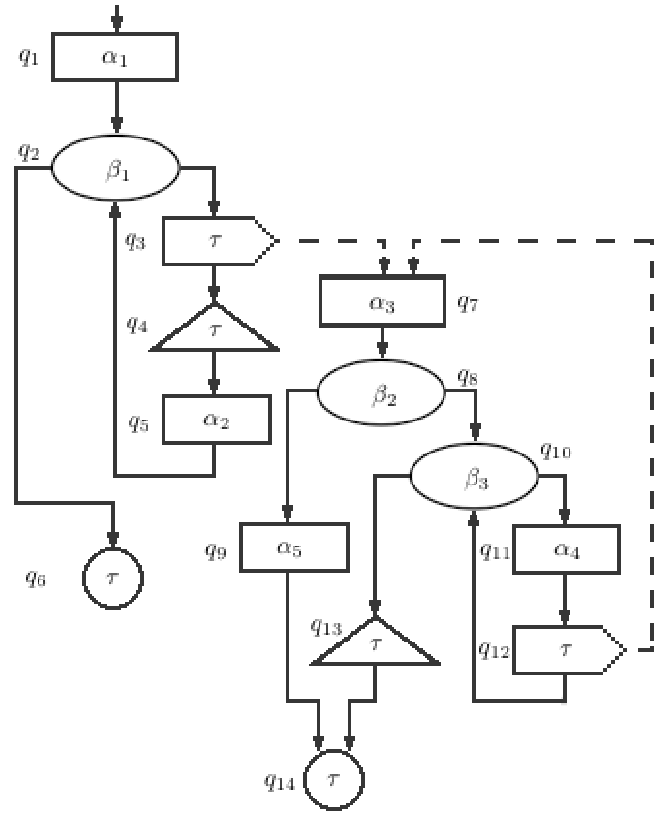

Figure 2. RPPS for example (1)

- $Q_G = \{q_0, q_1, ..., q_n\}$ — a finite set of vertices; every vertex being one of five types: action, selection, call, synchronization or ending;
- q_0 — the entry vertex (root);
- $\mapsto_G$: $Q_G \to Q_G^*$ — a function that maps every $q \in Q_G$ with its successors;
- $L_G : Q_G \to \Lambda_\tau \times Q_G^*$ — the function labelling graph vertices with symbols from Λ_τ. L_G also associates pcall vertices with the invoked vertex. Λ_τ is a basic set — alphabet containing abstract action names and silent actions τ.

A vertex q' is the successor of a q vertex, if $\mapsto_G$ assigns the q vertex to the vertex q'. The end vertices have no successors. Call vertices (pcall) are connected with two vertices: one vertex is the successor, another one is the called vertex.

The RPPS for the RPP of the heat equation solving can be seen in Figure 2. Here vertices q_1-q_6 correspond to the *main* function; q_7-q_{14} correspond to the *NextLayer* function. The solid lines designate $\mapsto_G$ transition, the dot lines designate L_G(Pcall).

A set of hierarchical states of $G \in RPPS_{\Lambda}\tau$ is a set $M_G = \{(q_1, \xi_1), ..., (q_n, \xi_n)\}$, where $q_1, ..., q_n$ are vertices from Q_G, and such that

- $\emptyset \in M_G$;
- $\xi_1, ..., \xi_n \in M_G$.

Example 2. The scheme from figure 2 generates $\xi = \{(q_4, \{(\tau, \{(q_9, \emptyset), (q_9, \emptyset), (q_9, \emptyset)\})\})\}$, a multiset corresponding to the situation in which the main function calls the Recursive Parallel procedure NextLayer, and NextLayer function, in turn, calls other three activations of NextLayer. These activations perform the calculations q_9.

Recursive-parallel program execution can be described as a sequence of hierarchical states $X = \xi_1 \longmapsto \xi_2 \longmapsto \xi_3 ... \xi_n$.

Let us give a formal definition of a new Trace Graph model.

We will define the Trace Graph of a recursive program as a finite graph $T = (Q_T, q_0, \mapsto_T)$, where

- $Q_T = \{q_0, q_1, ..., q_n\}$ — a finite set of vertices. There exists a function $f : Q_T \to Q_G$, such that it establishes a mapping of each vertex in Q_T to exactly one vertex in Q_G.
- q_0 — the entry vertex (root);
- $\delta : Q_T \to Q_T^*$ — a function which maps every $q \in Q_T$ to its succesors.
  If $f(q_i)$ is an action vertex, or a synchronization vertex, or a choice vertex, then $|\delta(q_i)| = 1$. If $f(q_i)$ is an end vertex, then $|\delta(q_i)| \leq 1$

The T structure differs from the existing Trace Graph structure first of all by a new vertex type, which was previously ignored during the Trace Graph construction — a choice vertex.

The RPPS for the RPC code will be constructed by means of the compiler. G_T will be shown in the renderer together with T.

V. A brief overview of requirements to the RPC extension. ARPC.

To make the RPC language convenient for algebraic calculations, new statements are planned to be embedded. We will call the extended language Algebraic RPC (ARPC). Let us consider some of these statements.

A. Stencil statement

A stencil is a special statement that specifies the way in which the work is divided into parts in a procedure. The aim of introducing the stencils is to separate the logic of the recursive parallel dividing of work from calculations.

The ARPC is a macrodefinition language, where every stencil is a parameterized macro. Here, the notion of a macro has a more general meaning than it is usually accepted in C. The ARPC language allows us:

- to declare macros with parameters, where another macro can be used as a parameter;
- to use the nesting of macros;
- to declare a macro in a code both before and after its call;

For instance, we can write the *VectorDivision* stencil, which specifies the manner of vector division for the RPP from Example 1, giving us an opportunity to specify calculations on a lower level of the recursion later.

```
$stencil VectorDivision(procedure,begin,
            end,min,branching,Computing)
 if (P_(end) - P_(begin) > P_(min))
 {//If the segmentation
  //limit is not reached.
  for(int i=0;i<P_(branching);i++)
  {
   struct NLParam pbl;
   pbl.begin = P_(begin)+i*(P_(end)
   -P_(begin))/P_(branching);
   pbl.end = P_(begin)+(i+1)*
   (P_(end)-P_(begin))/P_(branching);
   PCall(procedure, &pbl);
  }
  Wait(); //Synchronisation.
 }
 else
 {
  $Computing
 }
}
```

Here, the *$stencil* macro declares a stencil named *VectorDivision* with input parameters *procedure, begin* e.t.c.

The *$def ... $endd* command is used in the ARPC to declare a macro, and the *$ins* command is used to include a library stencil.

```
struct NLParam
{
   int begin,end,min;
   int branching;
   float tau,h;
}
parallel(NextLayer, NLParam)
{
  $ins stencil VectorDivision(NextLayer,
  begin,end,min,branching,COMPUTING)
  $def COMPUTING//Macros definition
  //...calculations in the
  //deepest recursion level
  $endd
}
int main(...)
//...
```

The *$stencil* macro implementation requires solving a problem of limitations which will constraint the code to be substituted in the stencil and a problem of protection of variables the values of which should be affected only by the stencil logic.

B. Generic procedures

One more type of a parameterized macro which is planned to be embedded in the ARPC is a generic procedure. The purpose of this construction is to declare a recursive procedure, the structure of which depends on specific input parameters. In practice, it means that the compiler generates several procedures for different levels of the recursion from the macro.

The generic procedure can be used to vary the shared memory type, a task type and a semantic structure. To vary a procedure structure in accordance with parameters, the ARPC command *$if⟨condition⟩ ... [$else ...] $endif* should be used.

The way in which the compiler will reduce the number of generated procedures could be of great interest for further research.

C. Other suggestions

The RPC compiler implementation will probably allow to hide bulky work with parameter blocks. In [2] there are some other constructions, such as procedure specialization and a sticking-together command. They are planned to be developed further.

VI. Conclusion

In this paper we considered some requirements for RPC extension through macrodefinitions for algebraic calculations. For programs written in this language the automatic construction method of the execution model is described.

Let us take a look at the further research directions.

- The ARPC statement designing.
- The ARPC to RPC compiler implementation.
- The RPPS constructor implementation.
- Further research work with the Trace Graph format and renderer.
- Investigation of benefits of the created tools for the RPC program verification.

New opportunities and problems arise from expanding the RPC and implementation of the ARPC to RPC compiler. The compiler would enrich the Trace Graph construction algorithm by adding special trace operators to the RPC code. New ARPC statements should also be represented in the Trace Graph.

For the development of a verifying compiler we suppose to use of such formalisms as a hierarchical system of interacting automata or Nested Petri Nets[4].

References

[1] O. B. Kushnarenko *Recursive Parallel Program semantic and methods of it analysis* // Ph. D Thesis Grenoble, France 1997

[2] Badin N.M, Brodsky G.M., Sokolov V.A. *A Recursive Parallel Programming Language and its application to algebraic computations* // Joint NCC and IIS Bulletin Computer Science Vol. 11, Ershov institute of informatics systems SB RAS, Novosibirsk, Russia, 1999, pages 1-14.

[3] Vasilchikov V.V. *Parallel programming facilities for computing systems with dynamic load balancing* // Yaroslavl, YSU, 2001.

[4] I. A. Lomazova. *Nested Petri nets: modeling and analysis of the distributed systems with object-oriented structure*//Moscow, Science world, 2004