# Towards Automatic Generation of Parallel Programs

**Nikola Ivković, Danijel Radošević, Ivan Magdalenić**
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
{nikola.ivkovic, danijel.radosevic, ivan.magdalenic}@foi.hr

**Abstract**. *Although there are various parallel programming models introduced and supported by different communication protocols, the building of parallel applications is still a kind of handwork. This paper deals with opportunity of usage Generative Programming techniques in parallelization of program functions written in a standard, non-parallel manner. Particularly, the usage of SCT generator model is discussed within generation of parallel programs based on the MPI communications protocol. An example of such generator was developed and tested.*

**Keywords.** generative programming, generator, parallel program, distributed system

## 1 Introduction

Various optimization problems, simulation and scientific calculations are often time-consuming and require computers with powerful and expensive hardware. Often, the computing problems can be decomposed into several subtasks that can run simultaneously on multiple computers. Parallel execution of programs enables their execution faster and such implementations are cheaper. But the parallel program execution brings new problems. All tasks cannot be parallelized. It is difficult to create and synchronize tasks and it is hard to discover bugs in programs. These are some of the obstacles in use of parallel programs.

This paper deals with opportunity of using Generative Programming techniques in parallelization of program functions written in a standard, non-parallel manner. There are a lot of generative programming techniques, and we decided to apply the already proven SCT model of source code generator [1]. By using generative programming techniques, we want to hide the complexity of constructing parallel programs that can run on multiple computers using the MPI protocol.

The basic idea of our solution is as follows. A programmer divides the problem into subtasks that can run in parallel. Our system generates needed source code that enables parallel execution. The programmer executes the program on multiple computers. The programmer does not need to have the knowledge of parallel programming techniques.

Issues addressed in our solution are creation and scheduling of tasks on multiple computers, their mutual communication and results collection. This is the first step toward building of such a system and the proposed solution is not complete. For example, a problem of possible various data type is not covered in this paper.

The paper is organized as follows: a background to the research is given in section 2. A model of parallelization is presented in section 3 which is followed by a case study described in section 4. The short description of SCT source code generator model is presented in section 5 followed by description of generator implementation. The conclusion is given in section 6.

## 2 Backgrounds to the Research

Because of all the hardness of parallel programming, for a long time researchers are trying to resolve a problem of automated parallelization. At this point, a complete automation of code parallelization at the compiler level seems to be unachievable goal. Currently there are only partial solutions. One is automatic parallelization of sequential code that can produce parallel code in some special cases. The rest of the code that could be parallelized is left sequential due to imperfection of current methods and tools. For a single multiprocessor machine this can be achieved purely on compiler level. For a system of processors that communicate over network some standardized protocol for exchanging data has to be supplemented, usually with additional libraries.

Another approach is to relay on human intervention, but also to provide substantial help for creating parallel programs. We use this second approach by using SCT source code generator model [1] for the generation of the needed source code. The SCT model is designed to work with code-fragment-sized components. The same approach is used in [2]. Our components are not necessarily strictly connected to program organizational units, like classes or methods. Consequently, our approach differs from the metaclass-based approaches, as described by Grigorenko et al. [3] and Tolvanen and Rossi [4].

Our implementation uses Message Passing Interface (MPI) that has become *de facto* standard for distributed system application. The MPI is a language-independent communications protocol that uses a message-passing paradigm to share data and states among a set of cooperative processes running on a distributed memory system [5]. MPI specification (Forum, MPI) defines a set of routines to support various parallel programming models such as point-to-point communication, collective communication, derived data types, and parallel I/O operations [5]. It is widely used in versatile distributed systems for solving challenging computational problems [6, 7,8, 9, 10].

## 3 A model of parallelization

To utilize a parallel architecture, a computational task has to be disassembled to a set of smaller subtasks that can be executed in parallel. There are different paradigms of parallel execution, but multiple instructions – multiple data streams (MIMD) architecture is the most general and most useable in practice.

In principal MIMD system can be implemented as shared memory or distributed memory system. A shared memory system consists from tightly coupled processor cores that have direct access to a pool of memory called shared memory. A distributed memory system is made from independent processing nodes that communicate using network.

In generally it is easier to write programs for a shared memory system then for a distributed memory system, since reading and writing memory doesn't need special mechanicals for communication. Nevertheless, an appropriate synchronization for writing memory is inevitable to ensure correct execution. Another advantage of a shared memory system is a fast access to the shared memory, while nodes in a distributed memory system have to exchange messages to communicate (which usually takes mach more time). Time to exchange a message is more critical in the distributed memory system called grid. In a grid nodes are usually geographically distant and have network connections of lower speed, then in the distributed memory system called cluster, where nodes are connected with a high speed LAN network. In contrast to shared memory systems that are less scalable and have to be specially manufactured, distributed memory systems are very scalable and are also cheaper (because they can use standard computers connected with a standard network technology).

In distributed memory systems there are numerous possibilities to connect nodes in different topologies. Also, nodes can be in the peer-to-peer relation or can have same hierarchy or master-slave relation.

Our model of parallelization is based on MPI technology that is designed for distributed memory systems (any topology), but can also be used in shared memory systems. It can also be combined with massive parallelism provided by GPU architecture [11]. Although simple, a master-slave model, it is suitable for very wide set of practical problems. In the master-slave relation one node – the master is coordinating node and all the others are slaves – the nodes that do parallel calculations and report their results to the master node.

In our model the essential module that is automatically generated is implemented with `ExecuteInParallel()` function. From a programmer's point of view our `ExecuteInParallel()` function is a black box that receives vector of input data and procedure to be used on these data in parallel fashion. As an output, `ExecuteInParallel()` gives a vector of results. Abstract representation of such module is given in the Figure 1.

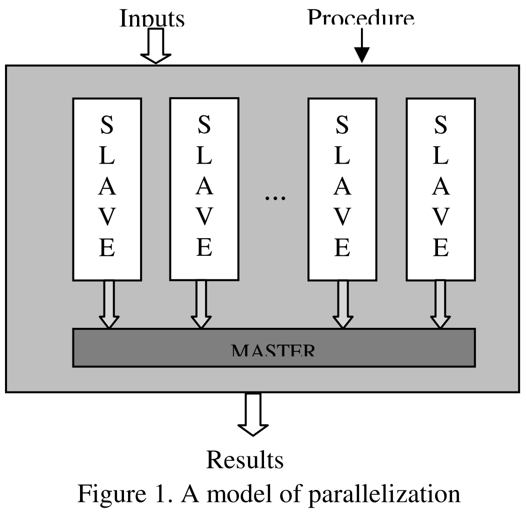

Figure 1. A model of parallelization

To gain maximal speedup it is the best to use all available nodes in the system. One node should be dedicated to a master process, and the rest *k* nodes should be assigned to *k* slave processes. If number of subtasks provided by a programmer are greater than a number of slave processes then some nodes will have to do more than one task. The tasks are scheduled in round robin fashion, so the slave process number *i* should execute tasks $i$, $i + k$, $i + 2k$, $i + 3k$, … After the master process receives the results from all subtasks, it provides a vector of results to a programmer as an output (Figure 1).

## 4 Case Study

An aim of this work is to examine possibility of using generative programming to hide the complexity of using a distributed system of processors to speed up time demanding computations. As the first steps toward this goal is to make a case study on a simple example. Calculating transcendental number π on arbitrary number of significant digits is possible by calculating partial sum of series, where partial sum can be conveniently defined by recurrent relation (1)

$$a_n = a_{n-1} + \frac{(-1)^{n+1}}{2n-1},\ with\ a_0 = 0 \qquad (1)$$

The term $a_n$ represents approximation of number π/4. For a higher value of $n$ the approximation of number π is more accurate. The recurrence relation (1) can easily be implemented in a programming language, for example as in the Code 1.

*Code 1 Simple sequential code in C/C++ language for calculating number π*

```
double PI(long N){
  double ai  = 0, sign = 1, i;
  for (i = 1 ; i <= N ; i++){
    ai += sign / (2 * i - 1);
    sign *= -1;
  }
  return ai * 4;
}
```

The main problem with this procedure is that, in order to achieve high precision, it is necessary to calculate this term in a large number of iterations. Therefore, an available time is limiting factor to achieve high accuracy. To maintain code simple we use `double` data type, but in general abstract data types that can store number with arbitrary precision can be used.

The first step in writing a program that can execute this calculation in parallel is to decompose an original task of calculating π to a number of smaller tasks. New tasks should be able to do parts of original task in a way that their results can easily be combined into final solution. This decomposition can be done in many different ways so in our opinion it is most suitable to leave this choice to a programmer.

A complete source code of our implementation is shown in code 2. A code on the grey background should be written directly by the programmer and all the rest is automatically inserted by program generator. In our case study we divided task in a way that smaller task calculates partial sum only for iterations from some subinterval.

As a language of implementation a C++ was chosen since C & C++ are regularly used for high performance computing. To maintain code more compact an object-oriented programming model was used in our implementation. A class of objects named `FractionOfPi` contains starting and ending iteration as well as resulting value for a fraction of π. Besides constructor, it implements member function `perform()` that calculates a fraction of π and stores it to the `partialSum`, and the member function `result()` that simply retrieves calculated value.

Programmer also needs to divide task of calculating π into smaller tasks, and to call function `ExecuteInParallel()`. All the work of calculating π is done in parallel, using nodes of a distributed system. At the end, the partial results are store in the vector `ps`. If necessary a programmer can combine provided results in the final result, in his case simply by summing the fraction of π. The rest of the code, with white background, deals with technical details to provide parallel execution and necessary communication and synchronization between nodes. A `MasterProces()` function receives the results from tasks executed on the other nodes and also provides synchronization by suspending the execution of master process until all scheduled tasks are finished. Another function, `SlaveProces()` execute one or more smaller tasks and, at the end of each task, send the results to the master process. Generator also inserts code that maintains data about distributed system; at the beginning initializes and at the end releases MPI resources.

## 5. Usage of generator

The SCT generator model defines the source code generator from three kinds of elements: Specification (S), Configuration (C) and Templates (T). All three model elements together make the SCT frame (Figure 1). The **Specification** contains features of generated application in a form of attribute-value pairs. The **Template** contains source code in a target programming language together with connections (replacing marks for insertion of variable code parts). The **Configuration** defines the connection rules between specification and template.

A starting SCT frame (Figure 2) contains the whole specification, the whole configuration, but only the base template from the set of all templates. Other SCT frames are produced dynamically, one SCT frame for each connection in upper template. SCT generator model is generator model based on dynamic frames, unlike some other frames-based generator models, like XVCL [12]. Generator works as a variation mechanism. It propagates the features specified in Specification on a set of program fragments, named as Templates. The connection rules are determined by Configuration.

*Code 2 A distributed implementation for calculating the number π in C++ with MPI*

```cpp
#include"mpi.h"
#include<vector>
#define PARTIALPIMESSAGE 531

struct MPI_related_data{
  int thisProcesID;
  int NumberOfProcesses;
} mpi_data;

class FractionOfPi{
  long long start, stop;
  double partialSum;
public:
  FractionOfPi(long long _first, long long _last):start(_first),stop(_last){}
  void perform(void);
  void result(double &sum) {sum=partialSum;}
};
                                                                                              // #class#
void FractionOfPi::perform(void){
  double ai  = 0, sign = (start%2==0)? -1 : 1;

  for (long long i = start ; i <= stop ; i++){
    ai += sign / (2 * i - 1);
    sign *= -1;
  }
  partialSum=ai * 4;
}
void MasterProces(std::vector<double> & output){
  MPI_Status stat;
  int NumberOfSlaves = mpi_data.NumberOfProcesses - 1;
  for(int procNum = 1; procNum < mpi_data.NumberOfProcesses; procNum++)
    for(size_t i = procNum; i <= output.size(); i += NumberOfSlaves)
      MPI_Recv(&output.at(i-1), 1, MPI_DOUBLE, procNum, PARTIALPIMESSAGE, MPI_COMM_WORLD, &stat);
}
void SlaveProces(std::vector<FractionOfPi> & input){
  double r;
  int NumberOfSlaves=mpi_data.NumberOfProcesses-1;
  for(size_t i=mpi_data.thisProcesID; I <= input.size(); i += NumberOfSlaves){
    input.at(i-1).perform();
    input.at(i-1).result(r);
    MPI_Ssend(&r, 1, MPI_DOUBLE, 0, PARTIALPIMESSAGE, MPI_COMM_WORLD);
  }
}
void ExecuteInParallel(std::vector<FractionOfPi> & input, std::vector<double> & output){
  if(mpi_data.thisProcesID==0){
    output.resize(input.size());
    MasterProces(output);
  }
  else  SlaveProces(input);
}
#include<iostream>
using namespace std;
int main(int argc, char **argv){
   atexit((void (*)())MPI_Finalize);
  if(MPI_Init(&argc,&argv)!=MPI_SUCCESS) exit(1);
  if(MPI_Comm_size(MPI_COMM_WORLD,&mpi_data.NumberOfProcesses)!=MPI_SUCCESS) exit(2);
  if(MPI_Comm_rank(MPI_COMM_WORLD,&mpi_data.thisProcesID)!=MPI_SUCCESS) exit(3);
  vector<FractionOfPi> pp;
  vector<double> results;
  pp.push_back(FractionOfPi(1,100000000));        // #interval# -> 1,100000000
  pp.push_back(FractionOfPi(100000001,200000000));// #interval# -> 100000001,200000000
  pp.push_back(FractionOfPi(200000001,300000000));// #interval# -> 200000001,300000000
  pp.push_back(FractionOfPi(300000001,400000000));// #interval# -> 300000001,400000000
  ExecuteInParallel(pp, results);
  if(mpi_data.thisProcesID==0){ // this block of code is executed only by master process
   double totalsum=0.0;
   for(size_t i=0; I < results.size(); i++) totalsum += results.at(i);
   cout<<"PI = "<<totalsum<<endl;
}// end of block executed only by master process
return 0;
}
```

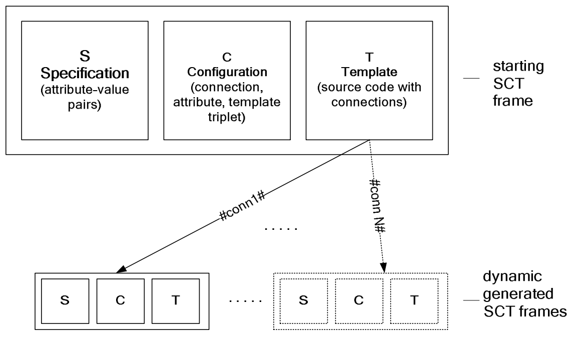

Figure 2. SCT Frame

The Specification consists from *attribute-values pairs* and for the given example could look as follows:

```
OUTPUT:out1

out1:output/parallel.cpp

class:FractionOfPi  // class name
tasks:4             // number of tasks
//intervals:
interval:1,100000000
interval:100000001,200000000
interval:200000001,300000000
interval:300000001,400000000
```

There is one type of output specified, *out1*, and one output file to be generated, *output/parallel.cpp* . The output type is connected to appropriate top-level template, which is defined in Configuration as *#1#*.
The Configuration connects attributes to links (replacing marks) that are used in Templates:

```
//main template
#1#,,main.template

//simple connections:
#class_name#,class
#interval#,interval

// using subordinated template:
#class#,class,class.template
    . . .
```

Templates and their connections could be easily represent by Configuration diagram [1], as shown in Figure 3. The *Main template* contains the basic structure of the code to be generated, including the majority of the parallel resources. Triangles represent replacing marks i.e. parts of the code to be defined during the generation process. Rounded rectangles represent values from Specification.
This architecture of generator offers a lot of flexibility into design of parallel systems. All three model elements (Specification, Configuration and Templates) could be easily extended enabling adaptation of generator system to the needs of parallel processing.

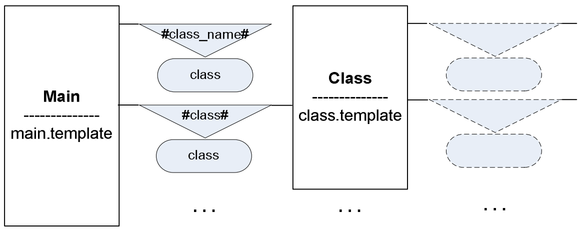

Figure 3. Configuration diagram of the example

## 6 Conclusion

Our research shows that using generative programming can hide technical peculiarities and complexity from programmer and yet provide it with ability to use distributed system to speed up computationally demanding calculations. By employing distributed system this method has very high scalability.
Although, this model is developed on a simple case study it seems general enough to handle a whole class of practical problems. In future work we are planning to extend our model so it can handle more general cases and possibly to offer some alternative modes and network topologies through our program generator.

## References

[1] Radošević D., Magdalenić I., “Source Code Generator Based on Dynamic Frames”, Journal of Information and Organizational Sciences, vol. 35, no. 2, pp. 73–91, 2011.

[2] Griss M. L. Product line architectures. *In G. T. Heineman, & W. T. Councill (Eds.), Component-based software engineering: Putting the pieces together* (pp. 405-420). Boston: Addison-Wesley.

[3] Grigorenko P., Saabas A., Tyugu E. Visual Tool for Generative Programming. *Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* ACM Publ., pp. 249–252, 2005.

[4] Tolvanen J.P., Rossi M. Metaedit+: Defining and using domain-specific modeling languages and code generators. *In OOPSLA 2003 demonstration*, 2003.

[5] Ekanayake J., Qiu X., Gunarathne T., Beason S., Fox G. High Performance Parallel Computing with Cloud and Cloud Technologies. *Cloud*

*Computing and Software Services: Theory and Techniques*, CRC Press (Taylor and Francis), pp. 1-39, 2010.

[6] M. Waintraub, R. Schirru, C.M.N.A. Pereira: Multiprocessor modeling of parallel Particle Swarm Optimization applied to nuclear engineering problems. *Progress in Nuclear Energy*, 51(6–7): 680-688, 2009.[8] M. Pedemonte, S. Nesmachnow, H. Cancela: A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8): 5181-5197, 2011.

[7] O. Nesterov: A simple parallelization technique with MPI for ocean circulation models. *Journal of Parallel and Distributed Computing*, 70(1): 35-44, 2010.

[8] M. Chau, D. El Baz, R. Guivarch, P. Spiteri: MPI implementation of parallel subdomain methods for linear and nonlinear convection–diffusion problems. *Journal of Parallel and Distributed Computing*, 67(5): 581-591, 2007.

[9] J. Zhao: IB: A Monte Carlo simulation tool for neutron scattering instrument design under PVM and MPI. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 659(1): 434-441, 2011.

[10] D. Komatitsch, G. Erlebacher, D. Göddeke, D. Michéa: High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20): 7692-7714, 2010.

[11] C. S. Ierotheou, S. P. Johnson, M. Cross, P. F. Leggett: Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes. *Parallel Computing*, 22(2): 163-195, 1996.

[12] Zhang H., Jarzabek S., "XVCL: a mechanism for handling variants in software product lines", *Science of Computer Programming*, 53(3): 381-407, 2004.