Seeking Appropriate Tools for Programming the Parallel Computations

Andrzej Zadrożny

Institute of Computer Science, Polish Academy of Sciences,
ul. Jana Kazimierza 5, 01-248 Warsaw, Poland

Abstract. Concurrent and distributed computations are in use since many years. Now, the time has come for parallel computations. For nowadays computers are equipped with several processors(or cores). We start with the observation that no programming language comes with tools appropriate for programming parallel computations. There are several languages and/or libraries that allow to program concurrent and distributed computations. Base on our experiment we propose a new parallel model. The programmers will appreciate a possibility to manage the processors (cores) and to communicate between objects of processes in a truly object way.
Our experiments concentrates around LOGLAN programming language. Because LOGLAN lacks parallel programming mechanism it can be a good platform for experimental implementation. The proposed model is not tied to LOGLAN platform and should be implemented for other if proven useful.

1 Introduction

Concurrent computations are getting more popular like never before. Both versions of non-sequential computations: distributed and parallel, are applied more and more frequently. This is stimulated by the growth of capabilities of single PC (it's computing power, memory capacity and core availability in single unit) and availability of clusters. These features encourage new development in this areas.

Concurrent computations first were realised as processing unit switching between active tasks. CPU time was managed and divided by the operating system scheduler for all working tasks. This schema is still used by operating system.

Distributed computations emerged due to popularization of computer networks. Lack of shared memory

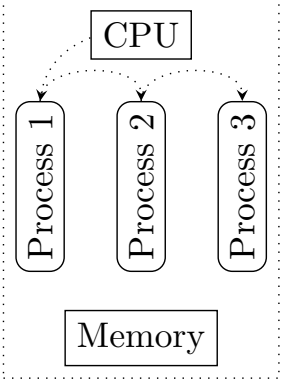

Fig. 1. Concurrent model

among computing units is the essential difference between distributed and concurrent programs. Fundamentals of this concept were defined by C.A.R. Hoare[1][2]. Who also distinguished the case of disjoint parallelism. Nowadays this schema of concurrent computations is called share-nothing architecture and probably is the most popular.

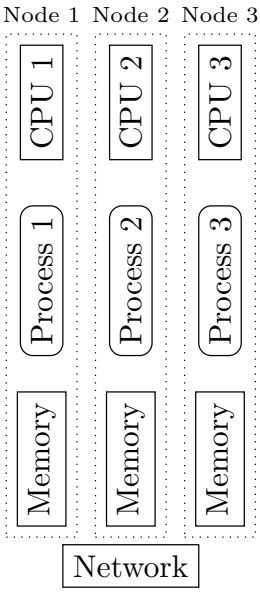

Fig. 2. Distributed model

It's popularity is related to huge impact of Apache Hadoop on industry. A whole ecosystem has grown around Hadoop and Hadoop Distributed File System (HDFS). It now consist of many libraries, languages and frameworks. The concept of programming in Hadoop, called MapReduce, is based on methodology proposed by J. Dean and S. Ghemawat in the paper[3]. It assumes partitioning of main computation task into small sub-tasks. Computation of each sub-task takes place in a distributed shared-nothing environment. After all tasks are finished results are combined into single final result. In between each computation phase of algorithm partial results are stored on a shared distributed file system. MapReduce is broadly used thanks to it's characteristics. This schema is naturally simple. Algorithm is somewhat fault tolerant. If some sub-tasks fail, they can be reassigned and computed by non-failed nodes.

Usage of shared-nothing architecture resulted in popularisation of MapReduce methodology. At the same time it leaves a lot of place for other optimisations. An a example of such optimisation is Apache Spark[4] project. Apache Spark extends MapReduce programming schema with concept of in-memory caching intermediate sub-tasks results.

MapReduce methodology was a simplification of earlier available techniques. One of the first widely used standard was Message Passing Interface (MPI). It is a standardized and portable message-passing designed pattern. In which processes communicate with others by means of passing messages.

This approach was adapted 30 years ago by Loglan development team under the name Alien Call protocol. It was suited to object-oriented language. It was extended in LOGLAN'82[5][6]. A programmer creates a method in a process which becomes a micro service. This method can be called by other processes in Virtual Loglan Processor. Method call will be handled if a serving process is active and accepts the call. Alien Call protocol turned out to be useful in programming of distributed computations.

Similar message passing pattern and micro services where used in Java RMI. Server program implements some interfaces and publishes it with a name. A java client program needs to know which interface is implemented. After network connection initialisation a client program binds a remote object by its

name to local variable. Client program can call remote method from server program. The biggest difference between LOGLAN's Alien Call and Java RMI is the initialisation method. Client and server programs are initialised separately on different machines. In LOGLAN all processes are initialised from single virtual environment from main program.

From concurrent processing emerges one more model: parallel processing. It is available on systems where are multiple processing units and they share memory. The tasks are not switched while processing by single processor but are computed truly simultaneously by multiple processors. However, from programming point of view implementing concurrent and parallel computations are all the same.

Lets think about Java, or any other language equipped with threads. A programmer can start two threads, but he never knows if those threads are to be executed in parallel or concurrent manner.

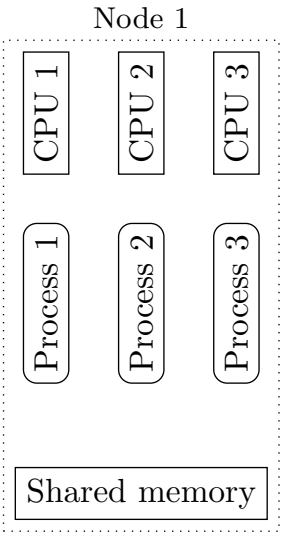

Fig. 3. Parallel model

This approach can be efficient when writing programs that should work on nodes with single cores and multiple cores as well. But our intuition is that parallel processing has a mix of properties of both concurrent and distributed: shared memory and truly simultaneous. We should treat parallel computations as completely different model from concurrent computations. A new model to be found. We could look at it as improved distributed model, rather than faster concurrent model. In this concept parallel processing still lacks proper managing and programming tools to really leverage it's potential.

Regardless of model being used programmer needs to take care of problems related with non-sequential processing. These problems are known: deadlock, starvation, livelock, race condition and others. Some models are more prone to particular problems than others.

This paper is a report of current state of research and it brings perspectives for future work.

In next chapter we discus use case of LOGLAN language and Alien Call protocol as simulation of parallel computation on matrix multiplication. Our deliberations are based on Cannon's Algorithm[7] and SUMMA Algorithm[8]. In subsequent chapter we discus changes done to Loglan language runtime environment (Virtual Loglan Processor VLP). In the last chapter we summarise profits and digest possible research and future development.

2 Case study

The base for our case study is distributed matrix multiplication.

$$C = A * B$$

Cannon's Algorithm divides input and output matrixes into p^2 sub-matrixes (figure 4). Number of sub-matrixes is equal the number of processors. Cannon's algorithm requires that the dimmentions of original matrixes are multiplicity of sub-matrixes dimmentsions. Each processors' $P_{i,j}$ task is to compute a sub-matrix of matrix C of dimmentions N x N.

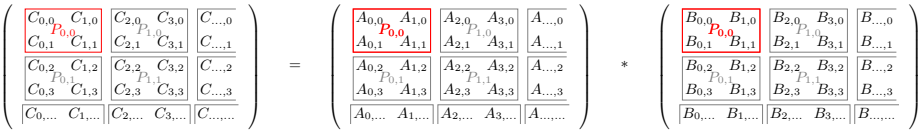

Fig. 4. The schema of data splitting in Cannon's Algorithm.

Before first iteration matrixes are split and transferred to processors. At each iteration each processors multiplies sub-matrixes and sums up to the output sub-matrix.

Between each iteration sub-matrixes are shifted to the next processor (figure 5).

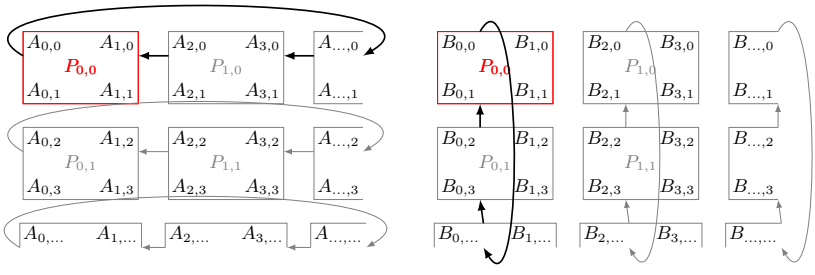

Fig. 5. The schema of data shifting in Cannon's Algorithm.

Our algorithm differs from Cannon's algorithm in following manner. Each processor $P_{i,j}$ gathers whole rows from $N * i$ to $N * (i + 1)$ of matrix A and whole columns from $N * j$ to $N * (j + 1)$ of matrx B (figure 6). Authors of SUMMA[8] also took this approach. Cannon's algorithm gathers only parts of rows and columns of A and B for each round. After each round it rotates parts of input data between processors. Our approach simplifies the algorithm, takes up more memory, but loses less time on data reshuffle.

Authors of SUMMA algorithm describe[8] other optimisations used to gain performance. We omit those optimisations as we want to focus on communication aspects of algorithm.

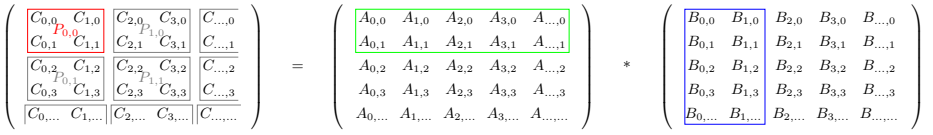

Fig. 6. The schema of sub-matrix splitting in our algorithm and in SUMMA algorithm.

This algorithm has been chosen because of it's characteristics. It is an example of disjoint parallelism defined by C. A. R. Hoare. All processors use overlapping parts of matrices A and B. But each processor writes only its own part of matrix C. No other process depends on parts of matrix C that is written by other processor.

In order to gain some experience we developed a version of algorithm working on a cluster. This cluster is called Virtual Loglan Processor. Nodes are connected by network. This leads to the following assumption on data:

- the data are distributed over network (e.g. because they were registers in this distributed way), or
- there is not enough space to store all three matrices A, B, C on one computer,

Below we present our implementation:

Listing 1. matrix multiplication in LOGLAN with Alien Call protocol

```
1  program MatrixMultiplication;
2  unit SliceMatrix: class(x1,x2,y1,y2,size:integer);
3    var T: arrayof arrayof real, i,k: integer;
4      unit mult: function(x: SliceMatrix): SliceMatrix;
5        var i,j,k:integer, s: real;
6    begin
7      result:= new SliceMatrix(x.x1, x.x2, y1,y2, size);
8      for j := y1 to y2 do
9        for i := x.x1 to x.x2 do
10         s:=0;
11         for k := 1 to size do
12           s:= s + T(j,k) * x.T(k,i);
13         od;
14         result.T(j,i) := s;
15       od;
16     od;
17   end mult;
18
19   begin
20   writeln("create matrix ",x1,x2,y1,y2);
21     array T dim(y1: y2);
22     for i:=y1 to y2 do
23       array T(i) dim (x1:x2) od;
24   end SliceMatrix;
```

```
25
26  unit Table: process(node, size:integer);
27    var Tab: SliceMatrix ,k:integer;
28    unit get: function(i,j: integer):real;
29    begin result:=Tab.T(j,i); end get;
30    unit put: procedure(i,j:integer, wartosc: real );
31    begin Tab.T(j,i) := wartosc; end put;
32    unit fin: procedure; begin continue := false end fin;
33  var continue: boolean;
34  begin
35    Tab:=new SliceMatrix(1,size,1,size,size);
36    continue := true;
37     k:=0;
38    return;
39    while continue do
40      accept put, get, fin;
41    od;
42  end Table;
43
44  unit Multiplier: process(node,x, x2, y, y2, size: integer, A,B,C: Table)
        ;
45          var sc, sa, sb:SliceMatrix, i,j: integer, tmp:real;
46  begin
47    return;
48    sa := new SliceMatrix(1,size,y,y2,size);
49    sb := new SliceMatrix(x,x2,1,size,size);
50    for i:=1 to size do
51      for j:=y to y2 do
52        sa.T(j,i):= A.get(j,i);
53      od;
54      for j:=x to x2 do
55        sb.T(i,j):= B.get(i,j);
56      od;
57    od;
58    sc:=sa.mult(sb);
59    for j:=y to y2 do
60      for i:=x to x2 do
61        call C.put(i,j,sc.T(j,i));
62      od;
63    od;
64  end Multiplier;
65
66  var A,B,C:  Table, D: Multiplier, i,j,m,n,n2,node:integer;
67  begin
68    n := 2;
69    m := 1000;
70    node :=1;
71    n2 := m div n;
72
73    A := new Table(node,m);
74    resume(A);
75
76    B := new Table(node,m);
77    resume(B);
78
79    C := new Table(node,m);
80    resume(C);
81    for i := 1 to n do
82      for j := 1 to n do
83        D := new Multiplier(
84           (i-1)*n+j+1,(i-1)*n2+1,i*n2,
85           (j-1)*n2+1, j*n2, m, A,B,C
86        );
87        resume(D);
88      od
89    od;
90  end program MatrixMultiplication
```

For comparison we provide a quotation of MPI implementation of SUMMA algorithm from the paper[8].

Listing 2. matrix multiplication in C with MPI (SUMMA algorithm)

```
#include "mpi.h"
/* macro for column major indexing */
#define A( i,j ) (a[ j*lda + i ])
#define B( i,j ) (b[ j*ldb + i ])
#define C( i,j ) (c[ j*ldc + i ])
#define min( x, y ) ( (x) < (y) ? (x) : (y) )

int i_one=1;
double d_one=1.0,
d_zero=0.0;
void pdgemm( m, n, k, nb, alpha, a, lda, b, ldb, beta, c, ldc, m_a, n_a,
      m_b, n_b, m_c, n_c, comm_row, comm_col, work1, work2 )
int m, n, k,
nb,
m_a[], n_a[],
m_b[], n_b[],
m_c[], n_c[],
lda, ldb, ldc;
double *a, *b, *c,
alpha, beta,
*work1, *work2;
MPI_Comm comm_row, comm_col;
{
  int myrow, mycol,
  nprow, npcol,
  i, j, kk, iwrk,
  icurrow, icurcol,
  ii, jj;
  double *temp;
  double *p;

  MPI_Comm_rank( comm_row, &mycol );
  MPI_Comm_rank( comm_col, &myrow );

  for ( j=0; j<n_c[ mycol ]; j++ )
    for ( i=0; i<m_c[ myrow ]; i++ )
      C( i,j ) = beta * C( i,j );
  icurrow = 0; icurcol = 0;
  ii = jj = 0;

  temp = (double *) malloc( m_c[myrow]*nb*sizeof(double) );
  for ( kk=0; kk<k; kk+=iwrk) {
    iwrk = min( nb, m_b[ icurrow ]-ii );
    iwrk = min( iwrk, n_a[ icurcol ]-jj );

    if ( mycol == icurcol ) {
      dlacpy_( "General", &m_a[ myrow ], &iwrk, &A( 0, jj ), &lda, work1
          , &m_a[ myrow ] );
    }

    if ( myrow == icurrow ) {
      dlacpy_( "General", &iwrk, &n_b[ mycol ], &B( ii, 0 ), &ldb, work2
          , &iwrk );
    }

    RING_Bcast( work1 , m_a[ myrow ]*iwrk, MPI_DOUBLE, icurcol, comm_row
        );
    RING_Bcast( work2 , n_b[ mycol ]*iwrk, MPI_DOUBLE, icurrow, comm_col
        );

    dgemm_( "No_transpose", "No_transpose", &m_c[ myrow ], &n_c[ mycol
        ], &iwrk, &alpha, work1, &m_b[ myrow ], work2, &iwrk, &d_one, c,
        &ldc );
```

```
57
58        ii += iwrk; jj += iwrk;
59        if ( jj>=n_a[ icurcol ] ) {
60          icurcol++; jj = 0;
61        };
62        if ( ii>=m_b[ icurrow ] ) {
63          icurrow++; ii = 0;
64        };
65      }
66      free( temp );
67  }
68  RING_Bcast( double *buf, int count, MPI_Datatype type, int root,
        MPI_Comm comm ) {
69      int me, np;
70      MPI_Status status;
71      MPI_Comm_rank( comm, me ); MPI_Comm_size( comm, np );
72      if ( me != root)
73      MPI_Recv( buf, count, type, (me-1+np)%np, MPI_ANY_TAG, comm );
74      if ( ( me+1 )%np != root )
75      MPI_Send( buf, count, type, (me+1)%np, 0, comm );
76  }
```

Both examples are shortened for the purpose of presentation. Complete source code of our example is in git repository[9]. One should notice that our implementation uses single communication routine (lines 52, 55, 61). It calls remote method via Alien Call which is part of the language.

On the other hand SUMMA algorithm calls multiple MPI functions (lines 31, 32, 71, 73, 75) and uses as many MPI specific data types (lines 21, 70, 53, 54). As we can see inclusion in language remote procedure call can be beneficial.

3 New environment

Our algorithm has been implemented in LOGLAN'82 language. It was compiled on compiler from 1993 with later minor changes. Algorithm has been tested on experimental runtime environment Virtual Loglan Processor (VLP) implemented as a part of the author's research[10]. This new virtual machine uses wxWidgets and Boost libraries and is already prepared for running on Windows and Linux platforms. Change in virtual machine also entailed some changes in Loglan interpreter. Most substantial were interprocess communication changes.

One of our objectives was to check if we can run algorithm in parallel without extensive changes in the virtual machine and without changes in the compiler. We wanted only to check how big changes are required to create truly parallel environment. On the other hand we have made great effort in preparing new virtual machine and adapting whole package to nowadays operating systems.

Our second objective was to gather experience and experiment with model.

As a result of minor changes in virtual machine our algorithm was running as 4 processes on single machine on 4 CPU cores. This way we have simulated parallel environment. In essence, the computations were performed as in distributed environment.

Alien Call protocol makes some assumption that makes it difficult to parallelize computations. Each process has access only to fields declared inside of it or passed to it as arguments. If two processes run on the same interpreter

they are running concurrently. Memory sharing between processes on separate interpreters is impossible at all.

All resources are private. This assumption comply with the concept of shared nothing architecture.

In Alien call protocol an active process object executes its own instructions or method ordered remotely by other process. There is no possibility to execute two methods at the same time. This clearly shows that we are looking for other programming model. We would like to have a system which can execute more than one non conflicting method at a time. For this purpose we need to adopt a schema to manage or designate methods as conflicting or not.

4 Effects and perspectives

The main profit from the research is the increase of scalability of the Virtual Loglan Processor. Multiple instances can be run on single machine and effectively utilize ale available CPU cores.

We have gained some experience from the examples discussed above. We sketch a vision of the new model of parallel computations.

4.1 New model requirements

- programmer will have possibility to assign processes to particular CPU core,
- processes inside same interpreter can directly share memory objects,
- programmer will be able to manage or designate conflicting methods,
- parallel processes will be run truly parallel.

4.2 Development's roadmap

Virtual Machine rewrite DONE
Multi-platform runtime system adapted for modern operating systems written using wxWidgets and Boost libraries.

Alien Call - array variables IN-PLANNING
Allowing passing of array variables between processes. Will require compiler update or rewrite.

Parallel process IN-PLANNING
Creation of completely new type of programming language component. Will require compiler update or rewrite.

Parallel computation model IN-PROGRESS
Design, testing and verification of the model for parallel computations.

Below, we present a concept of code that we think should be runnable in the proposed parallel processing model.

Listing 3. parallel matrix multiplication concept

```
1  program ParallelMatrixMultiplication;
2
3  unit Table: class(node, size:integer);
4    var T: arrayof arrayof real, k: integer;
5  begin
6  array T dim(y1: y2);
7      for i:=y1 to y2 do
8        array T(i) dim (x1:x2) od;
9  end Table;
10
11 unit Multiplier: parallelprocess(node,x1, x2, y1, y2, size: integer, A,B
       ,C: Table);
12           var i,j: integer, s:real;
13 begin
14   return;
15   for j := y1 to y2 do
16       for i := x.x1 to x.x2 do
17         s:=0;
18         for k := 1 to size do
19           s:= s + A.T(j,k) * B.T(k,i);
20         od;
21         C.Tab(j,i):= s;
22       od;
23     od;
24 end Multiplier;
25
26 var A,B,C:  Table, D: Multiplier, i,j,m,n,n2,node:integer;
27 begin
28   n := 2;
29   m := 1000;
30   node :=1;
31   n2 := m div n;
32
33   A := new Table(node,m);
34   B := new Table(node,m);
35   C := new Table(node,m);
36   for i := 1 to n do
37     for j := 1 to n do
38       D := new Multiplier(
39         node,(i-1)*n2+1,i*n2,
40         (j-1)*n2+1, j*n2, m, A,B,C
41       );
42       resume(D, (i-1)*n+j+1);
43     od
44   od;
45 end program ParallelMatrixMultiplication
```

This code is much shorter than our simulation example. Thanks to memory sharing between parallel processes all data synchronisation code is unnecessary. Also code required by sub-matrix allocation was unnecessary.

Process allocation on particular core is done by resume command with additional parameter.

Acknowledgements

I thank my advisor professor Andrzej Salwicki for his support and all the comments.

The paper is co-founded by the European Union from resources of the European Social Fund. Project PO KL "Information technologies: Research and their interdisciplinary applications", Agreement UDA-POKL.04.01.01-00-051/10-00.

References

1. Hoare, C.A.R.: Towards a theory of parallel programming. In Hoare, C.A.R., Perrott, R.H., eds.: Operating Systemss Techniques. Academic Press, New York (1972)
2. Hoare, C.A.R.: Monitors: an operating system structuring concept. Communications of the ACM. **17**(10) (October 1974) 549–557
3. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. OSDI'04, Berkeley, CA, USA, USENIX Association (2004) 10–10
4. Matei Zaharia, e.a.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, NSDI 2012 (2012)
5. Bartol, W.e.a.: Report on the Loglan'82 programming language. PWN Polish Scientific Publisher (1984)
6. Ciesielski, B.: An implementation of distributed processes in Loglan'82. Master's thesis, Institute of Informatics, Warsawa University (1988) `http://lem12.uksw.edu.pl/images/2/29/ArtBolkaCiesielskiego1988.pdf`.
7. Cannon, L.E.: A cellular computer to implement the Kalman Filter Algorithm. PhD thesis, Montana State University (1969)
8. van de Geijn, R.A., Watts, J.: Summa: scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience **9**(4) (April 1997) 255–274
9. Loglan: Distributed matrix multiplication in Loglan repository on github. `https://github.com/lemlang/dmml/` (2015) [Online; accessed 30-September-2015].
10. Loglan: Loglan and VLP online repository on sourceforge. `http://sourceforge.net/projects/loglan82/` (2015) [Online; accessed 30-September-2015].