Computer Science Department
University of Valladolid
Valladolid - Spain

Tools to schedule and simulate parallel computations expressed as directed graphs

Arturo González-Escribano, Arjan J.C. van Gemund[2], and Valentín Cardeñoso-Payo[1]

[1] Dept. de Informática, Universidad de Valladolid.
E.T.I.T. Campus Miguel Delibes, 47011 - Valladolid, Spain
Phone: +34 983 423270, eMail:arturo@infor.uva.es

[2] Faculty of Information Technology and Systems (ITS)
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786168, eMail:a.j.c.vangemund@tudelft.nl

Abstract Some parallel computations, i.e. sparse-matrix factorizations, may be expressed as directed graphs. Once the graphs are obtained, they may be scheduled or mapped to a parallel machine in different ways. This document describes tools developed to experiment with these kind of computations, creating trivial schedulings, and generating MPI programs which simulate their behavior when executed on real machines. The tools have been used to experiment with some sparse-matrix factorization graphs on a Beowulf cluster.

Technical Report No. IT-DI-2007-0001

1 Introduction

Many parallel computations may be expressed as directed graphs. Nodes represent computation, and they have a number of floating point operations associated. Edges represent data dependencies or execution order between the computations represented by the nodes. These directed graphs are a compact and convenient form to represent the computations. Automatic graph transformation tools may be used to work with them, testing different implementation options.

To implement the computation it is first needed to schedule the nodes in the number of real available processors. Data dependencies propagated across different processors should derive in communications when a real implementation is built.

In this document we describe tools which, from an input graph, may create different schedulings for different number of real processors. The tools generate all the information needed for an MPI program simulator. The simulator executes in each processor the associated floating point operations for each node, and issues the needed communications.

The graph transformation tool is integrated with other previous graph tools used to transform the synchronization structure of the application. See [2,3] for details.

The tools have been used to experiment with some sparse-matrix factorization graphs obtained from civil-engineering examples with DIANA [1,4]. The simulations has been tested in a 20 nodes Beowulf cluster.

In sections 3 to 5 we present a description of the tools. In section 6 we describe the experiments and results obtained for the selected input graphs.

2 Parallel computations expressed as graphs

In this section we describe the semantics of the graphs used to represent the parallel computations.

2.1 Graphs

A *task graph* is a directed graph $G = (V, E)$, composed by a set of nodes V and a set of edges $E \subset V \times V$. Each node v_i represent a sequential computation. It has an associated number of floating point operations ($\tau : V \to \mathbb{Z}$), which must be computed to generate its result. This result is a data structure composed by a number ($\delta : V \to \mathbb{Z}$) of floating point elements. Without losing generality, we assume the data structure is a continuous array. If a transformation to marshall/unmarshall the data is needed, its cost may be measured and added to $\tau(v_i)$.

The set of directed edges $E \subset V \times V$ associates nodes, creating a partial ordered set. The order relations express:

- Data dependencies: Let $e = (v_i, v_j) \in E$ be an edge. The computation represented by v_j needs the data computed by v_i to proceed.
  If v_i and v_j are assigned to different processors, v_j need to wait until the data computed by v_i is ready and it needs to access that data. Data dependencies may derive on communications of the data computed on v_i to all the $v_j : (v_i, v_j) \in E$.

For our purposes it is not needed to distinguish between different data sizes on the data-dependency edges coming from the same node. We assume that all nodes $v_j : (v_i, v_j) \in E$, need all the data generated in v_i to proceed. If needed, an integer value could be associated to the nodes to represent it.
For simplicity, we also assume that the computation v_j may not begin until all the data needed from all predecessor nodes $v_i : (v_i, v_j) \in E$ is fully available.

– Synchronizations: Other added synchronizations, which express order but not data dependency may be also represented by edges. To represent synchronization points we will use nodes with a special load value $v : \tau(v) = -\inf$. These nodes do not compute anything. Any edge from/to a synchronization point $(v_i, v_j) : \tau(v_i) + \tau(v_j) = -\inf$, represent only an order precedence. Synchronizations may be implement as empty communications or using other synchronization mechanisms.

2.2 File format

For our tools we are considering a simple file format, in plain text, to store the task graphs. You can see an example in Figure 1.

```
T: 11
R: 2
t0: 5.0000 10 s3: 1 2 3
t1: 3.0000 4 s2: 5 4
t2: 10.0000 12 s1: 4
t3: 2.0000 24 s2: 4 6
t4: 4.0000 5 s3: 7 8 9
t5: 1.0000 9 s1: 7
t6: 11.0000 32 s1: 9
t7: 7.0000 14 s1: 10
t8: 6.0000 8 s1: 10
t9: 1.0000 3 s1: 10
t10: 9.0000 40 s0:
```

Figure1. Example of a task graph file

The first line indicates the number of *task* nodes. The second line indicates the number of *resources* or values associated to the task node. We use the resource values to store the number of floating point operations and number of data elements computed. Other tools may add new resource values to the nodes (see the following sections).

Each node is identified by an integer number. For each node we have a line starting with t and its identifier. After the first colon we find the values of the resources for that node. The first one represents the number of floating point operations. We use a floating point number for compatibility with other previous tools and to easily store the $-\inf$ value for the synchronization nodes. The second resource value is the number of result data-elements computed.

After the resource values there is an "s" followed by the number of successor nodes, nodes $v_j : (v_i, v_j) \in E$, for which v_i is this node. After the second colon we find the identifiers of those successor nodes. The line for a node without successors has no data after the indicator of no successors (`s0:`).

3 Tool 1: Trivial scheduling algorithm

In this section we describe a tool to generate a trivial list scheduling for an input graph and a given number of processors.

3.1 Trivial scheduling algorithm

We have implemented a trivial scheduling algorithm. It receives as input a task graph G and a number of processors P. It simulates an execution based on the estimated times of each node, assigning nodes to processors as they are available. No communication or task-switch times are considered (they are low-level and machine-dependent parameters we do not want to consider at this stage of the study).

The algorithm uses an auxiliary data structure to keep track of ready to execute task nodes, the state of each processor, and the simulated global time. At the start, all processors are idle.

Each task node enters the ready set and may be assigned to a processor as soon as all its precedent task are already finished. Nodes without predecessors are available at the start, and they are put in the ready list at the beginning. When a node is assigned to a processor, its expected finishing time is computed adding its number of floating point operations to the global simulation time. Synchronization nodes with $-\inf$ time are considered to have execution time 0 (no floating point operation).

After ready nodes are assigned to processors, we search for the minimum finishing time of a task assigned to a processor. The first task to finish is eliminated and the associated processors become idle. The successor nodes of the task are checked and introduced in the ready list when all their predecessors are already done. When there are no more nodes in the ready list and all processors become idle, all the computation has been scheduled and the algorithm ends.

We use two new resources assigned to each node to specify: The number of processor ($p : V \to \mathbb{Z}$), and the number of sequence in that processor ($s : V \to \mathbb{Z}$). The output is the same graph but with the new resource annotations. The algorithm is presented in Figure 2.

3.2 File format

We use the same file format described above to store the scheduled graphs. We use two new resource values on each node, one to store the number of the processor assigned, and another to store the order of sequence in that processor. Numbers of processors are numbered from 0 to $P - 1$ in the graph. You can see an example of a scheduling for 3 processors in Figure 3.

```
TrivialScheduling(G,P), Output G
1. ready = {v_j : ∄(v_i, v_j) ∈ E}
2. proc[i] = idle
3. numTask[i] = 0 : i = 1, P
4. endingTime[i] = 0, i = 1, P
5. time = 0
6. while ready ≠ ∅ and ∃ proc[i]≠idle, i = 1, P
    6.1. for i=1,P do
            if proc[i]=idle and ready≠ ∅ then
                v = extractNode( ready )
                proc[i] = v
                if τ(v) ≠ − inf then; endingTime[i] = time + τ( v )
                p( v ) = i
                s( v ) = numTask[i]; numTask[i] = numTask[i]+1
            end-if
        end-for
    6.2. next = j : endingTime[j]=min( endingTime[i] : i=1,P )
    6.3. for all k : ( proc[j], k ) ∈ E
            visits( k ) = visits( k ) + 1
            if visits(k) = numPredecessor( k ) then; ready = ready ∪ {k}
        enf-forall
end-while
```

Figure2. Trivial scheduling algorithm

3.3 Implementation

We have implemented the program in JAVA, using Graph classes and tools already developed for other experimentation (see [2,3]). In those works we described a structural transformation called *SP-ization* which transform the task graph to an equivalent which may represent a pure nested-parallel structure or *SP-graph*. The program may apply such structural transformation before the scheduling. This transformation add new synchronization nodes (with $\tau(v) = -\inf$). In Figure 4 we present the result of the scheduling of the SP-ization of the graph introduced in previous examples. It has two new synchronization nodes.

The program also allows to obtain information about the critical path after the scheduling, some workload and structure parameters of the graphs; before and after the SP-ization transformation.

4 Tool 2: Generation of message-passing information

This section describes a tool which generates information to implement the scheduled graph as a message-passing program. The tool receives a scheduled graph as parameter, and the output is a list of activities to execute on each processor.

The basic types of activities are:

exec <num> <load> Execute *load* number of *flops* on the assigned processor. The parameter *num* is the number of the task.

```
T: 11
R: 4
t0: 5.0 10 0 0 s3: 1 2 3
t1: 3.0 4 0 1 s2: 5 4
t2: 10.0 12 1 0 s1: 4
t3: 2.0 24 2 0 s2: 4 6
t4: 4.0 5 0 3 s3: 7 8 9
t5: 1.0 9 0 2 s1: 7
t6: 11.0 32 2 1 s1: 9
t7: 7.0 14 0 4 s1: 10
t8: 6.0 8 1 1 s1: 10
t9: 1.0 3 2 2 s1: 10
t10: 9.0 40 0 5 s0:
```

Figure3. Example of a task graph file

send <target> <size> <tag> Send a message to processor *target* with *size* floating point elements with the given *tag*. The tag is the number of the task which has generated the data, and it will be used in the target processor to retrieve the appropriate data when needed.

recv <sender> <size> <tag> Receive a message from processor *target* with *size* floating point elements with the given *tag*. The tag is the number of the task which has generated the data we require to proceed.

sendSynch <target> 0 <tag> Send a message from a synchronization node to processor *target* with the given *tag*. The tag is the number of the synchronization task. Synchronization messages have no data to transfer. Their size value is 0 by convention at this specification level. We distinguish the synchronization messages from the normal ones because they may be optimized in a different way. See below.

recvSynch <sender> 0 <tag> Receive a synchronization message from processor *sender* with the given *tag*. The tag is the number of the synchronization task.

4.1 Version 1: Full-Communication

In the first version we consider a model where the tasks sends their generated output to all the other tasks that need it. In this simple approach, one message is generated from each node to each successor.

Sometimes, successor tasks are scheduled in the same processor as the sender task. With this first approach we will find messages sent from processor i to processor i, to be received by future tasks also scheduled on i. A subset of the successors may also be scheduled on the same processor. Thus, we may find more than one message from processor i to j, with the same length and content, to be received by different successor task scheduled on j.

The implementation of this version will need to use a buffered communication system to allow messages sent from i to i, and will be barely efficient, as buffers will be easily saturated by the amount of duplicated messages. The duplication factor is a pa-

```
T: 13
R: 4
t0: 5.0 10 0 0 s3: 1 2 3
t1: 3.0 4 0 1 s1: 11
t2: 10.0 12 1 0 s1: 11
t3: 2.0 24 2 0 s1: 11
t4: 4.0 5 1 1 s1: 12
t5: 1.0 9 0 3 s1: 12
t6: 11.0 32 2 1 s1: 12
t7: 7.0 14 0 5 s1: 10
t8: 6.0 8 1 2 s1: 10
t9: 1.0 3 2 2 s1: 10
t10: 9.0 40 0 6 s0:
t11: -Infinity 0 0 2 s3: 5 4 6
t12: -Infinity 0 0 4 s3: 7 8 9
```

Figure4. Example of a task graph after a SP-ization transformation

rameter of the edge-density of the graph and the number of processors. In general it may lead to buffer saturation and dead-lock conditions very easily.

4.2 Version 2: Communications using memory

In this version we optimize the programs eliminating the duplicated and unneeded messages using local memory buffers.

Communications from i to i may be easily substituted by the successor task accessing the data directly if it is still stored in local memory. Moreover, it is perfectly possible for many equal messages from i to j to be substituted by only one message. The first receiving task on processor j may store the data in a local memory buffer. The last receiving task on processor j may free the local buffer after using the data.

We extent the type of activities to be executed by a processor with the following ones:

malloc <num> <size> Allocate a local memory buffer for the data of task *num* with a number of *size* floating-point allocation units.
free <num> Free the local memory buffer for the data of task *num*.

4.3 Optimization of the synchronization messages

Synchronization messages carry no real data. The successors of a synchronization node which are scheduled on the same processors, do not need to receive anything. They are already scheduled after the synchronization node. Those synch-messages are eliminated from the output lists.

When several successors of a synchronization task are scheduled on the same processors, it is enough to send one message. The first scheduled successor in that processor must receive (wait for) it. No other successor task scheduled after need to do anything.

As they have been scheduled after the first successor in the processor, and the first successor has already receive (wait for) it, they know the synchronization has been already received. Symmetrically, for predecessors of a synchronization node, only the last one scheduled on each processor need to send a synchronization message.

As the synchronization messages carry no real data, there is no need to use local memory buffers or any special activity for them.

4.4 Implementation and output format

We have implemented both, version 1, and version 2 plus the synchronization messages optimization. They have been included in the same class of the Trivial Scheduling program and they may be selected by arguments.

The output is a series of list of activities. Each list contains the activities to do in one processors. The list starts with the number of the processor and the number of following activities. Each activity is described in one line, preceding by a tabular. You can see an example in Figure 5. These lists correspond to the SP-version example graph presented previously.

5 Tool 3: Execution program

We have developed a program to simulate the execution of the activity lists generated by Tool 2. The simulation engine is a C program using MPI for the communications.

The program loads the activity list file. Each process discards the input until it finds the activity list corresponding with its MPI rank. It traslates it to an internal representation and stores it in a dynamically allocated array of activities.

The basic engine is a loop that iterates on the array of activities index, visiting each activity sequentially. Inside the loop there is a generalized conditional which selects the code corresponding to the type of activity and executes it with the given parameters.

Implementation of communications The actual program execute use point to point buffered communications (MPI_Bsend, MPI_Recv). The buffer is reserved at the start of the program. Its size should be adapted to use a significant part of the target machine memory, trying to avoid deadlock conditions and buffer saturation delays. The synchronization messages are implemented as point to point messages of size 1.

Memory management The operate with the local buffers easily, the program allocates an array of pointers of as many elements as tasks in the graph. When a *malloc* activity is found, the pointer of the new allocated local buffer is stored in the element of the array corresponding to the task which produced and sent the data.

6 Experiments design

The tools have been used to experiment with some sparse-matrix factorization graphs.

```
NumTasks: 13
NumProc: 3
p0: 34
    malloc 0 10
    exec 0 5.0
    send 1 10 0
    send 2 10 0

    malloc 1 4
    exec 1 3.0
    send 1 4 1
    free 0

    recvSynch 1 0 11
    recvSynch 2 0 11
    exec 11 -Infinity
    sendSynch 1 0 11
    sendSynch 2 0 11

    malloc 5 9
    exec 5 1.0
    free 1

    recvSynch 1 0 12
    recvSynch 2 0 12
    exec 12 -Infinity
    sendSynch 1 0 12
    sendSynch 2 0 12

    malloc 7 14
    recv 1 5 4
    exec 7 7.0
    free 4
    free 5

    malloc 10 40
    recv 1 8 8
    recv 2 3 9
    exec 10 9.0
    free 10
    free 7
    free 8
    free 9
```

```
p1: 22
    malloc 2 12
    recv 0 10 0
    exec 2 10.0
    sendSynch 0 0 11
    free 0

    malloc 4 5
    recv 0 4 1
    recv 2 24 3
    recvSynch 0 0 11
    exec 4 4.0
    send 0 5 4
    send 2 5 4
    sendSynch 0 0 12
    free 1
    free 2
    free 3

    malloc 8 8
    recvSynch 0 0 12
    exec 8 6.0
    send 0 8 8
    free 8
    free 4

p2: 19
    malloc 3 24
    recv 0 10 0
    exec 3 2.0
    send 1 24 3
    sendSynch 0 0 11
    free 0

    malloc 6 32
    recvSynch 0 0 11
    exec 6 11.0
    sendSynch 0 0 12
    free 3

    malloc 9 3
    recv 1 5 4
    recvSynch 0 0 12
    exec 9 1.0
    send 0 3 9
    free 9
    free 4
    free 6
```

Figure5. Example of a program activity lists

Input graphs: These graphs have been obtained from civil-engineering examples using the DIANA tools [1,4]. These graphs includes tasks for the domain decomposition and for the factorization of the corresponding sparse-matrix. The tasks related to input/output tasks have been eliminated. Our set of examples has 6 different graphs with a wide range of sizes.

Nodes
58
211
772
2014
3142
3541

Table1. Parameters of the 6 example graphs

Task workload: The tasks of the input graphs include the number of flops to execute in the task, and the number of floating-point elements generated.

The simulations has been executed using three different scenarios. When the number of flops per task is very low the communication costs may dominate the computation and determine the performance of the simulation. When the number of flops per task is very high, it may dominate and determine the performance. Thus, we have used three different multipliers of the number of flops per task to generate three different cases. The first case uses the number of flops included in the original graphs. The experiments show that the number of flops per task is low for the speed of the machines used for the experiments. The other two ones are hypothetical cases generated to produce the other situations.

Case 1: Number of flops in task v = $\tau(v)$
Case 2: Number of flops in task v = $100 \times \tau(v)$
Case 3: Number of flops in task v = $500 \times \tau(v)$

SP-ization transformations: We have conducted all experiments with the normal and the SP-ized versions of the 6 example graphs. The results allow us to compare the impact of the SP-ization on the performance of the simulated programs. The study has been complemented with the critical path values obtained on the scheduled graphs for the original and SP-ized versions. These measures only take into account the number of flops per task, with no communication costs. It may be used as reference.

Platform and execution details: We have run simulations of the scheduled graphs in a 20 nodes Beowulf cluster. Each node is a PC with a i686, 1.8MHz AMD Athlon(tm) 64 Processor 3000+, with 512 Mb of RAM memory, running a Linux Gentoo distribution based on a 2.6.17 kernel. They are connected by a 100Mbit Ethernet through a single HUB.

The graphs have been scheduled and executed for 2,4,6,8,10,12,14,16,18, and 20 processors.

For each graph, number of processors, and load multiplier, we have executed the simulation program three times. We present mean results of the three executions.

7 Results

In this section we present tables containing the results of the experiments. Tables 2 and 3 show the critical path measured after the trivial scheduling of the graphs. The original ones, and the SP-ized versions respectively. The numbers are the sum of the number of flops to execute by the tasks in the critical path of the graph.

Tables 4 and 5 present the mean execution time of the simulation program in the Case 1. The number of flops executed by the tasks are the same as indicated in the graph nodes. The numbers indicate that the amount of computation is too low for the speed of the machines. This motivates the experiments on Cases 2 and 3.

Tables 6 and 7 present the mean execution time of the simulation program in the Case 2 (tasks execute 100 times the indicated number of flops). Tables 8 and 9 present the mean execution time of the simulation program in the Case 3 (tasks execute 500 times the indicated number of flops).

8 Conclusion

In this report we present a collection of tools developed to experiment in a real PC cluster with computations described as task graphs. The task graphs are scheduled using a trivial scheduling algorithm to a given number of processors. A simulation program allows to execute the scheduled graph. It simulates the expected number of flops on each task and uses MPI communications to move the virtual results of the computations across nodes when needed. The programs are integrated with other previous graph transformation tools.

We have use the tools to experiment with some graphs representing real parallel sparse-matrix factorizations. The graphs have been scheduled and simulated for different number of nodes in a Beowulf cluster of up to 20 nodes.

The results obtained will be used to further study the performance effects produced on an application due to the graph restrictions imposed by the synchronization structure used at the programming level.

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	32313	456247	132971632	13554287	226786373	1734287477
2	17326	233295	66842725	6807001	113646247	868474159
4	12450	142411	34366067	3475783	60974764	479147805
6	12450	120408	28967179	2383178	44433166	353894396
8	12450	110103	21687211	1903853	37141516	308279167
10	12450	107170	21370920	1609029	33586779	293185713
12	12450	105180	21167752	1406953	32433667	287589904
14	12450	104675	21165963	1299434	32120865	282222258
16	12450	104665	21165963	1219408	31652597	283291651
18	12450	104660	21165963	1186161	31493900	282902143
20	12450	104655	21165963	1173987	31420779	280883165

Table2. Critical path values (flops) for original graphs

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	32313	456247	132971632	13554287	226786373	1734287477
2	17336	241755	66697323	6795796	114495632	874410638
4	12450	144510	34290955	3494233	59850748	455541044
6	12450	122556	29475250	2446434	45078568	355133030
8	12450	114767	21576084	1949297	38989165	307132290
10	12450	112951	21390671	1670218	34796009	293627472
12	12450	112412	21406328	1490235	32933454	288198865
14	12450	112403	21386680	1381599	32368844	284071841
16	12450	112400	21379416	1325015	31952869	281905479
18	12450	112393	21372299	1303281	31629323	280220263
20	12450	112390	21372299	1291758	31462675	279950761

Table3. Critical path values (flops) for SP-ized graphs

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0004	0.0026	0.5257	0.0652	0.9456	6.9297
2	0.0073	0.0243	3.7383	0.2972	4.3902	15.2156
4	0.0093	0.0338	7.8550	0.3984	8.5470	23.9968
6	0.0123	0.0399	11.1690	0.4341	8.9079	34.1140
8	0.0131	0.0547	13.4199	0.4870	12.2750	40.6763
10	0.0140	0.0593	15.1039	0.5213	14.3172	42.6729
12	0.0144	0.1020	14.9358	0.6159	14.7880	50.1490
14	0.0338	0.1408	16.5249	0.6848	17.0110	56.2432
16	0.0954	0.2521	16.9672	0.7312	17.9509	59.6822
18	0.0428	0.1592	15.6196	0.9137	18.3481	66.0759

Table4. Execution times (sec.) for original graphs, Case 1

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0004	0.0026	0.5257	0.0652	0.9456	6.9297
2	0.0077	0.0241	3.7701	0.3024	4.5066	15.1302
4	0.0095	0.0345	8.6610	0.4266	8.7832	25.6282
6	0.0229	0.0457	11.6751	0.4718	10.9473	31.7380
8	0.0138	0.0604	13.5019	0.5423	13.2348	37.1039
10	0.0169	0.0897	13.5091	0.6740	15.0246	38.9399
12	0.0229	0.0954	15.8959	0.7520	16.5948	46.4865
14	0.0235	0.1026	17.1688	0.8046	18.8837	51.5815
16	0.0358	0.1946	16.4186	0.9845	21.8919	55.5240
18	0.0424	0.1177	16.4042	1.0727	21.8734	62.0420

Table5. Execution times (sec.) for SP-ized graphs, Case 1

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0146	0.2032	50.3330	5.9453	89.7590	644.6169
2	0.0120	0.1145	30.0374	3.1989	53.0208	383.5885
4	0.0135	0.0760	21.4734	1.6899	32.8978	263.0631
6	0.0171	0.0839	23.3574	1.2090	26.3883	213.3361
8	0.0174	0.0861	21.8340	1.0689	25.2242	182.2359
10	0.0199	0.0956	20.8964	0.9432	28.6459	169.8299
12	0.0209	0.1632	21.0241	0.9529	25.6299	173.6611
14	0.0236	0.1402	22.4163	0.9912	29.3823	169.9251
16	0.1008	0.2579	22.0449	1.0886	29.3822	175.6892
18	0.1315	0.1764	22.3547	1.1850	28.3800	164.6466
20	0.0787	0.2248	22.3523	1.3221	28.3724	173.1850

Table6. Execution times (sec.) for original graphs, Case 2

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0146	0.2034	50.1303	5.9472	89.7590	644.6169
2	0.0132	0.1156	29.4870	3.1866	52.5162	371.3238
4	0.0133	0.0819	22.8757	1.8001	33.7676	253.9873
6	0.0169	0.0911	24.0115	1.3503	29.4902	214.0679
8	0.0180	0.0995	21.6528	1.1515	28.6315	155.9000
10	0.0217	0.1195	18.5972	1.0702	28.7325	159.6572
12	0.0255	0.1227	21.3955	1.0936	27.6039	158.4761
14	0.0279	0.1435	22.4091	1.1239	28.9869	154.9551
16	0.0423	0.2019	21.9747	1.3413	31.2409	156.9554
18	0.0494	0.1779	21.9695	1.4839	31.2689	157.1160
20	0.0569	0.2413	21.4394	1.6323	30.6010	160.2695

Table7. Execution times (sec.) for SP-ized graphs, Case 2

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0619	0.8709	285.2752	26.0715	470.8132	3740.9639
2	0.0352	0.4531	161.9287	13.6828	266.6213	2257.5166
4	0.0293	0.2851	92.9841	6.9767	154.9688	1559.5437
6	0.0320	0.2614	88.1184	5.0146	120.2221	1220.7146
8	0.0337	0.2711	69.6818	4.2194	106.8767	998.3038
10	0.0372	0.2551	56.8591	3.4229	111.7432	879.9780
12	0.0390	0.2560	55.9663	3.2657	93.3055	884.5915
14	0.0505	0.3723	59.4718	3.1095	100.0617	851.0053
16	0.1176	0.3934	57.9675	3.0377	95.4795	842.4235
18	0.0667	0.3588	56.5620	3.0921	87.2268	819.9980
20	0.0930	0.4285	59.2632	3.1043	92.4381	825.1205

Table8. Execution times (sec.) for original graphs, Case 3

	Graph nodes					
Procs	58	211	772	2014	3142	3541
1	0.0619	0.8709	285.2752	26.0715	470.8132	3740.9639
2	0.0364	0.4713	156.6670	13.6033	269.0309	2191.8169
4	0.0299	0.2988	97.7383	7.1975	159.6758	1477.7560
6	0.0334	0.2731	89.3556	5.0961	121.5531	1224.7509
8	0.0351	0.2605	67.8374	4.3686	110.6939	887.6477
10	0.0396	0.2933	55.4447	3.5921	105.3672	839.5077
12	0.0387	0.2814	57.0753	3.4164	94.1481	831.6882
14	0.0482	0.3078	57.8287	3.2136	96.5626	762.3328
16	0.0528	0.3581	57.0839	3.2737	93.8339	768.3052
18	0.0669	0.4039	56.0482	3.5136	89.3808	753.8629
20	0.0790	0.5276	55.8891	3.4426	83.2551	756.1538

Table9. Execution times (sec.) for SP-ized graphs, Case 3

References

1. DIANA FE Program. WWW, Jan 2000. On http://www.diana.tno.nl/.
2. A. González-Escribano. *Synchronization Architecture in Parallel Programming Models*. Phd thesis, Dpto. Informática, University of Valladolid, Jul 2003.
3. A. González-Escribano, A.J.C. van Gemund, and V. Cardeñoso. A new algorithm for mapping DAGs to series-parallel form. Technical Report IT-DI-2002-2, Dpto. Informática, Univ. Valladolid, Apr 2002.
4. H.X. Lin. A general approach for parallelizing the FEM software package DIANA. In *Proc. High Performance Computing Conference'94*, pages 229–236. National Supercomputing Research Center. National University of Singapur, 1994.