Predicting the Impact of Implementation Level Aspects on Parallel Application Performance*

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

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

[2] Dept. of Information Technology and Systems, Delft University of Technology.
P.O.Box 5031, NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786168 Email: a.vgemund@et.tudelft.nl

Abstract

SP (series-parallel) programming models show well known advantages when program structure, cost estimation and scheduling are an issue. Transforming a parallel computation which DAG is NSP into its SP equivalent under the restricted SP synchronization framework could lead to a loss of efficiency. Nevertheless, it has been recently shown that this loss is often limited to tens of percents, which makes SP programming still a good alternative. In that previous research, we introduced an analytic cost model which allowed us to predict the effects of DAG topology on the execution time of a computation when applied at the programming level. In this work, we describe how this cost modeling approach can also be taken to the implementation level in order to explain or even predict the sometimes surprising effects on performance, caused by mapping decisions and language or implementation details. The application of this new approach to predict various performance effects on real applications showing both restricted and non-restricted synchronization structures is also shown in this paper.

1. Introduction

The use of *structured* parallel programming models which associated DAG has series-parallel structure (SP) brings a number of interesting advantages. In particular, improvements in terms of formal cost estimation [5, 13, 14] and scheduling [1, 4] enable the design of automatic compilation techniques as a pathway to portability and paramount optimal performance. Furthermore, SP programming is significantly easier and less error-pruning than unstructured programming [15]. Several examples of SP programming are found throughout the parallel programming domain: the Bird-Meertens Formalism [14], SCL [3], BSP [17], NestStep [10], SPC [5], Cilk [2], and, to some extent, OpenMP [11][1].

Beside the obvious advantages of SP programming models, a critical question is to what *extent* the ability to express parallelism is compromised when parallelism is restricted to an SP form. Expressing a parallel computation which DAG is originally non-SP (NSP) in terms of its SP equivalent form, implies *adding* synchronization arcs in order to fulfill all existing precedence relations. In some cases, this could mean *increasing* the critical path of the computation DAG. As an example, consider the NSP DAG shown at the left of Fig. 1, modeling the programming solution of a generic computational problem, given on the basis of an explicit synchronization or message-passing programming model. Fig. 1-b) represents an SP programming solution in which local barriers are selectively added to ensure series-parallel structure in the graph. In the NSP version, the black nodes could be well executed in parallel since they represent independent computations, while in the SP solution they have been implicitly serialized as a consequence of the added dependencies. In the framework of a reasonably balanced workload situation, the execution times of the SP solution

* The authors would like to acknowledge the support of the European Commission through grant number HPRI-1999-CT-00026 (the TRACS Programme at EPCC)

[1] Although OpenMP directives are oriented to SP programming, compliant implementations could be found that serialize a nested parallel region. Moreover, OpenMP provides a library for variable blocking that can be used to produce NSP synchronizations.

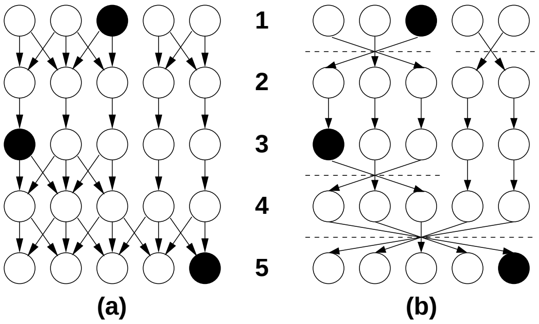

Figure 1. Sample NSP (a) and SP (b) graphs

would be similar to the NSP ones. If, as a pathological example, black nodes had a delay value of τ while the white ones could be executed in 0 time, the critical path of the SP solution would be highly increased. Nevertheless, unlikeness of such pathological workload distributions supports the acceptance of SP solutions for parallel processing on the basis of their unique benefits: an easy to understand programming style, adequacy for data-parallel synchronization structures and many task-parallel applications, and portability to any shared or distributed memory system.

In previous works [7, 8] we have already investigated the loss of parallelism associated to SP restricted form. Results showed that for most of the typical parallel applications scenarios, the loss is rather small (tens of percents), and performance is only slightly compromised. In our research, we introduced an analytic cost model [9] that allows prediction of the effects of DAG topology on the execution time of the computation. However, we learned that for some applications the computation graphs at the programming level can be greatly modified when implemented either on a specific machine model or using specific parallel programming tools. Now, we argue that a cost model can also be used at the implementation level to predict or explain the effects which mapping decisions or implementation particularities can have on performance.

Simple graphs and workload distributions will be used to model the new features introduced at the mapping or implementation level (including communication). The graphs are directly derived from the programs and all the knowledge at hand on the implementation details of the underlying parallel libraries being used.

Using either measured or estimated workload information, several experiments have been carried out over this graphs that show their ability to predict the essential behavior of different mappings of the algorithms. From this it can be concluded that the method can be confidently used in order to select both the best implementation and SP-ization technique for a given application. Also, off-line coarse performance estimations for high number of processors can be quickly evaluated as a byproduct.

An automatic or semi-automatic selection method can be derived from our cost models if a systematic way to obtain implementation task graphs from proposed mappings is provided and an automatic tool is built to simulate application executions from the graphs and workload information. Tools for critical path graph simulation from randomly distributed workloads are available. For highly complex graphs, we use a tool which generates C code simulation programs from graph specifications written in a simple language which supports different types of nodes, each type with its own workload pdf [16]. As for the systematic construction of implementation task graphs from mapping alternatives, it is the central part of this work to present a study of the balance between simplicity and accuracy in order to achieve that goal.

In Section 2 we present our cost model to measure performance effects at implementation level. A step by step example of the construction of the graph model of a sample application is given in Section 3. Section 4 summarizes the results obtained from graph simulations, discussion of the some observed effects, and a comparison with real data gathered from program executions.

2. Cost model

For our cost model at implementation level we extend graph basics from previous studies in order to accommodate node type annotations, so that as many node types as needed (tasks, communications, barriers,...) can be handled. A given workload probability distribution function can then be assigned to each node type.

2.1. Graph preliminaries

Every program is modeled by a AoN (*Activity on Nodes*) DAG, denoted by $G = (V, E)$, to represent the set of nodes (V) and dependencies (E) associated with a running program instance of given input-data size. Every node represents a task and belongs only to one node *type*. All the nodes of the same type have the same *pdf* for the *load* or *delay* values, which represent execution times. Edges only mean dependencies and have a "no delay" value. Communication delays must be included in terms of their own specific task types.

SP DAGs are a proper subset of task graphs and include only *series* and *parallel* structures, built by recursive application of fork/join and/or series compositions [6].

Let $W : V \to \mathbb{R}$ denote a *workload distribution* over the set of nodes of a given type. Given G and W, *Critical path* or *Cost*), $C(G)$, is defined as the maximum of all the accumulated delays computed over the set of full paths of the graph.

An *SP-ization* technique means the set of rules to transform an NSP graph to SP preserving all the precedence relations. So, it is a graph transformation ($T : G \to G'$) where G is NSP and G' is SP, and all the dependencies in G are directly or transitively expressed in G'. New nodes and dependencies may exist in G' which could increase the critical path. Let γ denote the minimum relative increment in the critical path caused by any given T:

$$\gamma_T = \frac{C(G')}{C(G)} \quad , \quad \gamma = \min_T(\gamma_T).$$

Clearly, γ is a function of DAG topology and workload W. For a given W, there exists a transformation T such that γ_T is minimal. However, the exact W is either not known or highly data-dependent, in general. Therefore, it cannot be used to decide the optimal SP solution for a given DAG. Thus, upper bound $\hat{\gamma}_T$ and mean value $\bar{\gamma}_T = \mathrm{E}[\gamma_T]$ for *any* possible W become relevant parameters for our study. Although a general optimal SP-ization does not exist for any G and W, we have experimental evidence that, except for pathological (extremely unlikely) values of W, the following conjecture holds [5]: $\forall G, \exists T : \gamma_T \leq 2$.

2.2. Deriving implementation level graphs

Tasks identification: We consider a task to be an atomic activity which can be executed independently of the local state of other activities (tasks). Tasks and communications must be identified from program specifications and mapping information for the implementation programming model and tools.

In explicit synchronization models (e.g. message-passing), it is easy to detect beginning and end of a task, since it is coded in the set of sentences between two consecutive communication directives. Since we have used MPI implementations in our examples, task identification is readily solved.

In those programming models in which implicit communication is used (e.g. shared memory), we must get to the implementation details of the model in order to interpret which memory accesses or primitives force synchronization and, thus, the ending point of a task.

Even worse, in certain kinds of applications, the execution of a communication directive is data dependent, which means that graph topology is also to be data dependent. In these cases, the graph should be dynamically constructed from executions with real data or several template graphs should be matched against typical execution conditions to seek for synchronization structures.

Once the tasks are identified, they should be classified in terms of statistical characteristics of their execution times using as many types of task nodes as needed, although the variety should be kept to a minimum (perhaps by means of more complicated distribution laws). In most situations, specially for highly parallel and scalable applications, the kernel of the application could be associated to just one node type which gathers similar pieces of code.

Communication model: In order to avoid highly complex graphs, we have used quite a simplistic communication model representation. Similar programming tools may hide very different communication details and even two different implementations of the same tool could well lead to different node types for the communication tasks. We must simplify, but keeping enough detail as to get a trustful approximation.

In our examples we have chosen two different implementations of the MPI interface: one for CrayT3E and the other for a Beowulf system. Both implementations share common characteristics that have allowed us to model both communication schemes in the same simple way. We propose the following communication model for the MPI implementation, which requires four types of communication nodes ($V_{c1}, V_{c2}, V_{c3}, V_{c4}$).

- An isolated point to point communication is considered to be atomically represented by only one node (V_{c1}).
- When a sequence of communications is being issued in a program, each of them should be divided in two nodes:
  1. The first node (V_{c2}) will correspond to the message buffering and communication initialization phase, which will also delay the beginning of the next communication.
  2. The second node (V_{c3}) will correspond to the communication itself and the reception of the message, and it will only delay the start of the receiving task.
- A broadcast communication will be represented by p (number of processors) simultaneous point to point communications (V_{c1}). This complies with the MPI interface, which implementation of a broadcast operation can or cannot be synchronized. Implementations of the broadcast operations could spawn the message through processors in different ways, typically by means of a virtual tree structure. For simplicity, we will consider all the nodes to have the same workload distribution. Measures in real machines support the accuracy of this simplification.

  A communication node will connect the communication initiating task with the next task in the same processor, to represent the cost of issuing the broadcast.

- Full barrier synchronization is a common aid for SP-ization and a new different node type is introduced for it (V_{c4}). For any given number of processors barrier delays are easily predicted from experimental measurements.

Workload information: Workload information can be obtained from sample executions of the code or it could be derived from any well-known formal prediction method.

In this work, we use *Gaussian* random distributions to calculate the load in each node. Mean and standard deviation values for task and communication nodes had been experimentally obtained in a previous work [7] from executing codes of the MPI implementations discussed here both in the CrayT3E and in the Beowulf system.

3. Example

Now, we show how to use the previous simplified cost model to predict important information about the effects of SP-ization techniques when faced to different implementations of the same algorithm. Further examples with similar results are available [12].

We have chosen an LU reduction application because it has characteristics which challenge simple graph modeling. At programming level, with no resource restrictions, all possible parallelism is exploited. In the resources mapping phase, however, data partition may affect the task structure of the application. The problem with LU reduction is the variety of possible mappings and implementations that can severely affect the synchronization pattern of the original program model shown in Fig. 2. We have chosen an implementation that brings load and communication balance distributing matrix rows with a stride equal to the number of processors. Thus, for p processors, processor i will store the following set of matrix rows:

$$R_i = \{r_i, r_{(p+i)}, r_{(2p+i)}, r_{(3p+i)}, \ldots\}$$

As iterations progress, executing tasks update less amount of data. In this way, modeling any task node load with the same random distribution is a rough approximation. However, we find enough accuracy for our purposes using statistical information from sample executions (mean values and deviations). The communication pattern plays the most important role in the results.

The stride used for rows distribution creates communication balancing. In each iteration one different processor has the row that all of them need to update the rest of their data. Cycling the processor that sends the row changes the graph topology.

Moreover, two implementations for the communication stage have been considered:

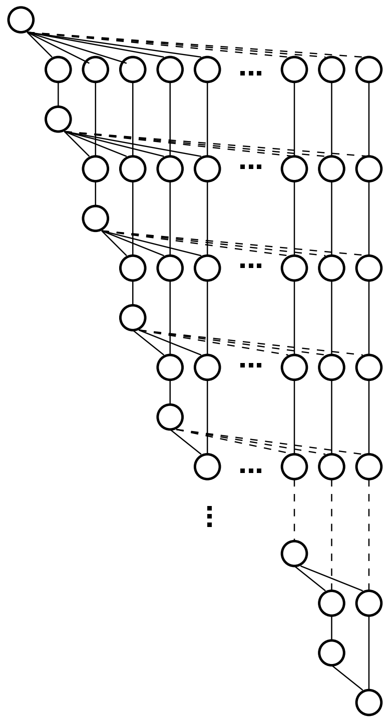

Figure 2. LU reduction: Programming level

1. **IMP-1:** A simple loop of point to point communications (See algorithm in Fig. 3).
2. **IMP-2:** A broadcast operation (See algorithm in Fig. 4).

SP versions of both implementations are easily constructed adding a full barrier synchronization after the communication stage of each iteration.

The corresponding graph models for a 4 processors mapping are shown in Fig. 5. The following graphics representation has been used in this figure for the different node types.

Tasks	V_t	White nodes
Full comm.	V_{c1}	Black nodes
First phase comm.	V_{c2}	Big dark grey nodes
Second phase comm.	V_{c3}	Small dark grey nodes
Barriers	V_{c4}	Light grey nodes

4. Results

Graph models have been used to simulate performance in a CrayT3E and a Beowulf system. Obtained results are presented here and predictions and comparisons with execution times of real implementations are discussed.

Mean values and deviations for the load on different types on nodes are provided by statistical experiments with real codes. The size of the problem is scaled up with the

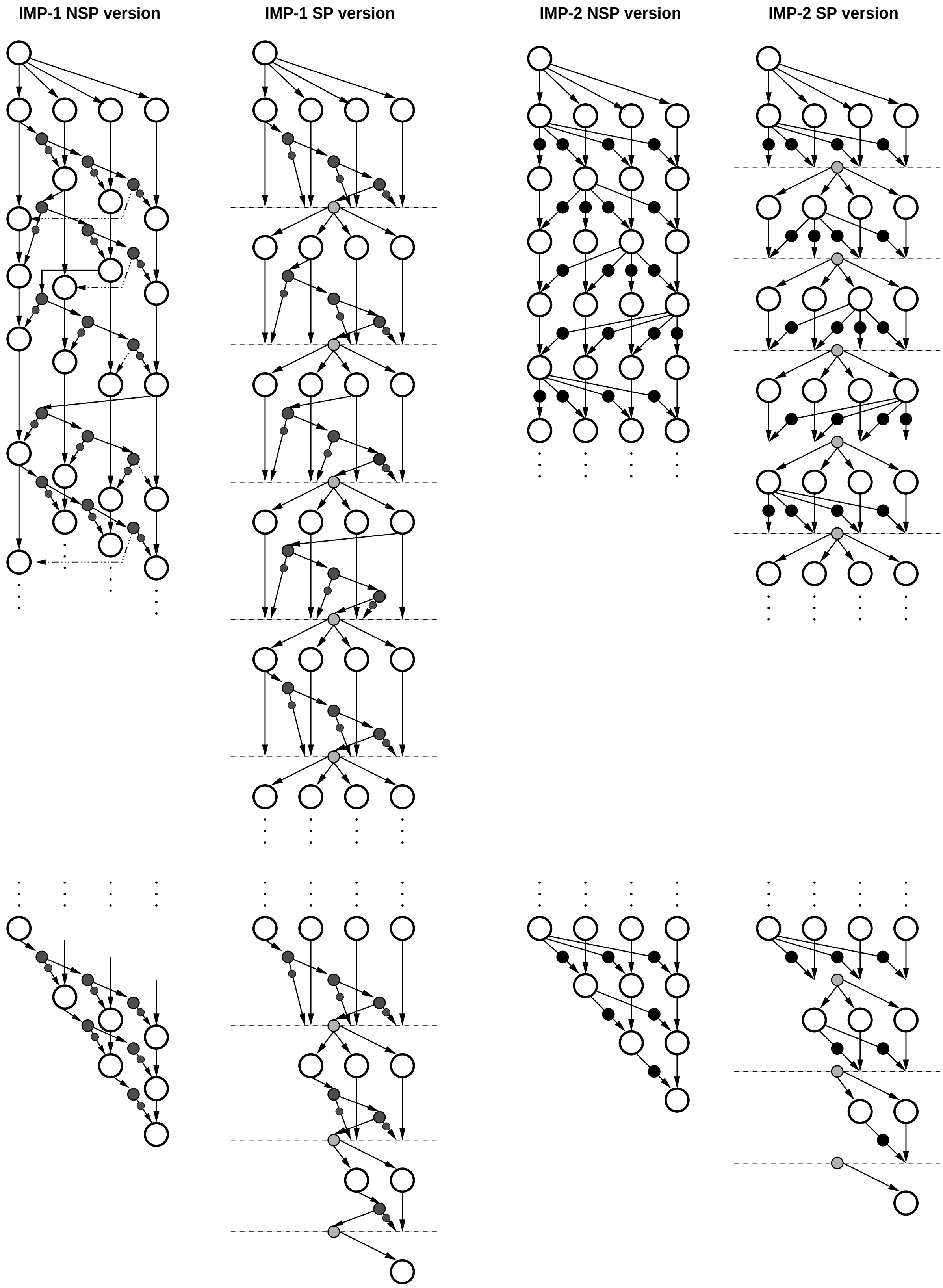

Figure 5. Implementation models of LU reduction with distributed rows

CrayT3E

# Proc.	V_t		V_{c1}		V_{c2}		V_{c3}		V_{c4}	
	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2
16	19	0.28	0.66	0.00001	0.22	0.00001	0.44	0.00001	0.104	0.000001
32	19	0.28	0.88	0.00001	0.22	0.00001	0.66	0.00001	0.143	0.000001
64	19	0.28	1.32	0.00001	0.22	0.00001	1.10	0.00001	0.175	0.000001
128	19	0.28	2.22	0.00001	0.22	0.00001	1.98	0.00001	0.224	0.000001

Beowulf

# Proc.	V_t		V_{c1}		V_{c2}		V_{c3}		V_{c4}	
	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2
2	19	0.31	1.00	0.0001	1.00	0.0001	0.00	0.0001	0.50	0.00001
4	19	0.31	2.00	0.0005	1.00	0.0001	1.00	0.0001	1.00	0.00010
8	19	0.31	6.00	0.0007	1.00	0.0001	5.00	0.0001	3.00	0.00100
16	19	0.31	9.00	0.0010	1.00	0.0001	8.00	0.0001	6.00 / 2.00*	0.00100

Table 1. Load estimated times (milliseconds)

```
DO iteration=0,n
   // COMMUNICATION
   IF mod(iteration,p) = myself THEN
      DO proc=1,p
         IF p ≠ myself THEN
            Send(proc,row)
         ENDIF
      ENDDO
   ELSE
      Receive(row)
   ENDIF

   Barrier (ONLY SP VERSION)

   // COMPUTING: UPDATE ROWS
   ...
ENDDO
```

Figure 3. LU reduction: IMP-1

```
DO iteration=0,n
   // COMMUNICATION
   IF mod(iteration,p) = myself THEN
      Copy row in sending position
      Broadcast(row)
   ENDIF

   Barrier (ONLY SP VERSION)

   // COMPUTING: UPDATE ROWS
   ...
ENDDO
```

Figure 4. LU reduction: IMP-2

number of processors, using matrices of double data size when doubling the number of processors. The initial size for each machine has been empirically calculated to obtain similar task times. Tab. 1 shows the estimated parameters in the two machines considered, for the number of processors available. The load values have been rounded up before using them for graph simulations.

CrayT3E has faster mean communication times with lower deviations, even for large number of processors. Thus, the results of the simulations will be more reliable. The barrier synchronization system is also more efficient when scaling up. It is noteworthy the strange effect of barrier times for 16 processors in the Beowulf system. After point to point communications full barrier time still grows up (6.00ms). However, after a broadcast operation the time is even smaller than with less processors (2.00ms). It seems that an optimization of either the MPI implementation or the hardware is carried out when a barrier is issued after a broadcast with all the processors in the system. Communication mean times in the Beowulf are in general not so much reliable, as unexpected peaks are commonly found.

Comparative results from real execution times and predictions with the graphs are shown in Fig. 6. In all cases the performance predicted times are similar to the real measures, and they show the same slope tendencies.

The first effect observed is that IMP-2 scales better than IMP-1. The graph model can be used to explain the effect. IMP-1 creates a strange communication pattern, that is not well balanced. The loop is always sending messages to processors in strict numbering order while the origin of communications is cycling. In Fig. 5 (NSP IMP-1), we can see that the first phase of each point to point communication, accumulated for all *send* primitives, is not evenly distributed to other processors. Depending on how significant is the mean load of the V_{c2} nodes compared to V_{c3}, the overall performance can be badly affected. Moreover, as more processors get involved, the delay grows higher. The relative importance of V_{c2} vs. V_{c3} loads is higher in the Beowulf system than in the CrayT3E, as shown in Tab. 1. However, we use many more processors in the CrayT3E. Thus, the final effect is even more noticeable in CrayT3E. The broad-

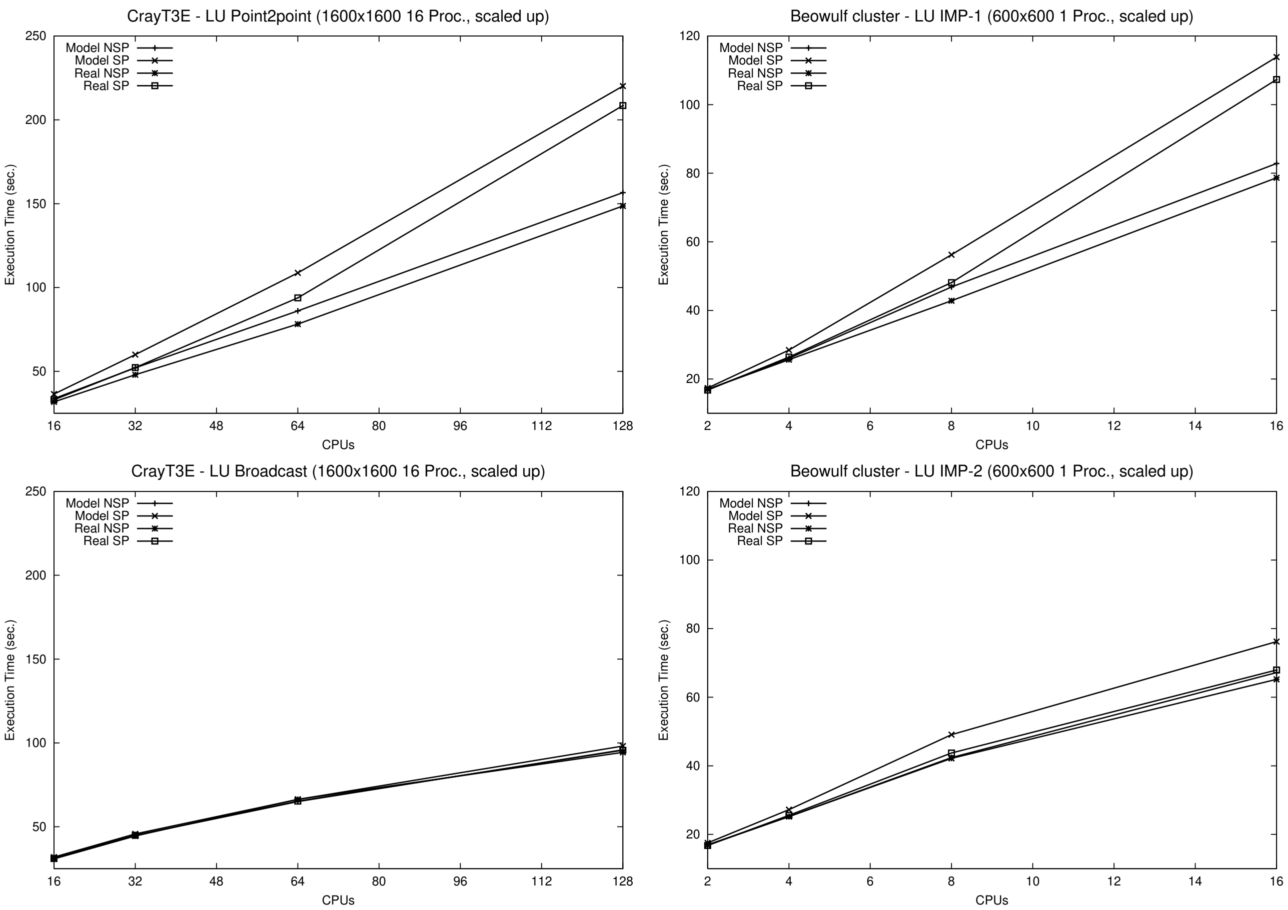

Figure 6. Performance comparison: Real times vs. predicted times

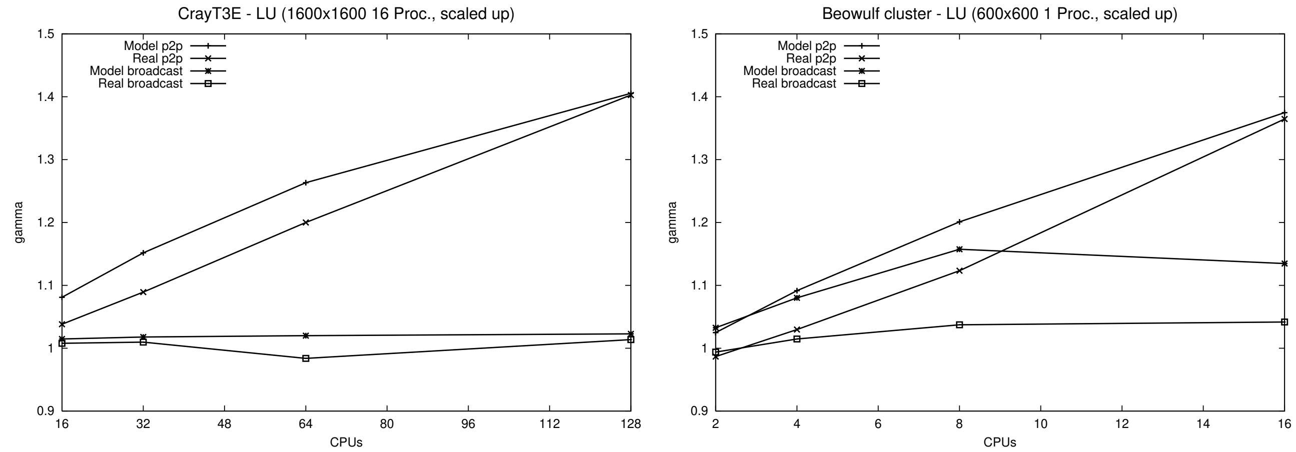

Figure 7. γ comparison: Real times γ vs. predicted times γ

cast primitive of IMP-2 scales clearly better than the IMP-1 for both NSP and SP version.

Changing the loop indexes to cycle with the processor initiating the communications will alleviate part of the problem. However, the SP version that adds a barrier after the communication stage is always delaying all processes up to the accumulation of all first phase communications. When the number of processors grows, the problem gets linearly worse. Fig. 7 shows the values of γ (performance loss due to SP-ization) for real and predicted results. It can be seen that IMP-2 is perfectly suitable for SP programming, as the γ values keep almost constant when the application is scaled up. The loss of performance in SP version of IMP-2, clearly seen in the Beowulf case, is generated by the times needed for barrier synchronizations when the number of processors grow up. Better barrier synchronization mechanisms will diminish this loss. The peeks in communication/synchronization times in the Beowulf also helps this grow.

We must also point out the curve slope differences between measured and predicted γ values. Since we used simplified graph models in our simulations, some irregularities appear, specially in the central part of the number of processors axis. In general, predicted results are higher than real measures for load times for nodes, which is a consequence of the rounded up approximations we have used. As the number of processors (and thus the execution time) grows, the relative importance of accumulated rounding errors is smaller. All this could perfectly explain the shape differences of the γ prediction curves.

With the predictions obtained with our simplified graph models we can recognize that IMP-2 is in general better than IMP-1 due to the implementation on the underlying message-passing library. IMP-1 is specially not well suited for SP-ization with full barrier synchronization. On the other hand, IMP-2 behaves correctly in SP version, providing a very small loss of performance.

5. Conclusion

In this paper we show that simple graph models representing only the basic features of communication and synchronization patterns can be accurate enough to predict the main effects of application mappings on performance, and specially to detect any further impact of any SP approximation techniques used.

Automatic or interactive tools to derive the simulation graphs from program mapping specifications could be developed. Hence, new possibilities are opened to exploit the benefits of SP programming techniques in compiling techniques and tools for parallel computing.

References

[1] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. Annual Symposium on FoCS*, pages 356–368, nov 1994.

[2] The Cilk Project.
WWW. on http://supertech.lcs.mit.edu/cilk/.

[3] J. Darlington, Y. Guo, H. To, and J. Yang. Functional skeletons for parallel coordination. In *Europar'95*, LNCS, pages 55–69, 1995.

[4] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET–UCT series–parallel graphs on two processors. *DIMACS Series in DMTCS*, 162:323–340, aug 1996.

[5] A. v. Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, July 1997.

[6] A. González-Escribano, V. Cardeñoso, and A. v. Gemund. On the loss of parallelism by imposing synchronization structure. In *Proc. 1st Euro-PDS Int'l Conf. on Parallel and Distributed Systems*, pages 251–256, Barcelona, July 1997.

[7] A. González-Escribano, A. v. Gemund, and V. Cardeñoso-Payo. Measuring the performance impact of SP-restricted programming in distributed-memory machines. In *Proc. XI Jornadas de Paralelismo*, pages 59–64, Granada (Spain), September 2000.

[8] A. González-Escribano, A. v. Gemund, V. Cardeñoso-Payo, J. Alonso-López, D. Martín-García, and A. Pedrosa-Calvo. Measuring the performance impact of SP-restricted programming in shared-memory machines. In *Proc. VecPar'2000*, pages 715–728, Porto (Portugal), June 2000.

[9] A. González-Escribano, A. v. Gemund, V. Cardeñoso-Payo, H.-X. Lin, and V. Vaca-Díez. Expressiveness versus optimizability in coordinating parallelism. In *Parallel Computing, Fundamentals & Applications. Proc. Int. Conf. ParCo99*, pages 526–533, Delft (The Netherlands), August 1999. Imperial College Press.

[10] C. Kessler. NestStep: nested parallelism and virtual shared memory for the BSP model. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas (USA), June-July 1999.

[11] OpenMP organization.
WWW. on http://www.openmp.org.

[12] PGamma: Measuring the performance impact of SP programming: Resources and tools. WWW. on http://www.infor.uva.es/˜arturo/pgamma.

[13] R. Sahner and K. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Trans. on Software Eng.*, 13(10):1105–1114, Oct 1987.

[14] D. Skillicorn. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[15] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[16] A. Vaca-Díez. Tools and techniques to assess the loss of parallelism when imposing synchronization structure. Technical Report 1-68340-28(1999)02, TU Delft, Mar 1999.

[17] L. Valiant. A bridging model for parallel computation. *C.ACM*, 33(8):103–111, Aug 1990.