

# Generic Concurrent Modules for Systolic Computations

Brian J. d'Auriol\*

Department of Mathematics and Computer Science  
The University of Akron  
Akron, Ohio, U.S.A. 44325-4002

Virendrakumar C. Bhavsar†

Faculty of Computer Science  
University of New Brunswick  
Fredericton, N.B., E3B 5A3, Canada

**Abstract** *We have proposed an Integrated System Model (ISM) to address the problem of high development costs of concurrent programs. The ISM is composed of multi-level modules that abstract both the high-level program design as well as the low-level optimization. Various possible implementations of systolic algorithms on multicomputers are considered. A generic program structure sufficient to represent these various implementations is also given. The intrinsic parallelism of a given systolic algorithm is embedded in the module level parallelism of the generic program structure. Several experiments on the concurrent implementations of the generic software modules for these algorithms are presented.*

**Keywords:** Concurrent Software Design, Concurrent Modules, Systolic Computations, Parallel Processing, Multicomputers, Transputers

## 1 Introduction

We consider a number of factors which affect the development and management of concurrent software. In particular, we consider those implementation factors which affect the design, coding, testing and measurement of concurrent programs. Included in this category are multiple possible parallel algorithms, the correspondence between such algorithms and the machine architecture, the specification of one or more integrated program designs, the translation(s) from the high-level design to code and evaluation techniques. Additional factors such as selection of source language and application do-

main also indirectly affect program development. The program developer needs to take into account these issues. Despite the availability of tools, the program developer remains significantly involved in the development and management of concurrent software. Thus, higher development costs are often incurred in concurrent software development.

In this work, we are interested in decreasing the program development time required to implement numerically intensive computing applications on parallel computers. It is assumed that a corresponding decrease in financial cost would also be realized. Consequently, we primarily concentrate on the implementation issues. Other issues such as performance and efficiency are also considered. We restrict ourselves to only algorithms which have systolic solutions and their execution on MIMD type multicomputers. However, the approach is also applicable to other types of algorithms and machines.

The identification and exploitation of the intrinsic parallelism in a systolic algorithm is well known (see for example [1, 2]). Most often, parallelism in systolic algorithms is at a very low level. Modern day multicomputer systems are well known to offer efficient execution only when there is coarse grained parallelism. Systolic algorithms can be partitioned to obtain coarse grained parallelism, see for example [1, 2, 3].

We observe that the traditional role of systolic algorithm partitioning leads to two problems. First, despite the adjustment of parallelism to coarse grain, little decrease in the program development time can be expected (since implementation factors are not addressed). Second, often a specific hardware architecture must be previously assumed and the partitioning is such that computations are

---

\*Partially supported by NSERC doctoral fellowship.

†Partially supported by NSERC grant, OGP0089.

‘mapped’ to this architecture.

One goal of traditional software design is to provide well-defined functional definitions of software modules. The two types of software modules that are of interest to us are concurrent modules, which provide for inter-module parallel execution, and generic modules which abstract the functionality but only partially include certain implementation details. Concurrent generic modules allow for the expression of parallelism while at the same time provide a program structure that can be used during the implementation. We refer to the parallelism introduced by concurrent modules as *module level parallelism*.

Note that the systolic algorithm partitioning process does not by itself allow for any definition of such module level parallelism. However, we report in this paper that an alternative interpretation of the partitioning process can be used to construct a basic definition for module level parallelism. Other aspects related to the goal of decreasing program development time are also addressed.

We propose an Integrated System Model (ISM) that both provides for a programming environment which would lower program development costs for concurrent programs and the efficient implementation of systolic algorithms on multicomputers. The ISM provides for a reduction in parallel program implementation development time by providing a mechanism to: (a) identify intrinsic parallelism in the given systolic algorithm, (b) consider various implementations (some of which may be very different), (c) estimate execution time performance, and (d) automate the construction of a selected (and thus efficient) implementation.

Our work differs from the traditional restructuring (parallelizing) compilers in that such compilers (often) identify sections of programs which inhibit parallelization and then restructure these sections to reduce and eliminate inhibitors [4]. Thus restructuring compilers target program construct level analysis and manipulation whereas our work is aimed at analyzing and manipulating generic concurrent program modules.

The proposed approach is consistent with that of concurrent program archetypes which have been proposed as a method to reduce the effort required to develop and implement parallel programs [5]. Our approach is also motivated by algorithm skeletons [6, 7] which provide for abstractions of useful parallel computational structures. Common elements between our model and skeletons include performance modeling and parameterization.

This paper is organized as follows. In Section 2, we present the initial formulation of the problem

and the approach taken while in Section 3, we describe an overview of the ISM. We summarize the procedure required to derive the generic concurrent modules in Section 4. Some of our experimental results relating to key aspects of the proposed system model are presented in Section 5. Finally, conclusions are given.

## 2 Approach

We first consider the software design phase and then the coding phase. Let  $S$  be a software system representing the final (coded) implementation of the input algorithm,  $\mathcal{A}$ .  $S$  can be described as a combination of software modules  $M_i$ , thus,  $S = \{M_1, M_2, \dots, M_n\}$ . The definition, nature, behavior and abstraction of these modules are not usually known *a priori*, rather, standard software engineering techniques can be applied to detail each module and the various relationships between the modules.

Let a module  $M_i$  be composed of a structure component  $M_i^s$ , and a behavior component  $M_i^b$ : thus,  $M_i = M_i^s \cup M_i^b$ . A module’s *structure* refers to any code representation that does not depend on any analysis of  $\mathcal{A}$  whereas a module’s *behavior* refers to any code representation necessary to represent aspects of  $\mathcal{A}$ . Thus, the structure is said to be invariant with respect to  $\mathcal{A}$  and the behavior dependent on  $\mathcal{A}$ . It follows that  $M_i^s \cap M_i^b = \{\}$ . Let  $\mathbf{M}^s$  denote the set of all module structures in  $S$ :  $\mathbf{M}^s = \{M_1^s, M_2^s, \dots, M_n^s\}$ .  $\mathbf{M}^b$  is similarly defined for all module behaviors. Note that  $\mathbf{M}^s$  must be defined prior to  $\mathbf{M}^b$ . In [8, 9], we showed that that  $\mathbf{M}^s$  is derivable (in some sense) from  $\mathcal{A}$ .

## 3 Overview of the Integrated System Model (ISM)

The ISM consists of four levels as shown in Figure 1. At the top level, two models are defined: the Generic Parallel Program Model (GPPM) and the Implementational Object Model (IOM). These two models embody the distinction between the derivation of  $\mathbf{M}^s$  and all other aspects of the compiling system, respectively. Under GPPM,  $\mathbf{M}^s$  is derived by considering the parallel execution of  $\mathcal{A}$  as would be executed on a parallel machine. Thus, the intrinsic parallelism of  $\mathcal{A}$  firstly needs to be identified and subsequently exploited such that it can be mapped to the parallel machine. GPPM relies on a lower-level model to accomplish the first requirement. The IOM provides for a generic description of necessary components of an executable application, for example, the source code of that implementation.

As such, various instantiations of IOM are possible, each instantiation representing exactly one possible implementation (we term such an instantiation an Implementation Object (IO)). Clearly, each IO represents, at the least,  $S$ ; therefore, the definition(s) of  $\mathbf{M}^s$  from GPPM must be firstly available. Thus, the IOM represents a generic description and only through an application of GPPM can a particular IO be instantiated. The IOM relies on several lower level models as well.

The Systolic Array Model (SAM) is a sub-model of GPPM that allows for the derivation of a systolic graph (essentially, such a graph is obtained from systolic array synthesis techniques applied to the data dependency graph of the algorithm) from  $\mathcal{A}$ . Systolic processing is a mature field and techniques are available to: (a) re-interpret a systolic algorithm as a systolic graph, and (b) generate the systolic array implementation (e.g. VLSI implementation) of the corresponding systolic graph (see for example, [1, 2]). In our case, we are interested only in the derivation of the systolic graph.

There are two abstractions for GPPM: (a) set of generic program modules for a given systolic algorithm, and (b) a definition of the modules constructed for a given input algorithm. These abstractions imply that GPPM is applied in two cases: (a) when no IO has been defined for a specific processor topology, (i.e., an instantiation of an IOM suitable for the derived  $\mathbf{M}^s$  has not yet been done), then GPPM is used to generate the basic definition of such an IO, and (b) when an IO suitable for a particular  $\mathbf{M}^s$  exists, the GPPM is used to transform the input algorithm so that it may be combined with the structure. Case (b) implies that the Program Structure Model (PSM) of the IO has been sufficiently well defined so as to allow for this combining operation.

Since the abstraction of an IO is to represent a particular implementation, there is a requirement to represent  $\mathbf{M}^b$  and fuse  $\mathbf{M}^b$  with  $\mathbf{M}^s$ . These representations must be defined in the IOM as suitable generic code. The lower-level PSM provides the necessary data and procedural methods required to support this functionality. For example, the implementation could be represented in different languages, or, different (in some limited sense) processing algorithms (as in the case of communication routers, see Section 5). Each language representation or algorithm would be encapsulated by the same structure, hence, the same IO would be suitable.

The Performance Model (PM) is mainly concerned with the prediction of the execution time for an executable application based on a particu-

lar IO. The PM is parameterized to some extent so that it may be applied to various architectures, as modeled by the Machine Model (MM).

## 4 Generic Parallel Program Model (GPPM)

A summary of GPPM procedures to derive  $\mathbf{M}^s$  is described here; details may be found in [8, 9].

The principal components of GPPM are: (a) selection of a systolic graph representation (denoted by  $\hat{G}$ ), (b) translation of  $\mathcal{A}$  to  $\hat{G}$ , (c) selection of a particular partitioning of  $\hat{G}$  thereby inducing an associated processor topology graph, and (d) representation of the programming features of all such induced processor topology graphs by  $\mathbf{M}^s$ . The SAM provides for both the underlying foundation for the first three as well as the translation of  $\mathcal{A}$  to  $\hat{G}$ . The selected graph is illustrated in Figure 2.

We generate a set of possible multicomputer processor topologies by considering various partitions of  $\hat{G}$ . Our viewpoint is that a particular multicomputer must have the ability to be configured in at least one of these pre-determined topologies. For example, in Figure 2, *orthogonal* partitioning illustrated by lines of the form  $c_o$ , will induce the graph shown in Figure 3 whereas *diagonal* partitioning illustrated by lines of the form  $c_d$  will induce the graph shown in Figure 4. Temporal partitionings as illustrated by the partition  $c_t$  in Figure 2 lead to a farming implementation where the various groups of computations that have been determined by the partition are ‘farmed’ out to processors. In our work, we have selected a linear processor topology as the required configuration induced by a temporal partitioning.

Lastly, we have identified  $\mathbf{M}^s = \{M_F^s, M_{IO}^s, M_R^s, M_C^s, M_{CM}^s\}$  where  $M_F^s$  contains the computational process (the functional equivalent of  $\mathcal{A}$ ),  $M_R^s$  provides for necessary communication routing on the multicomputer,  $M_{IO}^s$  provides user file services,  $M_{CM}^s$  contains communication specific data type definitions and  $M_{CM}^s$  provides for software configuration. These modules are illustrated in Figure 5 along with other aspects relevant to such software design. Essentially, these generic modules allow the representation of module level parallelism.

In summary, given an input systolic algorithm  $\mathcal{A}$  and the definition of  $\mathbf{M}^s$  above, various instantiations of  $\mathbf{M}^s$ , one per each candidate processor topology, represent the implementation of  $\mathcal{A}$  on a target multicomputer. Since each instantiation represents an implementation for a different processor topology, many essential details of the program

structure necessary to support the implementation are captured. This relieves a significant burden from the programmer: hence, lower development costs may be realized.

## 5 Experimental Results

A number of experiments have been conducted to illustrate key points of the ISM. In particular, we report: (a) the development of code according to the program structures of three derived Implementational Objects, (b) the sensitivity of the code structures with respect to performance, and (c) and the execution time comparison between the concurrent and sequential implementations.

We have defined the three implementational objects (IOs) by applying the GPPM model as described in this paper and the subsequent IOM components necessary (not detailed in this paper) in order to conduct our experiments. Following [10], we define the three IOs as: (a) the Diagonal IO, (b) the Farming IO (which results from temporal cuts on the systolic graph), and (c) the Sequential IO. These implementations were manually coded according to their corresponding derived module structures.

The implementations considered to date have been influenced by the occam 2 language and transputer architectures. The experiments discussed in this section were conducted on a nine transputer multicomputer consisting of one 4 MB, 25 MHz T800 transputer and eight 1 MB, 20 MHz transputers. The multi-transputer system was mounted as a back-end to a microcomputer and the connection board used was the B008 (which itself contained a T222 transputer not used as a computing device in our experiments). The B008 contains a reconfigurable crossbar switch which provided the ability to modify the processor topology of the transputer network. The occam 2 compiler provided in the IMSD7205 occam 2 Toolset product was used for all compilations.

Two systolic algorithms were selected for implementation. The Weighted Levenshtein Distance (WLD) has been used in symbolic [11] processing while matrix multiplication is heavily used in numerical processing (see for example, [1]). Note that in both cases, their systolic graphs closely resemble that shown in Figure 2.

Since the goal of these experiments was to explore the issues inherent in the ISM, little performance optimization of the code was done. Consequently: (a) in the Diagonal IO, each computation associated with a vertex in the systolic graph was represented by a distinct process, thus, signifi-

cantly increasing the communications overhead, (b) in both parallel IOs, the effective load balancing induced by the locations of the cuts on the systolic graph were not considered, and (c) in the Farming IO, a linear topology using single buffered communication was implemented. Moreover, the compiling/language environment provided some further restrictions including: (a) imposing a (relatively) small number of possible process instantiations for the Diagonal IO, resulting in our restricting several experiments to  $11 \times 11$  systolic graph sizes (inducing poor granularity match with the processor and resulting in little additional speedups), and (b) restricting the generality of communication routers in the Diagonal IO, thereby significantly adding to the communications overhead.

Figure 6 shows the code space considered for the Diagonal IO implementations. The seven selected implementations of the Diagonal IO are shown with the prefix 'cp'. The top two levels represent two factors impacting upon the module structure. Both factors pertain to the process representation of the set of functions associated with each vertex in  $\hat{G}$ . First, the 'Single Process Definition' defines a single process abstraction which represents all the functions whereas the 'Topological Process Definition' defines distinct process abstractions to represent select groups of these functions. The groups are determined based upon the impact of the systolic computation graph topology on the communication. The topological definition results into a smaller and more efficient implementation while increasing the configuration requirements slightly. Second, a similar grouping relating to the process-to-process communication datatypes (referred to as tagged protocols in occam 2) can be considered. The elimination of tagged protocols not only results in possibly smaller code, but also has an effect on the subsequent (and surrounding) program coding style. These effects combined allow for substantial simplification of code. The third level shown in the figure signifies the execution priority. Finally, the bottom level refers to the two different types of available router algorithms: the first come first serve algorithm or normal (N) and a forced round-robin (FRR). Essentially, the FRR algorithm provides a stricter assurance than the N algorithm for deadlock free communications required in certain processing cases.

The correspondence between the code space given in Figure 6 and the generic modules given in Figure 5 is as follows. The single versus topological process definitions impact upon  $M_F^s$  and  $M_C^s$ . Variant versus non-variant communication protocols impact upon all five modules. The priority is

relevant only for  $M_F^s$  and must be specified in  $M_C^s$ . The two router algorithms impact only  $M_R^s$ .

Figure 7 shows the effect of two selected implementation variations from the code space given in Figure 6. The curves labeled ‘Systolic, No Opt’ and ‘Systolic, Part Opt’ refer to the ‘cp1a’ variation whereas the curve labeled ‘Systolic, Full Opt’ refers to the ‘cp7’ variation. The ‘cp1a’ variation was compiled both without (the ‘No Opt’ version) and with (the ‘Part Opt’ version) certain compiler options which perform (mostly) communication optimization. The ‘Full Opt’ implementation also was compiled with these compile time optimization options. Our results strongly suggest that the performance is sensitive to the refined module structures representing that implementation. Moreover, native compiler optimizations must be allowable.

Recall that a partitioning consisting of two diagonal cuts determine a three processor requirement; however, there is no stipulation placed on the location of those cuts. Thus, there is considerable flexibility allowed for load balancing. In most of our experiments, we have relied on intuitive placements of the cuts. In one experiment (see Figure 8), we have briefly considered the effects of load balancing due to the location placement of the cuts on the systolic computation graph. One cut location is held fixed with the second varied. The location labeled ‘Main’ refers to the cut placed just right of the main diagonal, ‘+1 Offset’ refers to the cut placed one additional diagonal away etc. For the three Diagonal IO variations discussed above, the figure shows that a small difference in the execution times result from the different placement of cuts.

The execution time of the three Diagonal IO implementations discussed above along with two Sequential IO implementations (with and without compiler optimizing options) are also given in Figure 7. We have briefly considered the granularity problems inherent in the Diagonal IO; relevant results are shown in Figure 9. In this case, the computation time was artificially increased, and in conjunction with the use of a performance model, we established that the Diagonal IO (without additional modifications to the module structure) can provide marginal speedup. The execution time of the Farming IO along with the Sequential IO are given in Figure 10. Lastly, we have also considered the Farming and Sequential IOs for matrix multiplication. Figure 11 gives these results.

Based upon the above results, for the given size(s) of the WLD algorithm, the ISM would select the Farming IO as the best implementation. However, for the the selected size(s) of the matrix multiplication the ISM would select the Sequential

IO as the best implementation.

## 6 Conclusion

We have proposed an Integrated System Model (ISM) to address the problem of high economic development costs of concurrent programs. The ISM is composed of multi-level modules that abstract both the high-level program design as well as the low-level optimization. The Generic Parallel Program Model (GPPM) is central to the ISM. The main idea of GPPM is that a given systolic algorithm can be translated to a corresponding systolic graph that exhibits the intrinsic parallelism of the algorithm. The GPPM is used to induce a set of generic program structures that are sufficient for generic module definitions. Lastly, we have reported the results of several experiments on the concurrent implementations of the derived modules.

## References

- [1] G.M. Megson, *An Introduction to Systolic Algorithm Design*. New York, USA: Oxford University Press, 1992.
- [2] G. Megson, ed., *Transformational Approaches to Systolic Design*. 2-6 Boundar Row, London SE1 8HN, UK: Chapman & Hall, 1994.
- [3] D.I. Moldovan, *Parallel Processing, From Applications to Systems*. 2929 Campus Drive, Suite 260, San Mateo, CA, USA, 94403: Morgan Kaufmann Publishers Inc., 1993.
- [4] M. Wolfe, *High Performance Compilers for Parallel Computing*. 390 Bridge Parkway, Redwood City, CA, 94065: Addison-Wesley Publishing Company, 1996.
- [5] K.M. Chandy, “Concurrent Program Archetypes,” in *Proc. of the 1994 Scalable Parallel Libraries Conference*, (Mississippi State, Mississippi, USA), pp. 1–9, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, October 1994.
- [6] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing, Cambridge, Mass., USA: The MIT Press, 1989.
- [7] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While, “Parallel programming using skeleton functions,” in *Proc. of PARLE’93 – Parallel Architectures and Languages Europe* (A. Bode, M. Reeve,

and G. Wolf, eds.), Lecture Notes in Computer Science, pp. 146–160, Springer-Verlag, June 1993.

- [8] B. J. d’Auriol and V. C. Bhavsar, “Generic program structures induced by partitions of a systolic computation graph,” in *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS’98* (S. A. Y. Pan and K. Li, eds.), (Las Vegas, Nevada, USA), pp. 396–401, Oct. 1998.
- [9] Brian J. d’Auriol and Virendrakumar C. Bhavsar, “Generic Program Structures Induced by Partitions of a Systolic Computation Graph,” Tech. Rep. 97/04, Department of Computer Science, The University of Manitoba, Winnipeg, Manitoba, Canada, R3T 2N2, March 1997.
- [10] B.J. d’Auriol and V.C. Bhavsar, “Generic Program Representation and Evaluation of Systolic Computations on Multicomputers,” in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’96), Vol. III* (H. Arabnia, ed.), (Sunnyvale, California, USA), pp. 1213–1224, August 1996.
- [11] B.J. d’Auriol, V.C. Bhavsar, L. Goldfarb, “Systolic Array Implementations for Reconfigurable Learning Machines on Transputers,” in *Proc. of Supercomputing Symposium ’91* (V. Bhavsar and U. Gujar, eds.), (Fredericton, N.B., Canada), pp. 105–119, University of New Brunswick Press, Fredericton, N.B., June 1991.

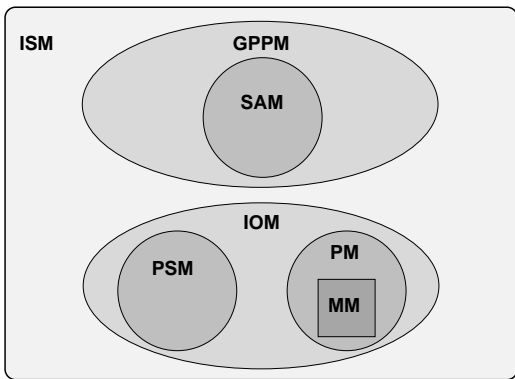


Figure 1: Overview of the Integrated System Model (ISM): GPPM - Generic Parallel Program Model, SAM - Systolic Array Model, IOM - Implementational Object Model, PSM - Program Structure Model, PM - Performance Model, MM - Machine Model.

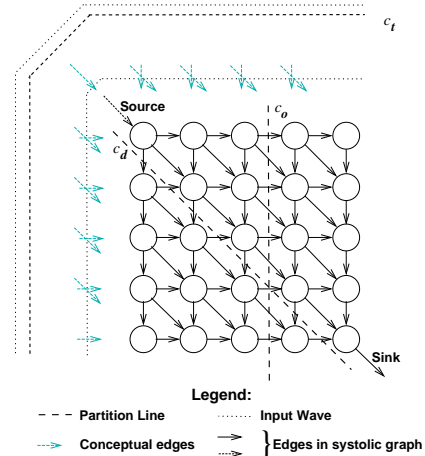


Figure 2: Structure of  $\hat{G}$  for a  $5 \times 5$  systolic graph and examples of three characteristic partitions on  $\hat{G}$ : conceptual edges are shown on the first input wave to illustrate the functionality of ‘systolic’ operation with the single source/sink.

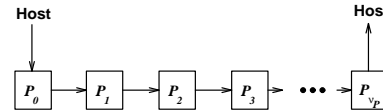


Figure 3: Linear network resulting from applying  $\zeta_o$  on  $\hat{G}$ .

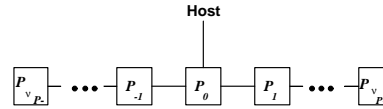


Figure 4: Bidirectional linear network resulting from applying  $\zeta_d$  on  $\hat{G}$ .

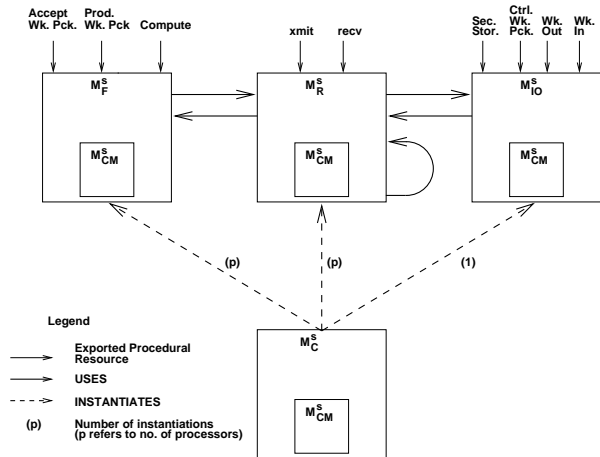


Figure 5: A derived program structure.

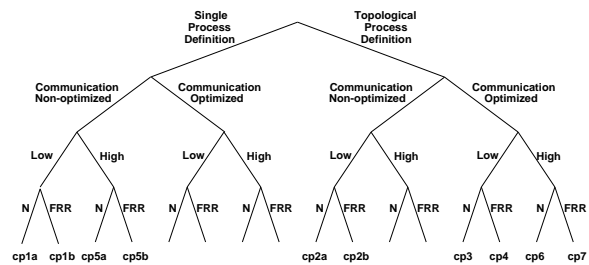


Figure 6: Diagonal Implementation variants: Low/High refers to the execution priority of the computations; 'N' and 'FRR' refer to two different router algorithms.

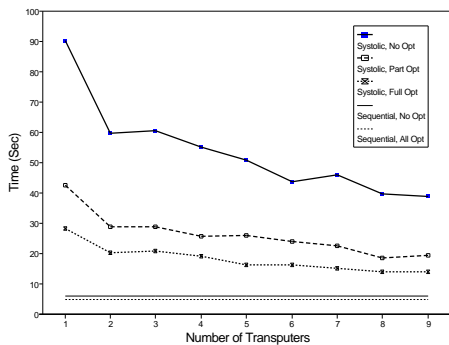


Figure 7: Execution times in seconds for several variations of diagonal and sequential implementations corresponding to 5040 computations: string lengths are 10 symbols each.

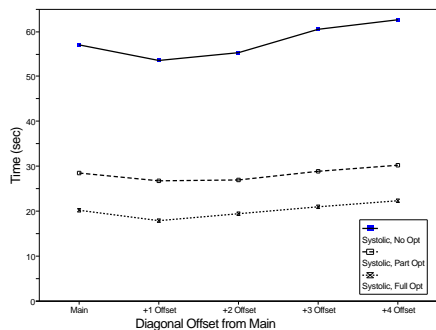


Figure 8: Execution times in seconds for three variations of the Diagonal Implementation for three transputers corresponding to 5040 computations: string lengths are 10 symbols each.

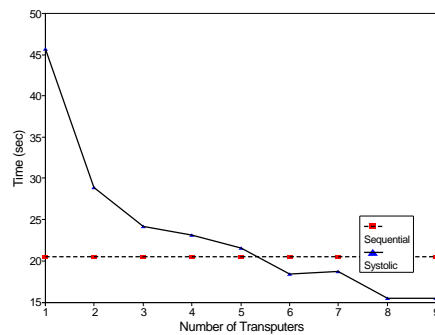


Figure 9: Execution times in seconds for the cp7 variation of the systolic implementation of the modified WLD computation compared with the corresponding sequential implementation

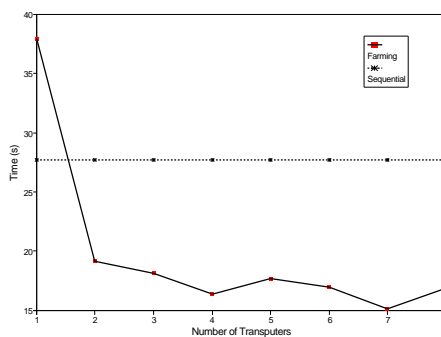


Figure 10: Comparison of execution timings for sequential and farming implementations: string lengths are 50 symbols each.

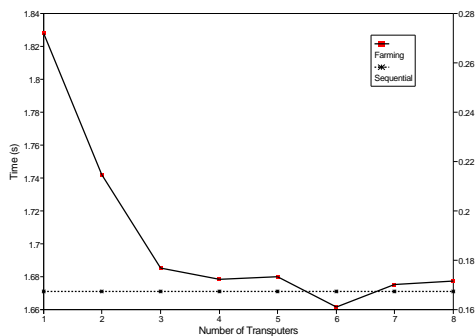


Figure 11: Comparison of execution timings for sequential and farming implementations: matrix size is  $30 \times 30$ .