# Generic Program Representation and Evaluation of Systolic Computations on Multicomputers

Brian J. d'Auriol*

*Department of Computer Science and Engineering,*
*Wright State University,*
*Dayton, Ohio,*
*USA, 45435*

Virendrakumar C. Bhavsar †

*Faculty of Computer Science,*
*University of New Brunswick,*
*Fredericton, New Brunswick,*
*Canada, E3B 5A3*

**Abstract**

An overview of the unified model (proposed earlier by us) for compiling systolic computations for multicomputers is given. Subsequently, we consider the diagonal and farming implementational objects which represent two specific cases of the implementational objects defined in the model. For these two objects, we give prototype codes in the Occam language and give performance models for multi-transputer systems. The prototype codes can be used to generate a source code for implementing a systolic computation on multicomputers and the performance models estimate the execution time of such implementations. Although we concentrate on a transputer environment, the proposed objects are general enough so that accommodation of other multicomputer environments is possible.

## 1 Introduction

We have earlier proposed a unified model for compiling systolic computations for distributed memory multicomputers [1, 2]. This model is generally applicable to a wide range of systolic algorithms, multicomputers and computer languages. Furthermore, the application of the model is expected to eliminate or significantly reduce existing problems with soft-systolic implementations. The indended use of the unified model is in a compiler that accepts a systolic algorithm specification as input and generates an efficient executable object program.

The unified model addresses some of the difficulties in more traditional approaches to advanced parallelizing compilers [1, 2]; in particular, the integration of a high level model to accomodate parallelization and a low level model, termed the implementational object model, to handle machine dependent issues. Under the unified model, multiple possible distinct implementations for a given input algorithm can be determined. Issues pertaining to source code representation, generation and evaluation for a particular machine environment are represented in a generalized structure. The implementational objects are required to be predefined for incorporation into an application of the unified model. It is in this context, in this paper, we restrict the discussion to the derivation of two particular implementational objects, the Diagonal and the Farming Implementational Objects.

This paper is organized as follows. An overview of the unified model is given in the following section. The derivation of the Diagonal and Farming Implementational Objects is given in Sections 3 and 4, respectively. Finally, concluding remarks are given.

*Supported by NSERC doctoral fellowship.
†Partially supported by NSERC grant, OGP0089.

## 2 Overview of the Unified Model

The unified model, $\mathcal{M}$, is defined as the quintuplet $\mathcal{M} = (\mathcal{A}, \mathbf{Z}, \mathbf{\Phi}, \mathbf{M}, \mathbf{\Pi})$, where $\mathcal{A}$ is the given systolic algorithm requiring implementation, $\mathbf{Z}$ is a set of partitioning strategies, $\mathbf{\Phi}$ is the set of implementational objects corresponding to $\mathbf{Z}$, $\mathbf{M}$ is the set of allowable multicomputers and $\mathbf{\Pi}$ is the set of tasks which embody the application of the various stages proposed in the unified model.

An application of the unified model consists of the execution of the tasks represented by $\mathbf{\Pi} = \{\Pi_1, \Pi_2, \Pi_3, \Pi_4, \Pi_5\}$. Here, $\Pi_1$ represents the process of transforming the given systolic computation into an intermediate form. $\Pi_2$ is the process of determining the partitioning strategy as well as the details of the implementational objects. $\Pi_3$ represents the process of partitioning the systolic array intermediate representation from $\Pi_1$, representing it in a source code intermediate form and evaluating the latter in terms of its expected execution time. $\Pi_4$ represents the code generation process. Finally, $\Pi_5$ represents the final stage of compiling the generated code into object code. $\Pi_2$ is formally defined as: $\Pi_2(\zeta, s) \mapsto \Phi_\zeta$, where $\zeta \in \mathbf{Z}$ and $s$ is a two-dimensional wavefront array, with a single source and a single sink, generated by $\Pi_1$. An example of a $4 \times 4$ array structure is given in Figure 1(a) where a circle represents a distinct sub-computation and a directed edge represents the dependence between the sub-computations. Since several candidate partitions $\zeta$ are input to $\Pi_2$, the execution of $\Pi_2$ leads to the corresponding implementational objects denoted by $\Phi_\zeta$.

An implementational object is the three-tuple $\Phi = (\Xi, \mathcal{P}, \Upsilon)$, where $\Xi$ is a set of *prototype codes* together with a set $\mathcal{P}$ of *implementation parameters* and a *performance model* $\Upsilon$. The prototype codes are pre-defined source code templates which essentially are rules for generating source code in a language for which a native compiler is already available. The set of implementation parameters describes the *exact* characteristics of the implementation and is used in the associated performance model. The performance model evaluates the expected execution time for the given systolic algorithm. The performance model is parameterized so that it can provide a good estimate of the execution time while maintaining general applicability.

The application of $\Pi_2$ represents a two-step process. In the first step, candidate partitions are applied to the systolic array generating a set of processor topologies required for each of the candidate partitions. For example, in Figure 1(a), two representative partitions corresponding to the Diagonal Partition, $\zeta_d$, and the Temporal Partition, $\zeta_t$, are illustrated by heavy dashed lines. From our previous investigations [1, 2], the processor topology associated with Diagonal Partitioning is shown in Figure 1(b), while that associated with Temporal Partitioning is shown in Figure 1(c). Note that many other partitions are possible. In the second step, the three components of the implementational object are defined.

In context of this paper, partitions of the form of $\zeta_d$ or $\zeta_t$ would result in the two implementational objects discussed in the following sections of this paper, namely, the Diagonal Implementational Object and the Farming Implementational Object respectively.

## 3 Diagonal Implementational Object

In this section, we discuss the Diagonal Implementational Object, $\Phi_d$, which is based on the processor topology shown in Figure 1(b). The details of the set of prototype codes as well as the performance model together with the implementational parameters for this object are given below. We have applied this object to the Weighted Levenshtein Distance systolic algorithm [2, 3].

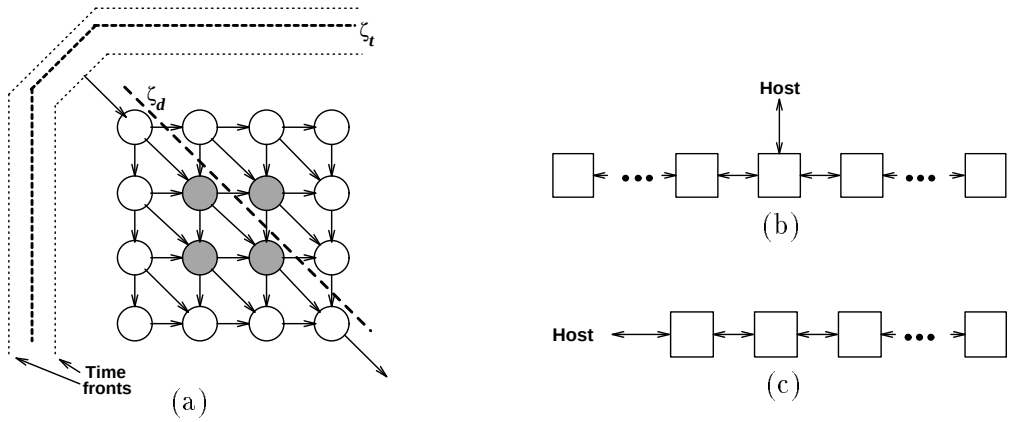



Figure 1: Systolic array representation, partitioning and resultant processor networks: (a) an example of $s$ and two partitions, $\zeta_d$ and $\zeta_t$, (b) processor network due to $\zeta_d$, and (c) processor network due to $\zeta_t$.

### 3.1 Prototype Codes

As previously discussed, each $\xi \in \Xi$ represents a complete program template for a specific high level language. We limit the discussion in this section to the Occam language as intended for transputer implementation since our experiments were performed in this environment [2]. The intent here is to present rules for the generation of the implementation rather than to present the actual methods used to generate the implementation. We further limit our discussion to a straightforward implementation wherein each of the systolic cells is implemented as a distinct process and each communication link as a distinct message passing channel. Our approach then, is to allocate groups (as determined by the partitioning procedure) of systolic cells together with their associated communication channels to physical processors and links of the multicomputer. This representation has been common in the literature (see for example, [4]). We recognize the possibility to amalgamate certain systolic cell processes prior to the allocation for purposes relating to optimization. We believe that the model presented in this paper can accommodate such optimization techniques, however, since this is an optional part of the model, this topic will not be discussed further.

This implementation approach requires program representation of varying communication behavior in certain regions of the array, implemenation of multiplexer and demultiplexer processes for communiation routing between processors and allocation of processes to processor constraints.

There are nine regions of $s$ wherein slight changes in the communication behavior can be observed. For example, communication to and from the four corners, the four edges not including the corners and the center will be different due to the different numbers of incoming and outgoing communication links. We collectively term all regions except the center as the boundary region (see for example the unshaded cells in Figure 1). The significance of this type of topological difference with respect to software implementations of systolic arrays has been reported in the literature (see for example Broomhead et al. in [5]). Furthermore, additional functionality is required by the boundary cell processes so that the inputs and outputs of the systolic array are distributed to appropriate starting locations.

A restriction of the targeted version of Occam is that only a *single* channel can be allocated to an external link. This results in the need to multiplex all channels which are required to be associated with a particular link. A corresponding demultiplexer would be required on the receiving processor.

Lastly, for practical reasons, the processes governing input to and output from the array need be allocated to the same physical processor. This includes user and/or file input or output.

There are four prototype code categories as follows.

1. **Configuration:** Configuration is required in order to instantiate, together with appropriate interconnections, the processes needed for the implementation and to map these processes to specific processors. Depending on the functional requirements of the processors involved, process instantiations include systolic cells, routers and input/output processes.

   A technique termed *systolic array mapping* is proposed as a framework in which to carry out these requirements. This scheme has two parts, a *rigid mapping* which defines a set of indexing abstractions and a *flexible mapping* which is used during process instantiation. An example of flexible mapping is given in Figure 2(a). This example shows the instantiation of cell processes in the center region including the appropriate channel allocations (channels are identified by the prefix `com.` ) and the passing of a unique process identification to each cell process. The rigid mapping establishes the appropriate indexing scheme for the channel arrays by defining appropriate identifiers. Two channels with the same address, differing only in their second dimension, are used to accommodate the multiplexing requirement. It is further necessary to divide cell processes according to their external communication requirements; this is represented in Figure 2(a) as Set A and Set B. While we are satisfied with this allocation scheme in the context of developing the implementational object, we recognize that further improvement may be made for the allocation of channels. Lastly, the instantiation of the routers and input/output process (not shown in the figure) follow this mapping scheme.

2. **Routers:** There are two algorithms which may be used to implement the routers. The first type has a simplier code structure and less constraints on the sequence of possible communications while the second type must be used whenever more than one distinct computation is active at the same time. The problem inherent whenever there are two or more distinct active computations is that the asynchrony of the communications can no longer be guaranteed (since it is possible for a cell performing the *second* computation to request interprocessor communication in such a way that the first computation can not complete an interprocessor communication request — deadlock). The second algorithm forces all interprocessor communication requests from the first computation to be serviced before allowing any from the second computation.

3. **Computation:** Although the computation specification is almost entirely based on the systolic algorithm, the program structure necessary to this implementation can be defined. Figure 2(b) illustrates this structure. There is a defined input section followed by the computation section and lastly, the output section. The channel instantiations in the configuration section match the order of the declarations in the procedure heading, thus ensuring connectivity between the two processes. As previously mentioned, such a structure has been commonly used in the literature.

4. **Communication:** The communication specification is also based on the systolic algorithm. As with the computation specification, a template is defined for the channel declarations. The procedure heading in Figure 2(b) shows that the six communication channels have the type `cell.com.go` . This type is composed of a set of time ordered sequences of primitive datatypes. The details of the `cell.com.go` type are constructed by the application of $\Pi_3$

```
... for each cell process in the center region do
  ... if (the cell process address is in set A) then
    PROCESSOR cell.processor[address]
      computation (address,
                   com.1.input[address+up][0],
                   com.2.input[address+up.left][0],
                   com.3.input[address+left][1],
                   com.1.output[address][0],
                   com.2.output[address][0],
                   com.3.output[address][0])
  ... if (the cell process address is in set B) then
  - similar PROCESSOR statement as above
```

(a)

```
PROC computation(VAL INT address,
    CHAN OF cell.com.go in1, in2, in3, out1, out2, out3)
  ... Declarations specific to the systolic algorithm.
  SEQ
    terminated := FALSE
    WHILE NOT terminated
      SEQ
        PAR
          in1 ? char1; cost1; prev.cell.value1
          in2 ? char2; cost2; prev.cell.value2
          in3 ? char3; cost3; prev.cell.value3
        ... Do cell computation
        PAR
          out1 ! char1; cost1; cell.value
          out2 ! null; 0.0 (REAL32); cell.value
          out3 ! char3; cost3; cell.value
:
```

(b)

Figure 2: Example Occam code: (a) cell process instantiation for flexible systolic mapping, (b) structure of the computation cell.

(see Section 2). Variables need to be declared with the correct datatypes and inserted in the appropriate input and output statements, a role of the prototype codes. The example shown in Figure 2(b) illustrates several variables (e.g. `char1, cost1` and `prev.cell.value1`) used in the communication with datatypes of `BYTE, REAL32` and `REAL32` respectively.

## 3.2 Performance Model

We now turn our attention to the development of a general performance model for use in predicting the expected execution of codes generated by this object. We consider a model for single processor implementations first, and extend this model for multiple processor implementations.

### 3.2.1 Single Processor Model

The total execution time, $T_{tt}$, is made up from two components: computation time $T_{cp}$ and communication time $T_{cm}$ and consequently

$$T_{tt} = T_{cp} + T_{cm}. \tag{1}$$

This will be the minimum time assuming no waiting time.

The computation time is simply the summation of the computation times of all cells,

$$T_{cp} = \sum_{j} k_{cp_j}; \quad 0 \leq j \leq n_1 n_2 - 1 \tag{2}$$

where $k_{cp_j}$ represents the computation time in the $j^{\text{th}}$ cell-process and is in turn based on the machine's computation speed parameter, $n_1$ and $n_2$ denote the rows and columns of the array respectively.

Usually, the computation will be the same within each cell-process and consequently Eq. (2) can be simplified to

$$T_{cp} = n_1 n_2 k_{cp}. \tag{3}$$

The communication time is a function of all communications over the array. Since individual point-to-point communication may vary, the individual communication times across the array must be

summed. There is also excess program code necessary to conduct the communication. Consequently,

$$T_{cm} = \sum_{j} \left[ \left( \sum_{k=1}^{C_j} k_{cm_{j,k}} \right) + k_{ccm_j} \right]; \quad 0 \leq j \leq n_1 n_2 - 1 \tag{4}$$

where $C_j$ is the number of out-going communications from the $j^{\text{th}}$ cell-process, $k_{cm_{j,k}}$ represents the time to effect point-to-point communication for the $k^{\text{th}}$ out-going transmission from the $j^{\text{th}}$ cell-process and $k_{ccm_j}$ represents the overhead of the *program's* communication code for the $j^{\text{th}}$ cell-process. The point-to-point communication constant, $k_{cm}$, represents the machine's communication parameter.

Usually, the program code will be constant for each cell: $k_{ccm_j}$ is constant for all $j$. Thus Eq. (4) can be simplified to

$$T_{cm} = n_1 n_2 k_{ccm} + \sum_{j} \sum_{k=1}^{C_j} k_{cm_{j,k}}; \quad 0 \leq j \leq n_1 n_2 - 1. \tag{5}$$

where $k_{ccm}$ refers to the program communication code constant.

### 3.2.2 Multiple Processor Model

We now extend the model developed in the previous section for multiple processor implementations. The approach is essentially the same as that for the single processor case, however, the addition of the router (as discussed in Section 3.1) impacts upon the communication. We denote this additional communication time by $T_{cb}$. The total execution time for each processor is calculated and the maximum over all processors becomes the expected execution time of the implementation. We make the assumption that *no idle time* exists on the dominating processor where the dominating processor is defined to be the processor which requires the greatest execution time ($P_i$ for which $T_{tt} = T_{tt_i}$ in the following equation). Given $N_P$ processors,

$$T_{tt} = \max_{i} \left( T_{tt_i} \right); \quad 1 \leq i \leq N_P \tag{6}$$

where

$$T_{tt_i} = T_{cp_i} + T_{cm_i} + T_{cb_i}; \quad 1 \leq i \leq N_P. \tag{7}$$

In the single processor model, the number of cells participating in the computation and communication was trivially available. However, the diagonalization of the computation matrix into bands complicates finding which cell-processes are allocated to a particular processor. We denote by $R$ the set of systolic cell processes which have been allocated to a particular processor. Furthermore, we denote by $\underline{R}$ the subset of $R$ such that every element in $\underline{R}$ has at least one communication channel which requires communication to one of the two processors other than the one it is allocated to. Similarly, we denote by $\overline{R}$ the subset of $R$ such that every element in $\overline{R}$ has at least one communication channel which requires communication to the other of the two processors. The definition of $R$ is more precisely given in [1, 2] wherein the systolic array is considered as a graph embedded into the Cartesian plane thus giving the identification of vertices and edges. In this case, $R$ is defined as a subgraph of $s$, as are $\underline{R}$ and $\overline{R}$ where the latter two subsets are precisely specified [2] in terms of their orientation in relation to the systolic array graph. (Refer to Example 3.1 and Figure 3 for an illustration of $\underline{R}$ and $\overline{R}$.)

Eq. (2) now becomes

$$T_{cp_i} = \sum_j k_{cp_j}; \quad j \in R_i;\ 1 \le i \le N_P \tag{8}$$

where $R_i$ is bounded by two neighboring partitions. Usually, the computation will be the same within each cell-process and consequently Eq. (8) can be simplified to

$$T_{cp_i} = ||R_i|| k_{cp}; \quad 1 \le i \le N_P. \tag{9}$$

We restrict the communication component, $T_{cm_i}$, to represent all internal communication on the $i^{\text{th}}$ processor. Consequently, Eq. (4) becomes

$$T_{cm_i} = \sum_j \left[ \left( \sum_{k=1}^{C_j^i} k_{cm_{j,k}} \right) + k_{ccm_j} \right]; \quad j \in R_i; 1 \le i \le N_P \tag{10}$$

where $C_j^i$ denotes the number of *internal* out-going transmissions from the $j^{\text{th}}$ cell process. Eq. (10) can be simplified in similar manner as that of Eq. (5),

$$T_{cm_i} = ||R|| k_{ccm} + \sum_j \sum_{k=1}^{C_j^i} k_{cm_{j,k}}; \quad j \in R_i;\ 1 \le i \le N_P. \tag{11}$$

Lastly, the additional *off-processor* communication must be taken into account. Similar to the internal communication, there are two components: the communication time between two processors (which also includes any additional communication between routers) and the program code overhead with respect to any routers in use. We denote $T_{cb_i}$ to be the off-processor communication for the $i^{\text{th}}$ processor and note that $T_{cb_i}$ depends upon both internal (the $i$ superscripts) and external (the $e$ superscripts) communication:

$$T_{cb_i} = \sum_j \sum_{k=1}^{C_j^e} (k^i_{cm_{j,k}} + k^e_{cm_{j,k}}) + \sum_l \sum_{k=1}^{C_l^e} k^i_{cm_{l,k}} + \left\{ \begin{array}{ll} 2k_{cr}, & \text{if } |\underline{R}'_i| = 0 \text{ or } |\overline{R}'_i| = 0 \\ 4k_{cr}, & \text{otherwise} \end{array} \right\} \tag{12}$$

where $k_{cr}$ is the router program code execution time (where the transmitter and receiver are assumed to have equal time), $C_j^e$ is the number of external communications from the $j^{\text{th}}$ cell, $j \in \underline{R}_i \cup \overline{R}_i$, $l \in \underline{R}'_i \cup \overline{R}'_i$ and $1 \le i \le N_P$. The relations $\underline{R}'$ and $\overline{R}'$ have similar definitions as their counterparts, $\underline{R}$ and $\overline{R}$ respectively, but are for those cell-processes which are allocated on the neighboring processors which communicate to the $i^{\text{th}}$ processor.

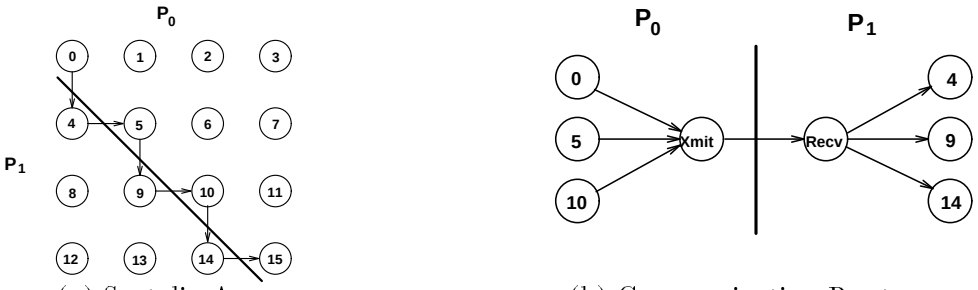
**Example 3.1:** We give an example of these relations as well as the interpretation of equation Eq. (12) for the partition diagrammed in Figure 3(a). Let $P_0$ be the central processor (which is also the dominant processor in this case). Then, $R = \{0,1,2,3,5,6,7,10,11,15\}$, $\underline{R}' = \{4,9,14\}$, $\overline{R} = \{3\}$, $\underline{R} = \{0,5,10,15\}$, and $\overline{R}' = \{\}$.

Consider the communication from processor $P_0$ to processor $P_1$. Cells $0,5$ and $10$ need to communicate with cells $4,9$ and $14$ respectively through the single router channel. This situation is illustrated in Figure 3(b) where, for example, the communication from cell 0 moves through an internal channel to the transmitting router, through an external channel to the receiving router and

finally, through an internal channel to the receiving cell, cell 4. In this case, there is one internal, one external and one router component for processor $P_0$. Additionally, there is also one internal and one router component for the reverse communication (that from cell 4 to cell 5). With reference to Eq. (12), the inner summation in the first term represents the internal and external communication resulting in communication from one processor to another. In this case, the summation can be eliminated since there is only one external communication from each of the first three cells (cells $0, 5$ and 10) and zero from the fourth cell (cell 15) in $\underline{R}$. Similarly, the inner summation in the second term represents the internal communication in the reverse direction and, as before, can be eliminated in this case. Since $|\overline{R}'| = 0$, the contribution of the two router code execution times complete Eq. (12). For this example, Eq. (12) reduces to

$$\begin{aligned} T_{cb_0} &= k^i_{cm_0} + k^e_{cm_0} + k^i_{cm_5} + k^e_{cm_5} + k^i_{cm_{10}} + k^e_{cm_{10}} + \\ & \quad k^i_{cm_4} + k^i_{cm_9} + k^i_{cm_{14}} + 2k_{cr}. \end{aligned} \tag{13}$$

With the added assumption that the same communication structure is used throughout, Eq. (13) can be further simplified to $T_{cb_0} = 6k^i_{cm} + 3k^e_{cm} + 2k_{cr}$.




(a) Systolic Array (b) Communication Routers

Figure 3: Communications across cell boundaries in $P_0$ and $P_1$ processors.

□

### 3.3 Parameter Set

There are five parameters which pertain to how the high level language program is constructed. Discussions of the various techniques and algorithms used in the prototype codes has been given in Section 3.1. We summarize these coding techniques in the context of defining the associated parameters as follows. Firstly, the code for cell-processes may be generated in either one of two modes which reflect the impact of topological variation upon program structure. Secondly, communication specific optimization may be applied to the generated code. Thirdly, the execution of the routers may be in either low or high priority. Fourthly, either of the two router algorithms may be chosen. Lastly, the level of native compiler optimization options needs to be taken into account. The availability of the various allowable options is a function of the prototype codes. However, the modeling of the impact upon the performance model is an issue of the parameter set.

None of the above five techniques are directly incorporated into the performance model described in Section 3.2. However and obviously, the choices regarding these techniques influence the prediction of the execution time. We consequently choose a particular set of available techniques prior to invoking the performance model. We note this is consistent with the definition of $\Upsilon$ given in [1, 2] where the set $p_f$ represents the choice of these five techniques.

The four parameters of the performance model presented in Section 3.2, namely $k_{cp}, k_{cm}, k_{ccm}$ and $k_{cr}$, represent the *contribution* of the fully integrated prototype codes with the algorithm requirements when they are to be executed on a target machine. These constants are derivable in the sense that their values may be determined by measurement and prediction. A determination of these constants for a particular application can be found in [2].

There are four corresponding parameters which describe only the contribution of the program code towards the generated source code program. The computation template is represented by $H_{cp}$, the communication template by $H_{cm}$ the communication code by $H_{ccm}$ and the router code by $H_{cr}$.

# 4 Farming Implementational Object

The temporal partitioning results in distinct computations which can be independently executed by using a processor farm. Since a processor farm is quite independent of a processor topology, we will consider the processor topology in Figure 1(c). The details of the prototype codes and the performance model, together with the implementational parameters for this object are given below. We have applied this object to a matrix times matrix systolic algorithm [2].

## 4.1 Prototype Code

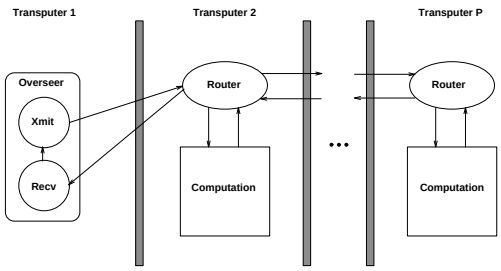
The approach is similar to that in Section 3.1 where we restrict our discussion of the prototype code for the farming implementation to the Occam language as intended for transputer implementation and present rules for the generation of the implementation.

In this case, a computation is a fixed unit of work which is encoded as a work packet and is distributed to a worker process. Consequently, since the definition of the computation does not depend upon the geometry of the systolic array, the software representation requirements are greatly simplified over those relating to diagonal partitioning.

There are four prototype code categories as follows.

1. **Configuration:** A single process is instantiated on the single master processor. This process is responsible for the user input/output or file input/output mechanisms as well as the computation loading mechanism which controls the allocation of work packets to the worker processes. For each of the remaining processors, termed the workers, a computation and a router process is instantiated. Figure 4(a) shows a schematic diagram of this allocation method.

   The configuration code required for this implementation is shown in Figure 4(b) and represents a straightforward encoding of the specifications shown in Figure 4(a). In the `CONFIG` section, the single process `over.seer` is allocated to the master processor. The router process `sl.slave` and the computation process, `compute,` are allocated in pairs to each worker processor in the network. The channel variables `forward` and `backward,` whose datatypes are specified in the include files, directly correspond with the horizontal arrows in Figure 4(a) (including those arrows to and from the overseer) while the channel variables `comp.load` and `comp.unload` correspond with the vertical arrows (not including the one in the overseer) in Figure 4(a). Finally, the correct attachments of these channel variables to the appropriate process are made though the use of appropriate array indexing when passed to the processes upon process instantiation (see Figure 4(b)).




(a)

```
... #INCLUDE files
... channel declarations
... #USE files
CONFIG
  PAR
    PROCESSOR processor[0]
      over.seer(forward[0], backward[0], fs, ts,
                [router.command FROM 0 FOR 4])
    PAR i = 1 FOR no.processor-1
      PROCESSOR processor[i]
        sl.slave(forward[i-1], forward[i], backward[i], backward[i-1],
                 comp.load[i], comp.unload[i])
    PAR k = 1 FOR no.processor-1
      PROCESSOR processor[k]
        compute(k, comp.load[k], comp.unload[k])
:
```

(b)

Figure 4: Farming implementational object: (a) schematic diagram of process allocations and interconnections, (b) the configuration code.

2. **Input/Output Controls:** We describe only one particular implementation mechanism to control the allocation of work packets to worker processes. We note that other variations are possible. The transmitter and receiver act as two concurrent processes, that is, the blocks in Figure 4(a) identified as `Xmit` and `Recv` are defined in parallel. The transmitter packages the data required for communication as a work packet and transmits this packet to the first router. As many packets are transmitted as there are available worker processors. A new work packet is transmitted whenever the receiver indicates to the transmitter that a result of a computation has been received (i.e. a processor is now free).

3. **Routers:** The functionality of the routers is simpler than their counterparts in the diagonal implementation. The routers shunt the work packet (received along one of the three input channels) to the computation module if no computation is currently being performed, otherwise the work packet is passed on to the next worker processor. The result packet from the current processor's computation process is given as input to the router which then outputs the packet in the direction of the master. Result packets may also be input from other routers in which case it is merely passed onwards. Since the router is implemented in two parts defined as operating concurrently, there is the added necessity of one section communicating with the other (via an internal channel) in order to correctly coordinate the activity state of the processor.

4. **Computation:** The computation module has the simplest structure of all prototype code specifications and is entirely dependent on the analysis of the algorithm's systolic array representation. The input to and output from this module match that given in the input/output specification.

### 4.2 Performance Model

We now consider a performance model for linear chained farming. We begin by considering the computation time only, while ignoring the effects of communication. For a given network of worker processors, the total time to complete all $N$ computations is given by the maximum computation time over all worker processors, denoted as $P^w$. Assuming that the processor farm evenly distributes the workload, the total computation time, $T_{cp}$, can be expressed as

$$T_{cp} = \frac{N \times \mathcal{I}_f}{P^w}; \qquad N \geq P^w \tag{14}$$

where $\mathcal{I}_f$ represents the amount of time required for a single computation. For a single worker network, the total time is equal to the number of computations multiplied by the amount of time for a computation. This is consistent with standard sequential processing. As more workers are added to the network, a decrease proportional to $1/P^w$ is expected. When the number of computations is equal to the number of workers, a single computation occurs on every worker and the time required is for one computation. For cases where $N \leq P^w$, all computations can be performed concurrently, consequently, the computation time is equal to $\mathcal{I}_f$. Note that Eq. (14) is a monotonically decreasing function.

The communication overhead increases with $P^w$ since, as more workers are added to the network, more computations can be spawned, and consequently the communication traffic increases. This increase is unevenly distributed, that is, as more workers are added, a greater *density* of the communication occurs between the master and first worker; somewhat less dense is the traffic between the first two workers. This trend continues for each pair of workers in the linear chain. A function such as $(1 - 1/x)$ may be used to model the communication time. In this case, the communication time, $T_{cm}$, can be expressed as

$$T_{cm} \quad = \quad k_g(1 - \frac{1}{P^w}) + \beta \tag{15}$$

where $k_g$ refers to a growth constant and $\beta$ represents the start-up delay required for the implementation.

The total execution time is given by, $T_{tt} = T_{cp} + T_{cm}$.

We have found it interesting to note that a performance model based on an exponential decay curve has been also found to nicely fit the data. Consider the function

$$T_{tt} \quad = \quad k_i \times e^{-\alpha(P^w - 1)} + \beta' \tag{16}$$

where $\beta'$ represents a constant whose value depends upon the computation being performed and $k_i$ represents a scaling factor. The value of $\alpha$ is computed from the following equation (note that Eq. (17) is easily derived from Eq. (16))

$$\alpha \quad = \quad \frac{ln(T_{tt} - \beta') - ln(k_i)}{P^w - 1}; \qquad P^w \neq 1. \tag{17}$$

We report further details of this alternative in [2].

### 4.3 Implementational Parameters

The relative simplicity of the performance models developed in Section 4.2 results in fewer necessary parameters. These parameters represent more abstract quantities than that for the diagonal implementation.

The number of computations, $N$, used in Eq. (14) is obtained from Theorem 2 [1, 2] and is essentially a quantity which is derived at the time the temporal partition is applied to the input systolic algorithm. The sequential time, $\mathcal{I}_f$, is obtainable from the performance model associated with the sequential implementation (not discussed in this paper). These parameters are themselves dependent on details pertaining to both machine and algorithm, consequently Eq. (14) has general applicability.

The two parameters in Eq. (15), $\beta$ and $k_g$ represent the communication behavior of the implementation. They are dependent on the size of the communication packets and the distribution of work packets to the worker processors respectively.

## 5 Conclusion

We have presented two particular generic program representation and evaluation models, namely Diagonal and Farming, for implementing systolic computations on multicomputers. Details of source code, performance modeling and parameterization for implementations on transputers for each have been presented. We have also presented an overview of the unified model (proposed in [1, 2]) wherein we have discussed the structure of the implementational object abstraction and have illustrated the partitioning strategies which lead to the implementational objects discussed in this paper.

The execution of the performance model associated with an implementational object provides estimates of the execution time that generated code, based on the associated prototype codes, would require. Where several candidate implementations are possible (as determined by an application of the unified model), the execution time estimates provided by the performance models are used to determine the most efficient implementation.

Further details of the unified model and the application of implementational objects appear in [1, 2, 3] and an overview of a compilation strategy can be found in [6]. A discussion of a tool to predict communication times on a T800 transputer based multicomputer (which we have used to provide values for some of the constants in the diagonal performance model) appears in [7].

## References


[1] B.J. d'Auriol and V.C. Bhavsar, "A Unified Approach for Implementing Systolic Computations on Distributed Memory Multicomputers," Tech. Rep. WSU-CS-95-02, Department of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435, USA, December 1995. Submitted to the Journal of Parallel and Distributed Computing, December, 1995.

[2] B.J. d'Auriol, *A Unified Model for Compiling Systolic Computations for Distributed Memory Multicomputers*. PhD thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, N.B. Canada, June 1995.

[3] B.J. d'Auriol, V.C. Bhavsar, L. Goldfarb, "Systolic Array Implementations for Reconfigurable Learning Machines on Transputers," *Proc. of Supercomputing Symposium '91*, Fredericton, N.B., Canada, June, 3-5, 1991, V. Bhavsar and U. Gujar, (eds.), University of New Brunswick Press, Fredericton, N.B., pp. 105–119, June 1991.

[4] G.M. Megson, *An Introduction to Systolic Algorithm Design*. New York, USA: Oxford University Press, 1992.

[5] D.S Broomhead, J.G. Harp, J.G.McWhirter, K.J. Palmer and J.G.B. Roberts, "A Practical Comparison of the Systolic and Wavefront Array Processing Architectures," *ICASSP 85, IEEE International Conference on Acoustics Speach and Signal Processing*, Tampa, Fl., March, 26-29, 1985, V. 1, (ed.), pp. 296–299, March 1985.

[6] B.J. d'Auriol and V.C.Bhavsar, "Systolic and Wavefront Array Algorithms on Distributed Memory, Multiprocessor Computers," *Proc. of Supercomputing Symposium '93-High Performance Computing: New Horizons (SS93)*, Calgary, Alberta, Canada, June, 6-9, 1993, L. Bauwens, (ed.), University of Calgary, Calgary, Alberta, pp. 47–54, June 1993.

[7] B.J. d'Auriol and V.C. Bhavsar, "COMMAN — A Communication Analyzer for Occam 2," *Transputer Communications*, Vol. 3, April 1996. in press.