WARP EXPERIENCE: WE CAN MAP COMPUTATIONS ONTO A PARALLEL COMPUTER EFFICIENTLY

H. T. Kung

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
U. S. A.

Abstract

Warp is a programmable, systolic array computer developed by Carnegie Mellon and produced by GE. A 10-cell Warp machine can perform 100 million floating-point operations per second (10 MFLOPS). A variety of applications have been mapped onto Warp. The experience has been that the mapping is not a real problem; in fact, usually near-optimal mapping is relatively easy to obtain, and the actual implementation of the mapping on the machine can often be automated. This paper explains why this is the case by examining some computational models which are frequently used on Warp. Carnegie Mellon and Intel are jointly developing a VLSI version of Warp, called *i*Warp. It is expected that many applications can be efficiently mapped onto low-cost *i*Warp arrays to achieve an effective computation bandwidth of about one GigaFLOPS.

1. Introduction

Many parallel computers are being used in a variety of applications today. Shared memory parallel computers include MIMD machines such as Alliant, Encore, Sequent, and Cray X-MP. Distributed memory computers include MIMD machines such as Hypercube and Transputer, and SIMD machines such as Connection Machine and DAP. Many more parallel machines of enhanced capabilities are under development. A happy experience shared by many users is that it has been relatively easy to map applications onto parallel computers.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

In 1984-87 Carnegie Mellon developed a programmable systolic array machine called Warp, that has a one-dimensional (1D) array of 10 or more processing elements [1]. The machine is currently produced and marketed by General Electric Company. Anticipating the future need for integrated Warp systems, Carnegie Mellon and Intel Corporation have been developing a VLSI Warp chip, called the *i*Warp chip. The *i*Warp system will be available in 1989-90.

Warp has achieved high performance in many application areas including low-level vision, signal processing, neural network simulation, and scientific computing. Like applications experience with many other parallel computers, the Warp experience is that mapping applications onto the machine has not been a real problem; usually near-optimal mapping is not difficult to obtain, and the actual implementation of the mapping on a parallel computer can often be automated. This paper explains informally how the mapping is done on Warp.

There are roughly three stages in solving an application problem on a parallel computer:

> Step 1: Application definition (e.g., by mathematical formula)
>
> Step 2: Computation specification (e.g., by program)
>
> Step 3: Computation on the parallel machine

Computational models characterize the inter-processor communication behavior of Step 3. We use computational models to describe important ways in which applications are mapped onto Warp.

Section 2 provides background information on Warp. Five frequently used computational models for Warp are presented in Section 3. These are models corresponding to local computation, domain partition, pipeline, multi-function pipeline and ring. The last section contains some concluding remarks.

2. Overview of Warp

The Warp machine has three components – the Warp array, the interface unit, and the host, as depicted in Figure 1. We describe the machine only briefly here; details are available from a separate paper [1]. The Warp array performs the bulk of the computation. The interface unit handles the input/output between the array and the host. The host sup-

plies data to and receives results from the array, in addition to executing the parts of the application programs that are not mapped onto the Warp array.

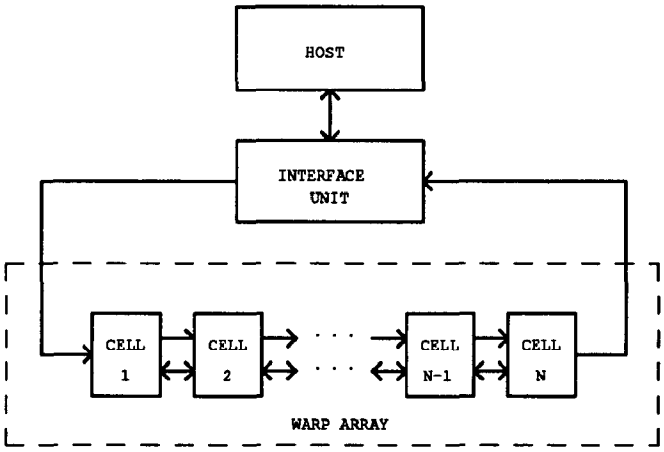

Figure 1. Warp machine overview

The Warp array is a 1D systolic array with identical processing elements called Warp cells. Data flow through the array on two communication channels (X and Y), as shown in Figure 1. The direction of the Y channel is statically configurable at compile time. By configuring the Y channel in the opposite direction from the X channel, a ring interconnection can be formed inside the 1D array. Another way to form a ring is to use the interface unit to connect the first and last cells of the array.

Each Warp cell is implemented as a programmable horizontal micro-engine, with its own microsequencer and program memory. The cell data path includes a 5 MFLOPS floating-point multiplier (Mpy), a 5 MFLOPS floating-point adder (Add), a local memory, and two data input queues for the X and Y channels. All these components are connected through a crossbar. An output port of the crossbar can receive the value of any input port in each cycle. Via the crossbar the floating-point units can directly access data at the front of any input queue, and insert computed results at the end of any input queue of the next cell. Data at the front of any input queue can also be sent directly to the next cell. A (much) simplified description of the Warp cell data path is given in Figure 2.

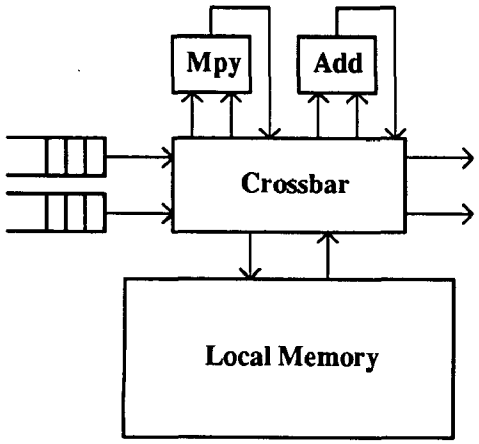

Figure 2. Warp cell data path (much simplied)

A feature that distinguishes a Warp cell from many other processors of similar computation power is its high I/O bandwidth – an important characteristic for systolic arrays. Each Warp cell can transfer up to 20 million words (80 Mbytes) to and from its neighboring cells per second. This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighboring cells and support fine-grain parallelism on the Warp array.

The host consists of a Sun-3 workstation that serves as the master controller of the Warp machine, and a VME-based multi-processor "external host", so named because it is *external* to the workstation. The workstation provides a UNIX environment for running application programs. The external host controls the peripherals and contains a large amount of memory for storing data to be processed by the Warp array. Its dedicated processors transfer data to and from the Warp array and perform operations on the data, with low operating system overhead.

Warp programs are written in a high level Pascal-like language called W2, which is supported by an optimizing compiler [5, 13]. To the application programmer, Warp is a 1D array or a ring of simple sequential processors, communicating asynchronously. Based on the user's program for this abstract array or ring, the compiler generates code for the host, interface unit and Warp array automatically. W2 programs are developed in a Lisp-based programming environment supporting interactive program development and debugging. A C or Lisp program can call a W2 program from any UNIX computer on the local area network.

Carnegie Mellon and Intel are jointly developing a large VLSI chip, called the *i*Warp chip, to implement an integrated version of the Warp cell. The *i*Warp chip is a programmable processor capable of delivering at least 20 or 10 MFLOPS for single or double precision floating-point computations, respectively. In addition, it has on-chip, built-in routing hardware for message passing. Thus the chip is both a computation and a communication engine. This chip together with a local memory form the *i*Warp cell, as depicted in Figure 3. The *i*Warp cell is a powerful building-block cell for a variety of processor arrays, including 1D and 2D arrays.

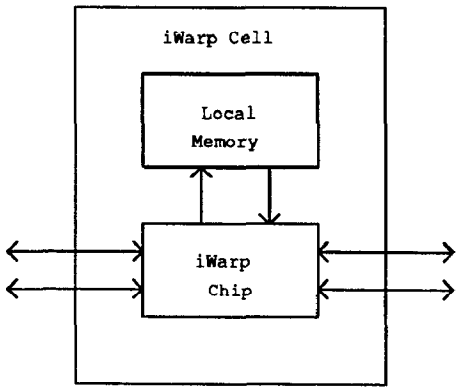

Figure 3. *i*Warp cell consisting of *i*Warp chip and local memory

3. Computational Models for Warp

We will describe the following computational models:

1. local computation;
2. domain partition;
3. pipeline;
4. multi-function pipeline; and
5. ring.

These models correspond to different ways in which cells interchange their intermediate results during computation. Under each model there may also be different ways in handling inputting and outputting for the processor array (see discussions below concerning the local computation model). Therefore the computational models are based on the communication behavior for intermediate results rather than input and output.

In the diagrams, cells in a 1D processor array are denoted by square boxes, and named as cell 1, cell 2, $\cdots$, cell N from left to right. Solid arrows denote data flows of intermediate results between cells.

3.1. Local Computation Model

The *local computation model* corresponds to the case where cells do not exchange their intermediate results during computation at all. Many computational problems have the property that elements in the output set are computed independently from each other. The use of the local computation model is natural in solving these problems on a parallel computer. In this model each output is computed entirely within a cell, and all the cells compute different outputs simultaneously. The main characteristic is that the entire computation for each output is done *locally* at a cell, i.e., the computation does not depend on intermediate results computed by other cells.

Various methods can be used to take care of the inputting and outputting for each cell. For example, before or during computation, the required input to a cell can be shifted in via the cells to the left, and during or after the computation the output produced by a cell can be shifted out via the cells to the right. This is depicted by Figure 4, where dotted arrows denote the shift-in and shift-out paths for input and output, respectively. To achieve high performance, it is important that the I/O time and computation time can be overlapped as much as possible.

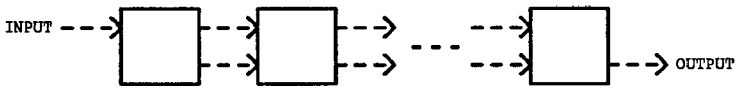

Figure 4. Local computation model, with input and output shifted in and out

Many image processing computations involve transforming an input image to an output image, using a kernel operator defined by, say, a 3×3 window. Figure 5 depicts such a transformation, with which each pixel in the output image depends on a neighborhood of the corresponding pixel in the input image. Clearly, all the pixels in the output image can be computed simultaneously and independently. Therefore the local computation model applies here. The figure illustrates that four cells can work on the four subregions of the output image independently, provided that the input pixels needed by each cell's computation are pre-stored in the cell. Note that cells computing adjacent subregions have overlapped input; the larger is the kernel the larger is the overlap.

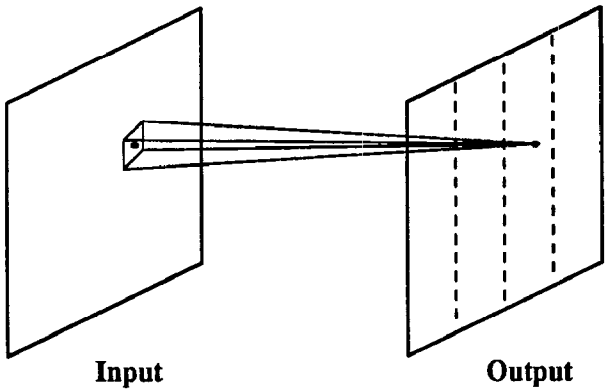

Figure 5. Local computation model for image processing using a kernel operator

As illustrated by the figure, the partitioning of the image processing task for the local computation model is straightforward. All that needs to be done is to partition the output image equally for all the cells. This partitioning has been automated – Carnegie Mellon has developed a compiler called Apply, which can generate W2 programs for image processing computations using a kernel operator, and other computations of similar kind [7].

Apply-generated W2 programs are able to overlap I/O with computation. While computing a row of pixels for the output image, a cell can output a previous row of pixels already computed and input a new row of pixels required for future computations. The Warp array supports this overlapping well, since the array has a high intercell communication bandwidth, and each cell is a horizontal micro-engine capable of performing many computation and I/O operations in each cycle. Because with Apply this overlapping is done automatically, Apply-generated Warp programs are often more efficient than the corresponding hand-generated code.

There is another interesting form of overlapping input with computation for the local computation model. Although all the cells compute different parts of the output set, the cells may share some input. In this case the shared input may be pumped systolically from cell to cell during computation. In the following this is illustrated with a matrix multiplication example.

Given $n \times n$ matrices A and B, we want to compute their product C on a linear processor array of k cells. We assume that k is much less than n, and in the illustration below, $k=4$. We evenly partition columns of B and C as shown in Figure 6 (a). Using the local computation model, cell i will compute the entries of submatrix C_i. As its inputs, cell i needs A and B_i. Therefore input A is shared by all cells. Cell i will first load entries of B_i into its local memory. Then during computation, entries of matrix A will be input to the left-most cell

in the row-major ordering, and shifted to the right from cell to cell, as depicted in Figure 6 (b). Cell i will perform inner products for all pairs of row and column in A and B_i, respectively. (Each entry of A will be input repeatedly as it will be used by each cell multiple times, one for each of the columns of B that the cell has.) Each inner product involves reading in a row of A from one of its input queues and a column of B_i from the cell's local memory, and performing a sequence of multiply-accumulate operations. By shifting in entries of A on-the-fly, each cell does not have to store the entire matrix. This can significantly save memory storage and access time for each cell [11].

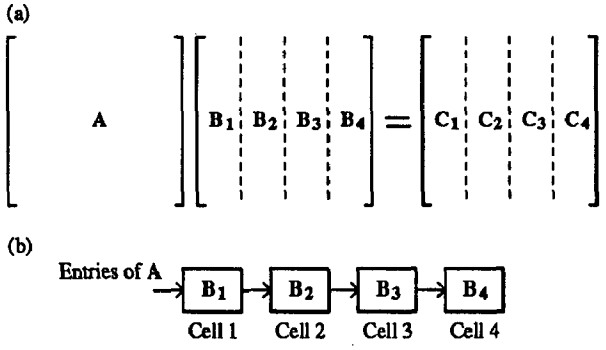

Figure 6. Matrix multiplication: (a) partitioning of matrices B and C, and (b) distribution of the resulting submatrices of B to the cells; entries of A moving to the right during computation

There are many other examples based on the local computation model. They include the discrete cosine transform [2] and the labeled histogram computation [12].

3.2. Domain Partition Model

For some applications the computation depicted in Figure 5 is repeated many times; each time a new output image is computed based on the previous output image. This computational process, called successive relaxation [15, 16], is depicted in Figure 7, where the grids correspond to the images.

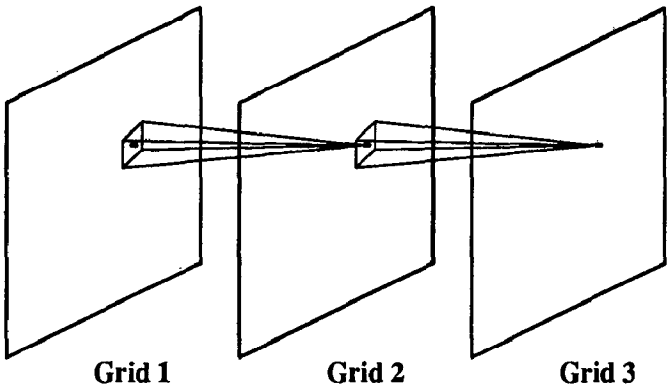

Figure 7. Successive relaxation

The successive relaxation process is often used in scientific computing. Consider, for example, the solution of the following elliptic partial differential equations using successive over-relaxation [18]:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x,y).$$

The system is solved by repeatedly combining the current values of u on a 2D grid using the following recurrence:

$$u'_{i,j}=(1-\omega)\,u_{i,j}+\omega\frac{f_{i,j}+u_{i,j-1}+u_{i,j+1}+u_{i+1,j}+u_{i-1,j}}{4}$$

where ω is a constant parameter. In the recurrence, values associated with location (i,j) of the grid have indices (i,j).

Suppose that the partitioning scheme of Figure 5 is used. Then when computing a new grid, each cell must import from its neighboring cells values computed for the previous grid. The required bi-directional data flows between neighboring cells are shown in Figure 8.

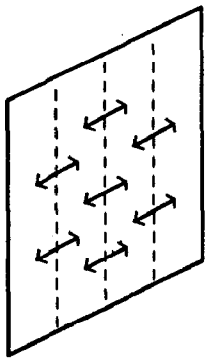

Figure 8. Bi-directional data flows for successive relaxation

With this example, the concept of the *domain partition model* can be easily introduced. The model arises when a problem domain (such as the grid space corresponding to an image, or to a finite difference or finite element modeling) is partitioned so that each cell handles a subdomain. This model differs from the local computation model in that each output is not computed *entirely* by a single cell. That is, once in a while the cell needs to receive intermediate results from its neighboring cells before it can proceed further with its computation. Figure 9 depicts the domain partition model.

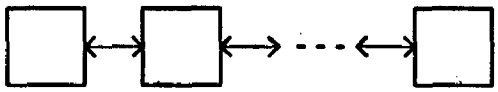

Figure 9. Domain partition model

There are many computations that can be naturally carried out using the domain partition model. Numerical simulations of properties of a physical object, formulated by either differential equations or Monte Carlo methods, can be partitioned along the physical space. A large file can be sorted on a 1D array by using the bi-directional communication to merge sublists sorted by individual cells. The merging can be done with only nearest neighbor communications, in a manner similar to that used in the odd-even transposition sort [3]. Labeling of connected components in an image can be done by using the bi-directional communication to merge labels of subimages computed by individual cells [12].

3.3. Pipeline Model

There is another (elegant) method to carry out the successive relaxation computation depicted in Figure 7 on a 1D array. This method uses pipelining. Instead of the data space, i.e., the grid, we partition along the time axis. That is, successive relaxation steps are done on successive cells. In the row-major ordering, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. Consider for example the successive over-relaxation computation described in Section 3.2. While a cell is performing the k^{th} relaxation step on row i, the preceding and next cells perform the $k-1^{st}$ and $k+1^{st}$ relaxation steps on rows $i+2$ and $i-2$, respectively. Thus, in one pass of the u values through a k-cell processor array, the above recurrence is applied k times. This process is repeated, under control of the external host, until convergence is achieved. In a similar way we can implement many other iterative methods such as Jacobi and Gauss-Seidel methods in a pipelined manner.

In this *pipeline model*, the computation for each output is partitioned into a sequence of identical stages, and cell i is responsible for stage i. A characteristic of this model is that cell $i+1$ uses computed results of cell i, as depicted in Figure 10. Intermediate results move in one direction and final results emerge from the last cell. I/O and computation are automatically overlapped; this is a major advantage of the model. The pipeline model is natural when implementing systolic algorithms where the partial results move from cell to cell and get updated at each cell they pass [9].

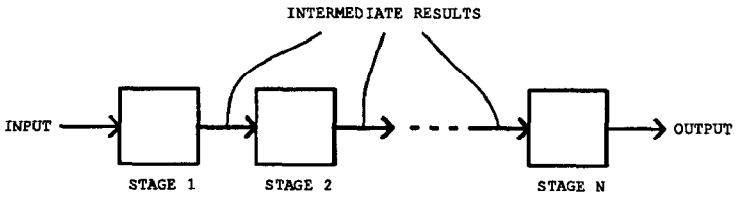

Figure 10. Pipeline model

Under the pipeline model, cell $i+1$ cannot start its operation until cell i completes at least a stage of computation. Thus for this model minimizing the latency between the starting times of adjacent cells is a major concern. This is in contrast with the domain partition model, for which the starting time of a cell does not depend upon any computed results of other cells.

For some computations the pipeline model represents the only efficient parallel implementation. To see such a case, consider a variant of the image processing task depicted in Figure 5. For this variant, in computing the value of each point, the *new* values of its neighbors will be used whenever possible. Suppose that using a 3×3 window, the computation follows the row-major ordering. Then computing the value of each new point uses the new values of the left neighbor and the upper three neighbors, which were computed earlier. A way of using the pipeline model is that cell i computes values of points in row i in the left to right order. Cell i is pre-stored with values of points in rows i and $i+1$. During computation, a copy of each new value cell i computes is sent to cell $i+1$. Note that cell $i+1$ can start its computation as soon as cell i has computed the values of the first two points in row i. We have implemented a version of this pipeline computation on Warp to solve a path planning problem using a dynamic programming technique [4].

3.4. Multi-function Pipeline Model

A single computation may involve a series of subcomputations each executing a different function. If these functions can be chained together on a 1D array, then a one-pass execution of the entire computation will be possible. This is the basic idea of the *multi-function pipeline model* [6]. In this model, the 1D array is a pipeline of several groups, each consisting of a number of cells devoted to a different function. The number of cells in each group is adjusted so that every group will take about the same time, in order to maximize the pipeline throughput.

This model is illustrated in the following example, which is a laser radar simulation implemented on Warp:

> Step 1: For every 1024-point input block, perform a 1024-point complex FFT. Partition each FFT output into 30 overlapped 256-element subsequences.
>
> Step 2: For each of the 30 256-element subsequences, perform the following operations:
>
> (i) multiply each element by a weight, which is a complex number;
>
> (ii) perform a 256-point complex inverse FFT; and
>
> (iii) compute the amplitude of each of the 256 outputs.
>
> Step 3: Threshold the resulting 30×256 image using 3×3 windows.

These steps are implemented with consecutive segments of the Warp array, as depicted in Figure 11.

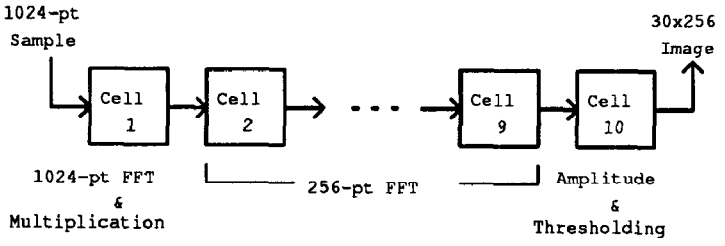

Figure 11. Multi-function pipeline model to implement a radar simulation on Warp

Figure 12 illustrates another possible use of the multi-function pipeline model in implementing the geometry system portion of 3-D computer graphics. The first cell performs the matrix multiplications, the next three cells do clipping, and the last cell does the scaling operation. Three cells are devoted to clipping as it requires more arithmetic operations than either matrix multiplication or scaling [8].

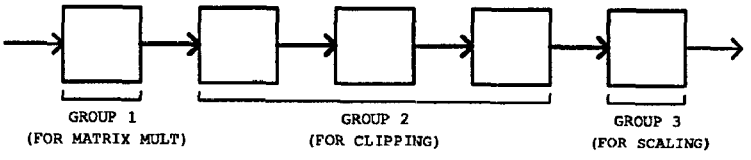

Figure 12. Multi-function pipeline model to implement a geometry system

The multi-function pipeline model is useful when a computation requires a number of small functions, each of which is not large enough to make an effective use of all the cells in a 1D array. Concatenating these functions in a chain offers an opportunity to use more cells effectively. Also, for some computations, it is inherent that one or few cells must perform functions different from the rest. For example, when performing a 2D convolution on a 1D array, some cells need to buffer a row of image and none of the other cells need to do that [10]. For some computations, the first and last cells of a 1D array carry out special functions such as interface with the outside world or preparation of data for the next phase of computation on the array. An example of this is a neural network simulation on Warp, where only the last cell performs weight updates based on weight changes computed by other cells [14].

To support the multi-function model, the processor array must allow *heterogeneous programming*, that is, different programs to be executed at different cells at a given time. Further, the rate of the input to a group may not be compatible to that of the output from the preceding group. Thus some buffering and flow control mechanisms need to be provided between each pair of cells. For the Warp array, all cells can be individually controlled, and dedicated hardware queues capable of performing flow control are available between adjacent cells.

In summary, the multi-function model differs from the pipeline model described early in that cells are now allowed to perform different functions. This flexibility in the usage offers the opportunity of effectively using a large number of cells in a 1D array.

3.5. Ring

A 1D array becomes a ring when the first cell is connected to the last cell. In the *ring model* intermediate results flow on a ring of cells.

An important usage of the ring model is the implementation of a large "logic" array of logical cells, under the pipeline model, with a small "physical" array of physical cells. One implementation is to have each physical cell handle a group of consecutive logical cells as depicted in Figure 14 (a). This will incur a large latency between the starting times of two adjacent physical cells, as the latency will be the sum of all the latencies incurred by those logical cells which are assigned to a physical cell. Another implementation is to use the physical array in multiple passes to simulate the function of the logical array, as depicted in Figure 14 (b). This multiple pass scheme can be implemented with a ring as shown in Figure 14 (c). The ring is formed by using a queue to connect the last physical cell to the first. The queue can store outputs from the last physical cell while the first is still busy in doing its computation for the current pass. This ring scheme incurs the minimum latency between the starting times of two adjacent physical cells.

Another major use of the ring model is in the implementation of broadcasting. Many computational problems involve multiple levels of computation as depicted in Figure 13 (a). Each value in a level depends on all the values computed in the previous level. For example, in the figure to compute b_1 in level 2 we need all the values in level 1, as indicated by the lines connecting b_1 with a_1, a_2, a_3 and a_4. Therefore all the values computed in a level need to be broadcast to all the cells which will be computing values in the next level. An example of such a computational problem is the back propagation neural network simulation [17], for which levels of computation correspond to layers of the neural network.

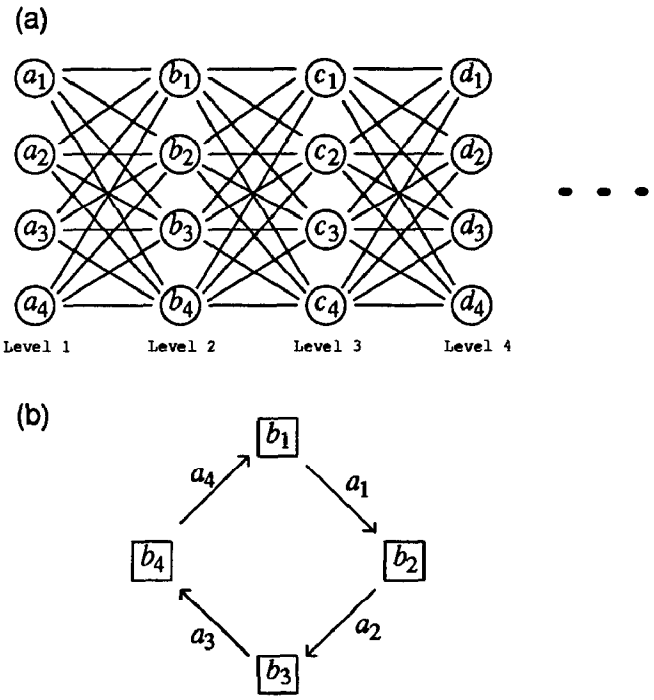

Figure 13. (a) Multi-level computation where results in one level are broadcast to the next level, and (b) using the ring model to implement the broadcasting

The ring structure can implement the broadcasting in a natural way, provided that the computation for each value is commutative and associative so that inputs in the previous level can be combined in any order. Figure 13 (b) illustrates the idea, by considering how values in level 1 can be sent to cells computing values in level 2. Assume that every value in a layer is computed by a separate cell, and for each i the cell which computes a_i will also compute b_i. Then by pumping the a_i's around the ring for a full cycle, as shown in Figure 13 (b), cell i (for every i) will be able to meet all the a_i's so it will have all the inputs to compute b_i. The computation of b_i will occur on-the-fly as each a_i passes by. Therefore computation and I/O are totally overlapped.

4. Concluding Remarks

This paper informally described a number of frequently used computational models for Warp. They represent various ways in which applications are mapped onto Warp. In general the mapping process for a parallel machine such as Warp is quite simple and intuitive; users spend most of their time in debugging programs rather than figuring out efficient mapping schemes. As parallel machines become mature,

better programming tools will be available to remove most of these programming difficulties, which are associated with any new computer system.

Besides describing applications usages, computational models provide a way to classify programming tools for the automatic generation of parallel programs. For example, the Apply programming tool is to generate parallel code for the local computation model. There are several on-going research projects at Carnegie Mellon intended to generate parallel programs for the other computational models such as the pipeline model.

There are a few features in the Warp architecture that contribute to its effectiveness in supporting the computational models described in this paper. These features include the powerful Warp cell, its high degree of programmability through the optimizing compiler, and the simplicity of the linear array interconnection. These ideas have played together well in the Warp architecture. The *i*Warp – the next generation of Warp – will support a larger set of computational models, due to its built-in routing hardware for message passing. Mapping applications onto an *i*Warp array will be even easier.

Acknowledgment

Many of the ideas presented in this paper were inspired by work done under the Warp project at Carnegie Mellon. The author is especially indebted to those members of the project, including F. Bitz, G. Gusciora, H. Ribas, P. S. Tseng, and J. Webb, for their implementation of some of the applications examples discussed in this paper.

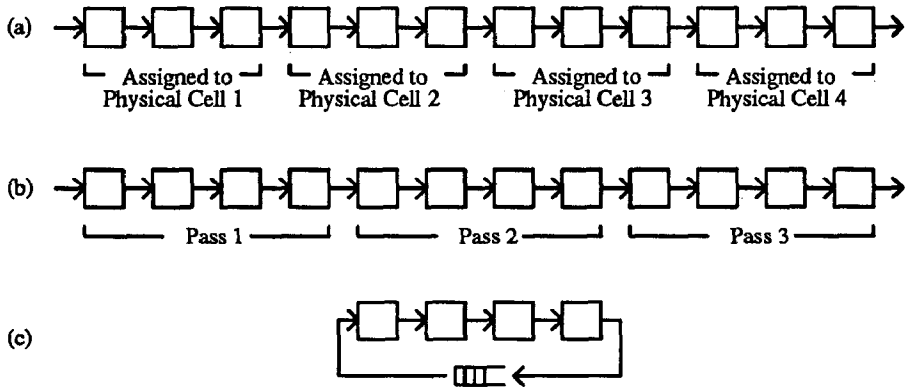

Figure 14. Implementing a large pipeline with a small physical array: (a) each physical cell is assigned to a set of consecutive logical cells, (b) using the physical array in multiple passes, and (c) using a ring to implement the multiple passes on the physical array

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 346-356.

2. Annaratone, M., Arnould, E., Kung, H. T. and Menzilcioglu, O. Using Warp as a Supercomputer in Signal Processing. Proceedings of ICASSP 86, IEEE, 1986, pp. 2895-2898.

3. Baudet, G. and Stevenson, D. " Optimal Sorting Algorithms for Parallel Computers". *IEEE Transactions on Computers C-27*, 1 (January 1978), 84-87.

4. Bitz, F. and kung, H. T. Path planning on the Warp computer: using a linear systolic array in dynamic programming. Proceedings of SPIE Symposium, Vol. 826, Advanced Algorithms and Architectures for Signal Processing II, Society of Photo-Optical Instrumentation Engineers, August, 1987. The final version is to appear in *International Journal of Computer Mathematics* (1988).

5. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.

6. Gross, T., Kung, H.T., Lam, M. and Webb, J. Warp as a Machine for Low-level Vision. Proceedings of 1985 IEEE International Conference on Robotics and Automation, March, 1985, pp. 790-800.

7. Hamey, L. G. C., Webb, J. A., and Wu, I. C. Low-level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, 1987, pp. 185-199. Edited by J. Kowalik.

8. Hsu, F.H., Kung, H.T., Nishizawa, T. and Sussman, A. Architecture of the Link and Interconnection Chip. Proceedings of 1985 Chapel Hill Conference on VLSI, Computer Science Department, The University of North Carolina, May, 1985, pp. 186-195.

9. Kung, H.T. "Why Systolic Architectures?". *Computer Magazine 15*, 1 (Jan. 1982), 37-46.

10. Kung, H.T. Systolic Algorithms for the CMU Warp Processor. Proceedings of the Seventh International Conference on Pattern Recognition, International Association for Pattern Recognition, 1984, pp. 570-577. A revised revion appears as Chapter 3 in *Systolic Signal Processing Systems*, edited by E. E. Swartzlander, Jr., pp. 73-95, New York, Marcel Dekker, 1987.

11. Kung, H. T. Systolic Communication. Proceedings of the 1988 International Conference on Systolic Arrays, May, 1988, pp. .

12. Kung, H. T. and Webb, J. A. "Mapping Image Processing Operations onto a Linear Systolic Machine". *Distributed Computing 1*, 4 (1986), 246-257.

13. Lam, M. S. *A Systolic Array Optimizing Compiler*. Ph.D. Th., Carnegie Mellon University, May 1987.

14. Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T. Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. Submitted to the IEEE Second International Conference on Neural Networks, April, 1988.

15. Rosenfeld, A. Iterative methods in image analysis. Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing, International Association for Pattern Recognition, 1977, pp. 14-18.

16. Rosenfeld, A., Hummel, R. A., and Zucker, S. W. "Scene labelling by relaxation operations". *IEEE Trans. on Systems, Man, and Cybernetics SMC-6* (June 1976), 420-433.

17. Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning Internal Representations by Error Propagation. In Rumelhart, D. E. and McClelland, J. L., Ed., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. I: Foundations*, Bradford Books/MIT Press, Cambridge, MA., 1986, pp. 318-362.

18. Young, D.. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.