# Reliable Systolic Computing through Redundancy*

Kunio Okuda[1], Siang Wun Song[1], and Marcos Tatsuo Yamamoto[1]

Universidade de São Paulo, Brazil,
{kunio,song,mty}@ime.usp.br,
http://www.ime.usp.br/∼song/

**Abstract.** The systolic array paradigm has low communication demand because it does not use costly global communication and each processor communicates with few other processors. It is thus suitable to be used in cluster computing. The systolic approach, however, is vulnerable in a heterogeneous environment where machines perform differently. In this paper we propose a redundant systolic solution with high-availability to deal with this problem. We analyze the overhead that results from the need to coordinate the actions of the redundant processors and show that this overhead is worth the performance improvement it provides.

**Keywords**: cluster computing, heterogeneity, redundancy, high-availability

## 1 Introduction

Since the early eighties, systolic arrays have been proposed to implement numerically intensive applications, e.g. image and signal processing operations such as the discrete Fourier transform, product of matrices, matrix inversion, etc. for VLSI implementation on silicon chips [3]. Given a sequential algorithm specified as nested loops, more formally as a system of uniform recurrence equations, dependence transformation methods [4–6] map the specified computation into a time-processor space domain that can be mapped onto a systolic array.

One nice property of a systolic algorithm is that each processor communicates only with a few other processors. It is thus suitable for implementation on a cluster of computers in which we wish to avoid costly global communication operations. A recent work [2] explores the systolic array paradigm in cluster computing. This approach, however, is not adequate in a heterogeneous environment where the performance of the computers may vary along time. Since the systolic structure is based on tightly-coupled connections, the existence of one single slow processor can compromise and degrade the overall performance. In this paper we propose a solution based on redundancy to deal with this problem. There are many techniques for dependable computing based on check-pointing

* Partially supported by CNPq Proc. No. 55.0094/2005-9 and 30.5218/03-4. The authors wish to thank the anonymous referees for their helpful comments.

and roll-back recovery [7]. The redundant approach is simple but we introduce some overhead to coordinate the actions of the redundant processors. We show that this overhead is worth the performance improvement it provides. The experimental results show that the incurred overhead is small compared to the overall performance we get over the non-redundant solution.

## 2 Matrix multiplication example

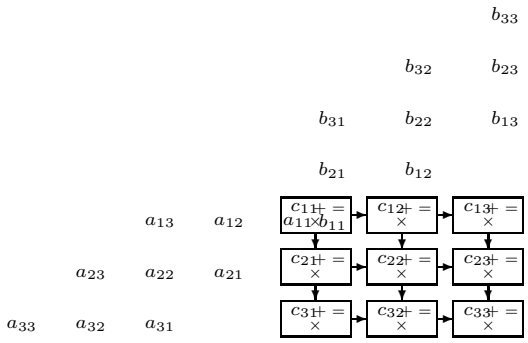


**Fig. 1.** Basic systolic matrix multiplication algorithm.

In [2] we use the systolic array structure to solve two basic problems: matrix product and alignment of two strings. We now use the matrix product example to illustrate the redundancy method. Given two $n \times n$ input matrices $A$ and $B$, we wish to compute matrix $C = AB$. The basic systolic matrix multiplication algorithm is shown in Figure 1. For matrices of size $n \times n$, the number of processors $p$ used is $n^2$. The input elements of $A$ and $B$ enter the systolic array and move across the array while elements of the product $C$ remain in the processors.

To implement this systolic algorithm on a cluster, synchronization can be implemented by using non-blocking sends and blocking receives. However, as observed in [2], the basic systolic algorithm is not suitable for cluster computing because of the fine granularity and the large number of processors required. To make the granularity coarser we consider sub-matrices instead of single elements in the basic algorithm. Assume the number of processors is $P = p \times p$ and assume also $n$ divides $p$. We can view the product of two $n \times n$ matrices as multiplying two $p \times p$ matrices whose elements are $n/p \times n/p$ sub-matrices.

## 3 Use of redundancy

The redundancy approach to deal with heterogeneity is relatively straightforward but nonetheless promising in terms of the results obtained. For this approach to be feasible, we rely on the abundance of computing resources in the cluster. One issue that needs to be addressed is how we employ redundancy. Another issue

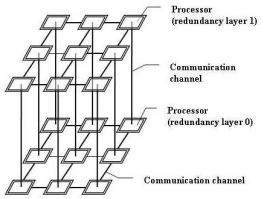


**Fig. 2.** Redundant systolic structure with degree of redundancy = 2.

is that the use of redundancy may incur in overhead and we need to investigate the influence of this overhead on the overall performance.

Assume we want to implement a parallel systolic algorithm that requires $p$ processors. To implement this algorithm, we use $kp$ processors, where $k$ is a small integer. To facilitate the presentation, we use $k = 2$. We first define a few terms. A *redundancy group* is a collection of processors that execute the same computation, with the same input data and produce the same output. The number of processors in each redundancy group is called the *degree of redundancy*. For simplicity, we assume the same degree of redundancy for all the redundancy groups. For each redundancy group of degree $k$, identify each processor of the group by the label $h$, where $0 \leq h < k$. With this, we denote by *redundancy layer* $h$ the collection of processors with label $h$. We use the term *bad processor* to denote a processor that out-stands negatively in terms of available capability to process the given application. Similarly, we denote by *good processor* the processor that out-stands in the group positively in performance.

The proposed redundant structure will be composed by copies of the original systolic array by adding, if necessary, communication channels among the redundancy layers, as shown in Figure 2.

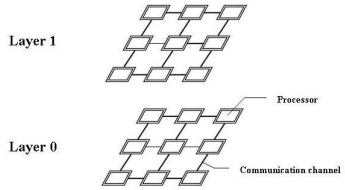


**Fig. 3.** Replicated independent systolic arrays.

A straightforward way to employ redundancy is merely to have $k$ copies of the original systolic structure and perform computation in each redundancy layer independently (see Figure 3). Whichever redundancy layer finishes first would report the desired result. There is practically no overhead incurred. Note, however, the existence of one bad processor in a redundancy layer determines the bad performance of the entire layer.

The above discussion motivates the definition of the *bad performance probability* of the redundant system. Given a redundant systolic structure of degree of redundancy $k$ and total of $kp$ processors in each redundant layer of $p$ processors, and given the existence of $m$ bad processors, the *bad performance probability* is the probability of the redundant system to perform poorly due to the influence of at least one of the $m$ bad processor. To compute this probability, consider $k$ urns each with $p$ balls. Given that a total of $m$ balls are red (bad), it is the probability of all the urns having at least one red ball.

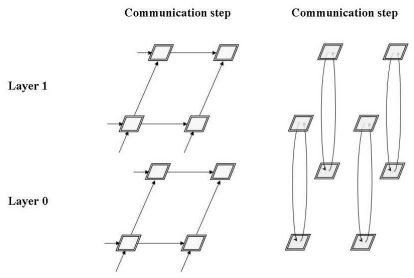


**Fig. 4.** Communication phase in the redundancy layers (left) and computation phase in the redundancy groups (right).

Alternatively, we can employ redundancy in each of the processors of the original systolic array (see Figure 4). In the original systolic algorithm each processor repeats three phases: *data input* from neighbor processors, *computation* of the received data, and *output* of computed data to neighbor processors. On the right of Figure 4 we show that each individual processor of the original systolic algorithm defines a redundant group, in which all its processors execute the same computation of the computation phase in parallel. The running time of the redundancy group to execute a computation is given by the processor that finishes first the given computation. Given $k$ urns each with $p$ balls, and knowing that $m$ balls are red, the bad performance probability is the probability of at least one urn containing all red balls.

During the computation phase, there is a competition among the redundant processors, so that only the result obtained by the fastest processor is considered. We create two processes in each processor: the *computation process* computes the product of the sub-matrices of $A$ and $B$, and the *control process* coordinates the processors of the redundancy group to determine the winner. The two processes share the same memory and mutual exclusion is enforced so that only one process can access shared data at a time. The control process needs to be informed when the computation process has finished the computation. The computation, on the other hand, needs to be informed by the control process when to abort its computation.

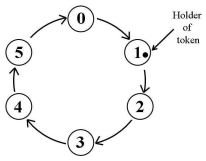


**Fig. 5.** The ring topology used by the token ring algorithm to ensure mutual exclusion.

To guarantee that only one processor is the declared winner within a redundancy group, we use the token ring algorithm [1]. The processors of the group have the ring topology and each processor is identified by an integer *label* from 0 to $k-1$. A *token* circulates from processor to processor in the ring. When the token reaches processor $k-1$, it returns to processor 0 and the cycle repeats. The processor that holds the token at any moment has the priority to enter a critical region and thus can execute the necessary tasks exclusively. If a processor holding the token does not want to enter the critical region, it simply passes the token forward to the next processor in the ring. See Figure 5. The token carries a token value, initially defined to be -1. The processor that finishes its computation and that currently holds the token assigns its label as the new token value and then passes it forward, declaring itself to be the winner. The new token circulates in the ring to signal all the participants to abort their computation.

## 4 Experimental Results

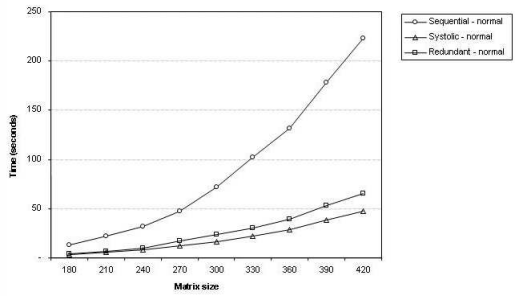


**Fig. 6.** Running times in a homogeneous environment

We ran experiments on a cluster of 16 microcomputers with a Switch 3COM 3300 and Fast Ethernet 100Mbit/s. Each microcomputer consists of a 1.2GHz Athlon Thunderbird processor with 256 KB L2 cache, 768 MB PC133 SDRAM and a 30 GB ATA100 hard disk. The operating system is Debian Linux 2.2.19. We use ANSI C, compiled under version GNU gcc 2.95.2-13, POSIX Threads

package for the local threads and LAM-MPI for the message exchanges. The test consists of running a sequence of 50 problems of matrix products. Figure 6 shows results in a homogeneous environment, with no *slow* machines. The matrix sizes tested were $180 \times 180$ up to $420 \times 420$.

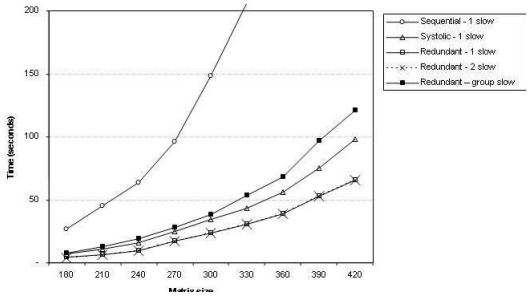


**Fig. 7.** Heterogeneous environment - "redundant - 2 slow": each group has a slow machine, and "redundant - group slow": all the machines of a group are slow.

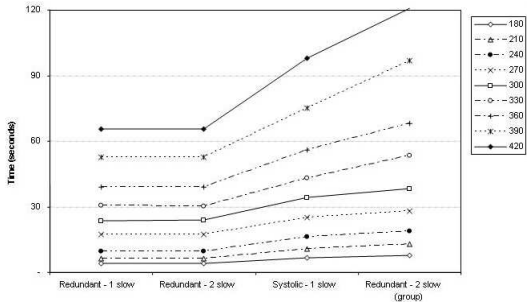


**Fig. 8.** Running times in a heterogeneous environment for different matrix sizes

To simulate a heterogeneous environment, we made one or more machines to act as *slow* machines, by running another process simultaneously. In Figure 7 we assume there is at least one *slow* machine in a redundancy group. Figure 8 shows the same results for several matrix sizes and slow machines with different degrees of *slowness*. Figure 9 shows the running times of the normal systolic algorithm without redundancy and the redundant systolic algorithm, for two matrix sizes and different degrees of slowness of the bad machine.

The experiment shows clearly the benefit of the redundant approach. The most interesting fact we observe in this experiment is that the redundant solution does not depend on the degree of slowness of the bad machine.

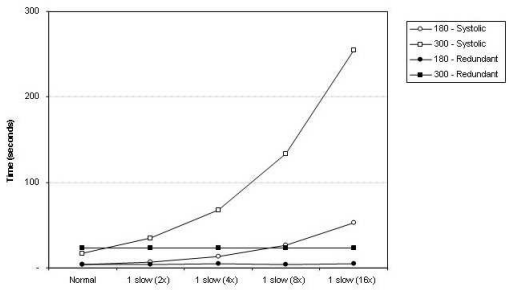


**Fig. 9.** The effect of one slow machine on the performance.

## 5 Conclusion

The systolic array paradigm has less demand on communication because they do not use the global communication primitives. The tightly coupled nature of its processors, however, show the vulnerability to the presence of even one single slow machine in the system. This paper proposes a way to use the abundant computing resources to deal with this problem. The use of redundancy do incur in additional cost, due to the overhead to implement the redundancy control mechanism. We compared the behavior of the sequential algorithm, the systolic algorithm without redundancy, and the redundant systolic algorithm, in homogeneous environment and also in a heterogeneous environment where one or more machines are forced to act as slow machines. Our experiment shows the benefit of the redundant approach. Despite the overhead, the redundant solutions outperform the non-redundant one. We note also that the redundant solution does not depend on the degree of slowness of the bad machine.

## References

1. D. Bird. *Token Ring Network Design.* Addison-Wesley, 1994.
2. U. K. Hayashida, K. Okuda, J. Panneta, and S. W. Song. Generating parallel algorithms for cluster and grid computing. In *The 2005 International Conference on Computational Science - ICCS 2005*, volume 3514 of *Lecture Notes in Computer Science*, pages 509–516. Springer Verlag, 2005.
3. H. T. Kung. Why systolic architectures. *IEEE Transactions on Computers*, 15:37–46, 1982.
4. D. I. Moldovan. *Parallel Processing: from Applications to Systems.* Morgan Kaufmann Publishers, 1993.
5. K. Okuda. Cycle shrinking by dependence reduction. In *Proceedings 2nd International Euro-Par Conference*, volume 1123 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, 1996.
6. P. Quinton and Y. Robert. *Algorithmes et architectures systoliques.* Masson, 1989.
7. M. Treaster. A survey of fault-tolerant and fault-recovery techniques in parallel systems. *ArXiv Computer Science e-prints*, pages 1–11, January 2005.