Generating Parallel Algorithms for Cluster and Grid Computing[⋆]

U. K. Hayashida[1], K. Okuda[1], J. Panetta[2], and S. W. Song[1]

[1] Universidade de São Paulo, Brazil,
{ulisses,kunio,song}@ime.usp.br,
http://www.ime.usp.br/∼song/
[2] Instituto Nacional de Pesquisas Espaciais,
Centro de Previsão de Tempo e Estudos Climáticos, Brazil
panetta@cptec.inpe.br

Abstract. We revisit and use the dependence transformation method to generate parallel algorithms suitable for cluster and grid computing. We illustrate this method in two applications: to obtain a systolic matrix product algorithm, and to compute the alignment score of two strings. The product of two $n \times n$ matrices is viewed as multiplying two $p \times p$ matrices whose elements are $n/p \times n/p$ submatrices. For m such multiplications, using p^2 processors, the proposed parallel solution gives a linear speedup of $\frac{mp^3}{(m+2)p-2}$ or roughly p^2. The alignment problem of two strings of lengths m and n is solved in $O(p)$ communication rounds and $O(mn/p)$ local computing time. We show promising experimental results obtained on a 16-node Beowulf cluster and on an 18-node grid called InteGrade, consisting of desktop computers.

1 Introduction

The abundance of low cost computing resources in clusters and the increasing network bandwidth make it attractive to run parallel applications with the so-called grid-based computing. We attempt to use efficiently the computational resources that are idle to obtain a system of high computing power with the existing machines. Due to the high communication cost in cluster and grid computing, we are interested in designing parallel applications with low demand on communication. To this end, we propose a method to design parallel algorithms with the nice property of each processing having to communicate with only a few others. It is based on a modification of the dependence transformation method that was originally proposed to design systolic arrays for VLSI implementation on silicon chips. Given a sequential algorithm specified as nested loops, or more formally as a system of uniform recurrence equations, the specified computation can be transformed into a time-processor space domain adequate to be implemented on a VLSI chip [2, 3, 7, 6, 8, 10].

⋆ Partially supported by CAPES, CNPq Grant Nos. 55.2028/02-9 and 30.5218/03-4.

Some nice properties of systolic arrays include the regularity on the layout of the processing elements and local communication where each processing element communicates only with a few neighbor processors. These features make it suitable for implementation on a cluster where we wish to avoid costly global communication primitives. We illustrate the proposed method by deriving parallel algorithms for matrix multiplications and for computing the alignment score for string comparison. We then present implementation results on a 16-node Beowulf cluster and also on an 18-node grid, called InteGrade [4], consisting of desktop computers.

2 Dependence Transformation

Given a sequential algorithm expressed as nested loops or, more formally, as a system of uniform recurrence equations [5], the dependence transformation method [3, 10] transforms the computations involved into a time-space representation. We illustrate this method through an example.

Example 1. Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, the matrix product can be expressed by a system of uniform recurrence equations, defined on the domain of all the points (i, j, k) for $0 \le i, j, k < n$.

$$
\begin{aligned}
&0 \le i < n, 0 \le j < n,\ 0 \le k < n,\\
&\quad C(i,j,k) = C(i,j,k-1) + A(i,j-1,k)B(i-1,j,k)\\
&\quad A(i,j,k) = A(i,j-1,k)\\
&\quad B(i,j,k) = B(i-1,j,k)
\end{aligned}
$$

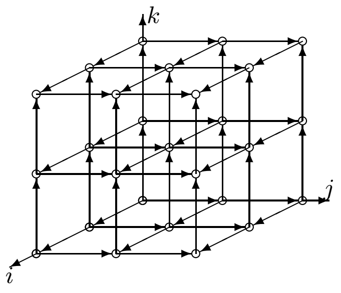


Fig. 1. Dependence graph assuming $n = 3$.

$A(i,-1,k)$ and $B(-1,j,k)$ are the coefficients a_{ik} and b_{kj}, respectively. The values of $C(i,j,-1)$ are assumed to be zero. The output values $C(i,j,n-1)$, at the right side of the equations, give the desired product. To compute $C(i,j,k)$ we use the value $C(i,j,k-1)$ that needs to be computed earlier. We say there is a dependence vector for variable C, denoted by $\theta_c = \begin{pmatrix}0 & 0 & 1\end{pmatrix}^T$. Likewise we have the dependence vectors $\theta_a = \begin{pmatrix}0 & 1 & 0\end{pmatrix}^T$ and $\theta_b = \begin{pmatrix}1 & 0 & 0\end{pmatrix}^T$

Consider a *dependence graph* which has as vertices the points of the domain and directed edges from vertices y to z if z depends on y. See Figure 1 for an example. Each point of the dependence graph represents some elementary computation that needs to be calculated. Through a *time function* τ and a *processor allocation function* α, we can transform the dependences so that the points that do not depend on each other can be computed in parallel.

The time function maps each elementary computation of the dependence graph into a positive integer that represents the time unit it is executed. Consider a domain point $z = \begin{pmatrix} z_0 \; z_1 \; z_2 \end{pmatrix}^T$. The following is a time function: $\tau(z) = \lambda_0 z_0 + \lambda_1 z_1 + \lambda_2 z_2 + \delta$, where λ_i and δ are integers, and satisfying the condition: if z depends on y, then $\tau(z) > \tau(y)$.

Let $\lambda = \begin{pmatrix} \lambda_0 \; \lambda_1 \; \lambda_2 \end{pmatrix}^T$. Then $\tau(z) = \lambda^T z + \delta$. Suppose z depends on y by the dependence vector θ, that is, $z - y = \theta$. Then we have $\tau(z) > \tau(y)$ or, equivalently, $\tau(z) - \tau(y) > 0$. This can be written as $\lambda^T(z - y) > 0$, or $\lambda^T \theta > 0$, or $\lambda^T \theta \geq 1$. Thus for each dependence vector θ_i we have $\lambda^T \theta_i > 0$.

Consider Example 1, we have $\theta_c = \begin{pmatrix} 0 \; 0 \; 1 \end{pmatrix}^T$, $\theta_a = \begin{pmatrix} 0 \; 1 \; 0 \end{pmatrix}^T$, $\theta_b = \begin{pmatrix} 1 \; 0 \; 0 \end{pmatrix}^T$. It can be shown [10] that $\tau(z) = z_0 + z_1 + z_2$ is a time function. That is $\tau(z) = \lambda^T z$ where $\lambda = \begin{pmatrix} 1 \; 1 \; 1 \end{pmatrix}^T$.

The processor allocation function α maps each point z (of a domain of n dimensions) onto a point (of a space $n-1$ dimensions) $\alpha(z)$ where the elementary computation associated to z is performed. Different points of the domain that are computed at the same time instant should be mapped to different processing elements, that is, $\forall z, y$ in the domain, $\alpha(z) = \alpha(y) \Rightarrow \tau(z) \neq \tau(y)$.

Quinton and Robert [10] show how to obtain a processor allocation function α, given the time function λ, by projecting the domain points according to a direction given by a vector u that is not orthogonal to λ. Consider a domain of dimension 3. Let $u = (u_0 \; u_1 \; u_2)^T$ be a non-null vector such that $\lambda^T u \neq 0$. It can be shown [10] that $\alpha(z) = (\alpha_0(z), \alpha_1(z))$ is a processor allocation function where $\alpha_0(z) = u_2 z_0 - u_0 z_2$ and $\alpha_1(z) = u_2 z_1 - u_1 z_2$. In our example, we wish to obtain a processor allocation function α that maps each one of the dependence vectors onto either the null vector or $(0 \quad 1)^T$ or $(0 \quad 1)^T$. The following is a possible processor allocation function.

$$\alpha(z) = \begin{pmatrix} 1 \; 0 \; 0 \\ 0 \; 1 \; 0 \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \end{pmatrix}.$$

Figure 2 shows how this systolic array computes the matrix product of two matrices A and B. Each processing element receives an element of matrix A and an element of matrix B, computes the product of the two elements and adds it to the element of matrix C it stores. This parallel solution requires a total of $3n-2$ computing steps plus the same amount of communication steps. We obtain a speedup in terms of computing steps of $n^3/(3n-2)$ or $n^2/3$ for large n. Notice that we waste some time to fill in and flush out the pipeline. If we need to obtain m matrix products, then the parallel time is improved with the amortization of the costs of pipeline fill-in and flush-out. For m products of two $n \times n$ matrices,

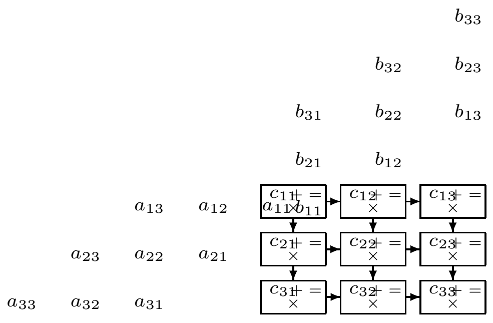


Fig. 2. Computing the product $A \times B$.

we need a total of $(m+1)n-1$ computing steps and communication steps. We now get a speedup in terms of computing steps of $(mn^3/((m+1)n-1)$ or n^2 for large n and m. Since we use n^2 processing elements, we get linear speedup.

Systolic arrays operate in a synchronous and lock-step fashion. Frequent synchronization of all the processor on a cluster or grid is too costly. However, synchronization of the *input, compute, output* steps of the parallel algorithm on the cluster can be achieved by using non-blocking sends and blocking receives.

The fine granularity of systolic algorithms is not suitable for cluster or grid computing due to the disparity between communication and computations speeds. To achieve a coarser grain, we can view each element of a matrix as a block or a submatrix. In other words, instead of each processor receives one single element of each matrix A and matrix B, it receives a submatrix of matrix A and matrix B. It can be shown that we still get linear speedup.

In the global atmospheric circulation model used for weather forecasting computation of Fourier and Legendre Transforms are needed [9]. A global model simulates the behavior of atmospheric fields along the discrete time. Partial differential equations are solved in two spaces: the grid space (here we use the term grid in the Mathematical sense) and the spectral space. The Fourier Transform moves a field between grid and Fourier representations while the Legendre Transform moves a field between Fourier and spectral representations. Fourier Transforms are computed by using the well-known FFT (fast Fourier Transform). Legendre Transforms constitute the most time-consuming part and can be computed as multiple matrix products, handled by the proposed algorithm.

3 A Parallel Algorithm for Alignment of Sequences

In [1] we have proposed a parallel algorithm for efficient sequence alignment. In this section we show that, by using the proposed dependence transformation method, it is straightforward to generate this algorithm. Consider two given strings A and C, where $A = a_1a_2 \dots a_m$ and $C = c_1c_2 \dots c_n$. To align the two

A	a	c	t	t	c	a	−	t	
C	a	t	t	c	−	a	c	g	
Score	1	0	1	0	0	1	0	0	3

A	a	c	t	t	c	a	−	t	
C	a	−	t	t	c	a	c	g	
Score	1	0	1	1	1	1	0	0	5

Fig. 3. Examples of alignment

strings, we insert spaces in the two sequences in such way that they become equal in length. See Figure 3 where each column consists of a symbol of A (or a space) and a symbol of C (or a space). An *alignment* between A and C is a matching of the symbols of A and of C in such way that if we draw lines between the matched symbols, these lines cannot cross each other. Figure 3 shows two simple alignment examples where we assign a score of 1 when the aligned symbols in a column match and 0 otherwise. The alignment on the right has a higher score (5) than that on the left (3).

A more general score assignment for a given string alignment considers insertion/deletion and match/mismatch of symbols, each with the respective scores. For example, a match has a positive score while the other operations negative scores. The similarity score S of the alignment between strings A and C can be computed by the following recurrence equation. For $0 < r \leq m, 0 < s \leq n$,

$$S(r,s) = \max \begin{cases} S[r,s-1]-t \\ S[r-1,s-1]+t & \text{(match) or} \\ S[r-1,s-1]-t & \text{(mismatch)} \\ S[r-1,s]-t \end{cases}$$

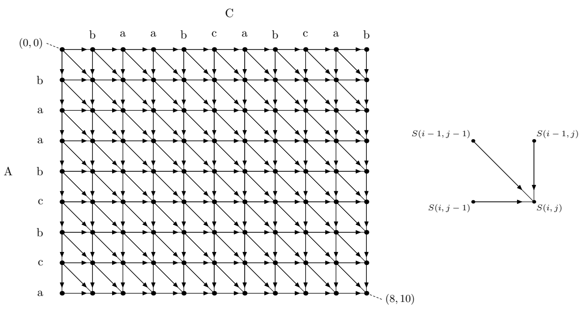
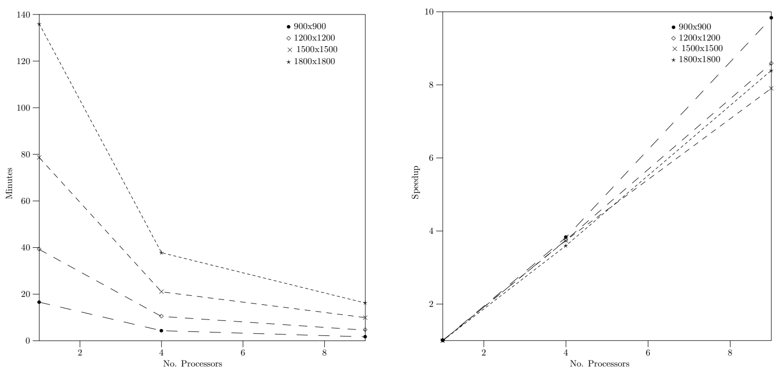


Fig. 4. Grid DAG G for A = baabcbca and C = baabcabcab.

An $l_1 \times l_2$ *grid DAG* (Figure 4) is a directed acyclic graph whose vertices are the $l_1 l_2$ points of an $l_1 \times l_2$ grid, with edges from grid point $G(i,j)$ to the grid points $G(i,j+1)$, $G(i+1,j)$ and $G(i+1,j+1)$. In our terminology, the grid DAG is a dependence graph. Associate an $(m+1) \times (n+1)$ grid dag G with the alignment problem as follows: the $(m+1)(n+1)$ vertices of G are in one-to-one correspondence with the $(m+1)(n+1)$ entries of the S-matrix. It is easy to see that we need to compute a minimum source-sink path in the grid DAG. In Figure 4 the problem is to find the minimum path from (0,0) to (8,10).

Let us now apply the dependence transformation method. From the recurrence equation, we obtain the dependence vectors: $d_1 = \begin{pmatrix} 0 & 1 \end{pmatrix}^T$, $d_2 = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$, $d_3 = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$. A possible time function is $\tau(z) = z_0 + z_1 = \lambda^T z$ where $\lambda = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$. For the processor allocation function α, we can use $u = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$.



(a) Running times.

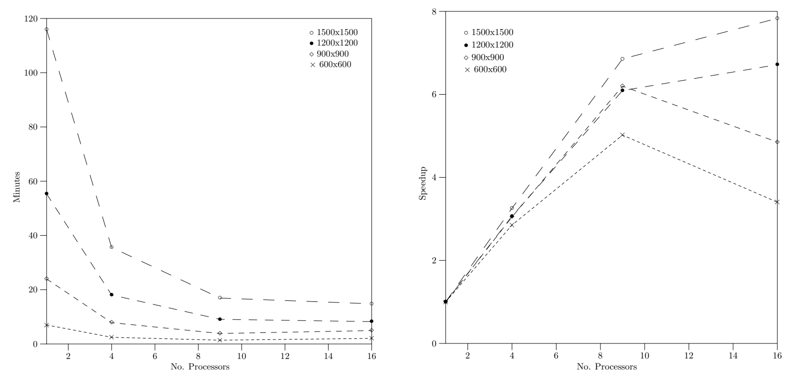
(b) Speedups.

Fig. 5. Matrix product results on a Beowulf cluster.

This will result in the following scheduling of the tasks to be executed by p processors, each with $O(mn/p)$ local memory. The string A is stored in all processors, and the string C is divided into p pieces, of size $\frac{n}{p}$, and each processor P_i, $1 \leq i \leq p$, receives the i-th piece of C $(c_{(i-1)\frac{n}{p}+1} \ldots c_{i\frac{n}{p}})$. P_i^k denotes the work of Processor P_i at round k. Thus initially P_1 starts computing at round 0. Then P_1 and P_2 can work at round 1, P_1, P_2 and P_3 at round 2, and so on. In other words, after computing the k-th part of the sub-matrix S_i (denoted S_i^k), processor P_i sends to processor P_{i+1} the elements of the right boundary (rightmost column) of S_i^k. These elements are denoted by R_i^k. The systolic algorithm requires $O(p)$ communication rounds and $O(mn/p)$ local computing time.

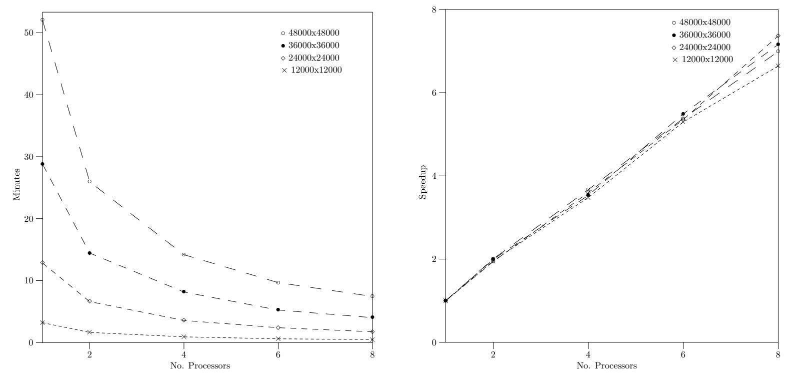
4 Experimental Results

We ran our experiment on a 16-node Beowulf cluster, using the LAM-MPI interface. Each node has a 1.2GHz AMD Thunderbird Athlon processor, 256 KB L2 cache, 768 MB of RAM memory and 30.73 GB hard disk. The nodes are connected by a Switch 3COM 3300 FastEthernet 100Mb. We also used an 18-node grid using *InteGrade* (see [4]) middleware that provides efficient use of desktop microcomputers that are idle and available in a computer laboratory for graduate students, consisting of Pentium II 400 Mhz and 3 Athlon 1700+ desktop



(a) Running times. (b) Speedups.

Fig. 6. Matrix product results on the InteGrade grid.



(a) Running times. (b) Speedups.

Fig. 7. String alignment results on the InteGrade grid.

microcomputers interconnected in a local area network by 100Mb Ethernet. We have implemented and used some of the communication primitives of the Oxford BSPLib such as bsp_put() and bsp_get(). The number of processors used is always a perfect square (1, 4, 9, etc),

Figure 5 shows the results obtained on the Beowulf cluster, for 50 matrix products of sizes 900×900 to 1800×1800. The super-linear behavior on some of the speedup curves can be explained by the more efficient use of cache in the parallel program. The speedups start to decrease for large matrix sizes, when the communication costs start to dominate in relation to the overall computation time. Figure 6 shows the matrix product results on the InteGrade grid of desktop computers. Figure 7 show the results for the string alignment algorithm, also run on the InteGrade grid.

5 Conclusion

We propose to use the dependence transformation method, to generate parallel algorithms for cluster or grid computing, after some suitable adaptation. It should be observed that the method is general. In fact we have used this method to derive parallel algorithms for other problems, such as convolution, and transitive closure. All these algorithms share the desirable property of local communication with a few other processors, with no global communication. TCP/IP connection start-up is heavy. Thus an important issue to be addressed is the reuse of connections

6 Acknowledgments

We wish to thank the anonymous referees for their helpful comments.

References

1. C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Proceedings of the 2003 International Conference on Computational Science and its Applications - ICCSA 2003*, volume 2668 of *Lecture Notes in Computer Science*, pages 249–258. Springer Verlag, 2003.
2. M. Cosnard. Designing parallel algorithms for linearly connected processors and systolic arrays. *Parallel Computing*, 1:273–317, 1990.
3. J. A. B. Fortes and D. I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, 1985.
4. Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.
5. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.
6. G. M. Megson. *An Introduction to Systolic Algorithm Design.* Oxford University Press, 1992.
7. D. I. Moldovan. *Parallel Processing: from Applications to Systems.* Morgan Kaufmann Publishers, 1993.
8. 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.
9. S. A. Orszag. Transform methods for calculation of vector coupled sums: Application to the spectral form of the vorticity equation. *Journal Atmospheric Science*, 27:890–895, 1970.
10. P. Quinton and Y. Robert. *Algorithmes et architectures systoliques.* Masson, 1989.