Coordination in the Poker Parallel Programming Environment: A Parallel Code Optimisation

Duane A. Bailey
Janice E. Cuny
Bruce B. Macleod

COINS Technical Report 85-21
August 1985

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Much of this work was supported by the Office of Naval Research, contract N000014-84-K-0647. Duane Bailey was supported under an American Electronics Association Fellowship from ComputerVision.

Abstract. In highly parallel computation, where a large number of closely coupled processors cooperate on a single task, communication overhead is often the dominant factor in efficiency. We report here on parallel code optimizations designed to reduce this overhead. Our optimizations are targeted for synchronous machines. Because we have found such machines to be difficult to program, we start with high-level asynchronous code which we automatically convert for synchronous execution before applying our optimizations. Using relatively simple transformations, we are able to demonstrate significant performance improvements for many programs.

In highly parallel computation, where a large number of closely coupled processors cooperate on a single problem, communication overhead is often the dominant factor in efficiency. This is particularly true for nonshared memory machines where processors communicate entirely through message transmission. Message transmission may be either asynchronous or synchronous. If it is asynchronous, the overhead includes the execution of handshaking protocols and idling ("busy waits") introduced by the underlying system to delay a process attempting a read before the corresponding data is available or a write to a full buffer. If it is synchronous, the overhead includes idling explicitly inserted by the programmer to insure that corresponding reads and writes occur simultaneously (handshaking is not needed). In either case, a program's runtime performance can be significantly degraded. We report here on optimizations aimed at reducing communication overhead for architectures with synchronous capabilities.

Our optimizations have been implemented as part of the Poker Parallel Programming Environment [1] which was designed to provide access to highly parallel machines in the Blue CHiP family of architectures [2]; specifically, Poker was designed as the front end for the Pringle, a 64 processor CHiP prototype[3]. Processors in the CHiP architecture interact through message transmission which, as currently implemented for the Pringle, is asynchronous. That is, message values are buffered with writing processors automatically delayed until buffer space is available and reading processors automatically delayed until data is available. CHiP processors, however, have access to a common clock and so it would also have been possible to implement truly synchronous communication in which processors are never delayed and corresponding I/O operations are timed to occur simultaneously without any handshaking.

In cases where processes have large, runtime variations in their relative rates of I/O (because for example of conditional branches in which the parallel clauses do not contain the same number or type of I/O operations), asynchronous execution is preferrable. For the cases without such variation, however, synchronous execution often leads to performance improvements both because I/O operations do not require handshaking and because synchronous programs have a predictability which permits a number of program restructuring optimizations. At the same time, we have found that it is difficult to capitalize on this potential efficiency because synchronous programming is extremely difficult. The programmer must consider the relative progress of all processes simultaneously; if this is done at a low level, he is inundated with details and his code is often machine and problem size dependent; if it is done at a high level, he must make worst case assumptions about the timing of corresponding blocks of code which introduces unacceptable delays. In addition, the optimal patterns of processor interactions are often complex and non-intuitive. As a result, our approach is to allow the programmer to use the high-level, asynchronous communication protocols of XX [1] (the programming language used within Poker to specify sequential code) which we automatically convert for synchronous execution where practically possible.

We have previously reported on algorithms for accomplishing this conversion, a process we called *coordination* [4]. In the next section, we discuss coordination as it is implemented in the Poker Parallel Programming Environment and we demonstrate that coordination often achieves significant improvements in runtime performance. In the following section, we use coordination as the basis for parallel code optimizations that result in further improvements for many programs. Finally, we discuss areas for additional research including the scheduling of multiple processes on a single processor and generalizations to architectures in which processors are closely coupled but do not share a common clock.

COORDINATION ALGORITHMS

Within Poker, coordination occurs as the second phase of the XX compiler. In the first phase, an intermediate assembly code is produced with annotations for the coordinator (including the classification of each instruction – operation, read, write or branch – and the time required for that instruction). The coordinator then restructures the assembly code into synchronous code which is passed to the final assembly phase.

Although theoretically any program that runs with finite message buffer space can be coordinated, it is not always practically advantageous to do so. For example, the simple program pictured in Figure 1 can be converted to synchronous mode only if additional communication is inserted to "inform" Process B of the branches taken by its neighbor.

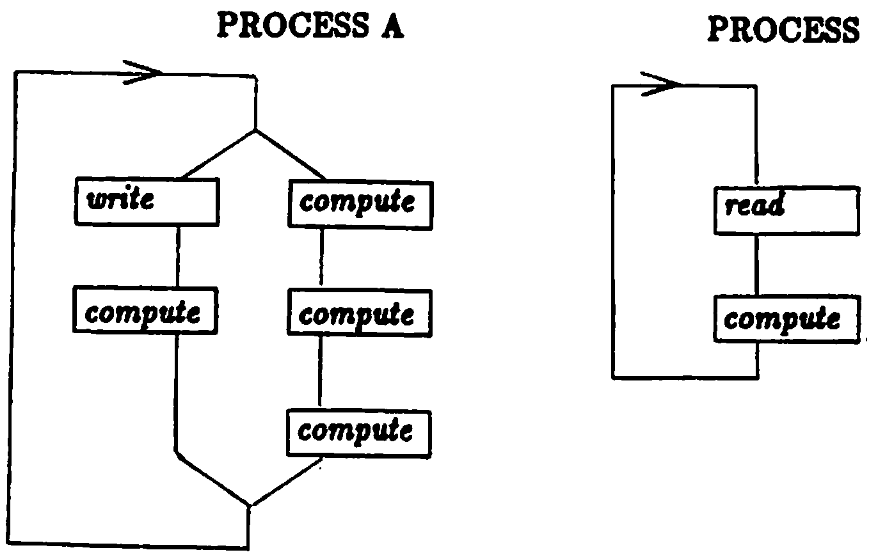


Figure 1. A two process system that requires additional communication for synchronisation.

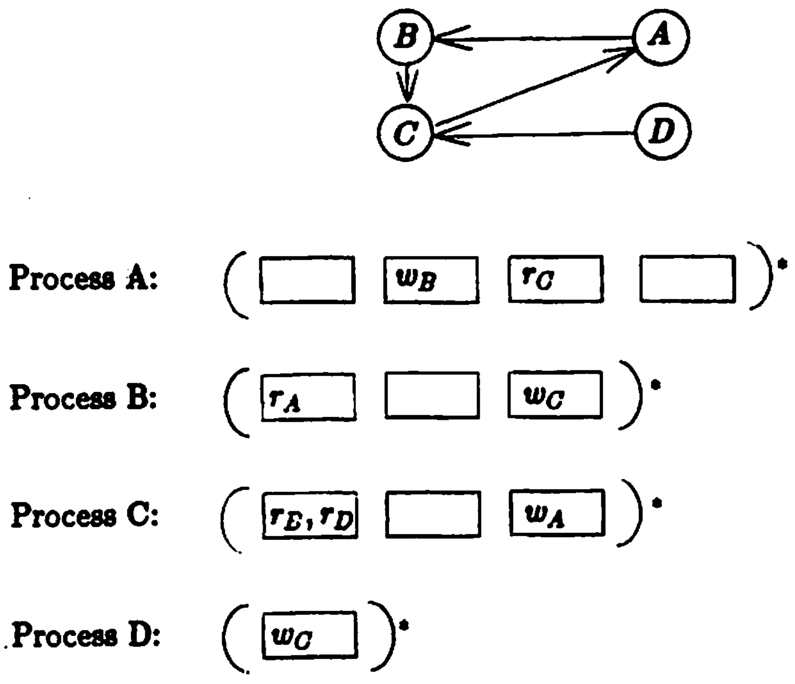
Since communication costs are usually significant, it would be more efficient to leave this program for asynchronous execution. Consequently, we do not coordinate all programs but limit our attention to those that can be converted without the addition of communication.

This restricts us to programs with very simple control structures: we can allow only unnested loops and conditionals with balanced I/O (a conditional has *balanced* I/O if both of its clauses contain the same number and type of I/O operations; they do not have to be in the same order). It should be noted that almost of the programs written for the CHiP architecture that we have encountered meet these constraints. This is because CHiP programs are written in "phases" corresponding to different interconnection structures and we coordinate only a single phase at a time. Within a phase, we encounter only very simple control structures. More general programs will have to be decomposed into appropriate segments before our techniques can be used. In any case, programs that we can not convert are not altered in the coordination phase; programs that we can convert are coordinated with one of two algorithms (previously described in detail [4]).

Our first algorithm works on any input program subject to the above restrictions on control structure. It forces all processors to execute their loop iterations in lock step by simulating a single iteration and replacing busy waits with explicit idle operations, as in the example shown in Figure 2. This results in a performance improvement if the cost of the original, asynchronous I/O plus waits is greater than that of the new, synchronous I/O plus inserted idles.

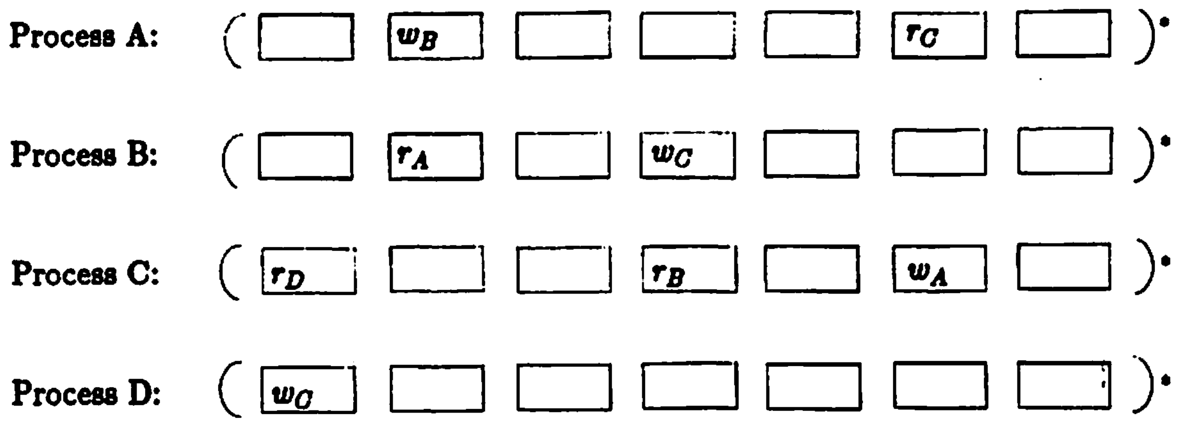
Table 1 gives the results of coordination with this algorithm for a variety of parallel programs. Since the conversion algorithms do not change the computation of the programs*, they can only reduce *communication overhead* , the time required for handshaking protocols, explicit idles, and busy waits. We therefore express our results in terms of the percentage reduction in communication overhead. The table gives the computation time

* We do not, at this time, do any dependency analysis, and therefore, we re-order I/O operations (saving and later restroing the appropriate values); we do not re-order any other operations.



B
A
C
D
Process A: ([ ] [w_B] [r_C] [ ])*
Process B: ([r_A] [ ] [w_C])*
Process C: ([r_B, r_D] [ ] [w_A])*
Process D: ([w_C])*

Figure 2a. An asynchronous system and its communications graph. Blank boxes denote computation; subscripts indicate the destinations of read and write operations.



Process A: ([ ] [w_B] [ ] [ ] [ ] [r_C] [ ])*
Process B: ([ ] [r_A] [ ] [w_C] [ ] [ ] [ ])*
Process C: ([r_D] [ ] [ ] [r_B] [ ] [w_A] [ ])*
Process D: ([w_C] [ ] [ ] [ ] [ ] [ ] [ ])*

Figure 2b. The synchronous version of the above program as produced by our first coordination algorithm. Blank boxes denote either computation or inserted idles.

Figure 2.

Program	Base Time	Asynchronous Loop Time	Synchronous Loop Time	Overhead Reduction
Grid Sort [5]				
4×4	2782	6508	4754	47.1
8×8	2782	6508	4754	47.1
LU Decomposition [6]				
4×4	1377	2614	2488	10.2
8×8	1377	2936	2788	9.5
Matrix Multiplication [6]				
(cyclic) 4×4	1325	2509	2069	37.2
(cyclic) 8×8	1325	2605	2262	26.8
Multigrid [7]				
4×4	4420	10157	10109	0.8
8×8	4420	11017	10872	2.2
Ring Maximum				
10 nodes	823	1439	976	75.2
20 nodes	823	1446	1326	19.3
30 nodes	823	1448	1326	19.5
40 nodes	823	1464	1326	21.5
Transitive Closure [8]				
4×4	3802	7252	5903	39.1
8×8	7210	15486	11314	50.4
Tridiagonal Linear System Solver [9]				
4 nodes	1619	3040	1776	89.0
8 nodes	1619	3007	1966	75.0
Vector/Matrix Multiplication [6]				
4 nodes	1232	2645	1966	48.1
8 nodes	1232	2627	2009	44.3
Dynamic Programming [8]				
4×4	3584	5844	6727	-39.1
8×8	8200	15240	26058	-153.7
Fast Fourier Transform [10]				
4×3	1708	2483	3149	-85.9
8×4	1708	2507	4487	-247.8
Matrix Multiplication				
(acyclic) 4×4	1325	2799	2064	49.9
(acyclic) 8×8	1325	3021	2168	50.3
Tree Summation				
15 nodes	861	1442	2758	-226.5
63 nodes	861	1434	4308	-501.6

Table 1: Communication overhead reduction achieved by coordination with Algorithm 1. The tested programs are listed in the first column with annotations giving the number of processors and their configuration. The computation time for a single iteration is listed in the second column in microseconds. Asynchronous and synchronous loop times are given in the third and fourth columns, also in microseconds. Reduction in communication overhead is given in the final column in percentages.

for a single iteration of the algorithm excluding all I/O overhead, a value we call the *base computation time*,the asynchronous iteration time (averaged over ten iterations), the synchronized iteration time, and the percentage reduction that we have achieved. Where the values differ from processor to processor, the percentage is based on the processor with the smallest, initial overhead. It should also be noted that in some cases – such as the LU decomposition and the cyclic matrix multiplication – our performance improvement decreases as the number of processors goes from 16 to 64. While we have not been able to run larger simulations, we believe that this is due to "edge effects" in the smaller configurations unduly influencing timings. We expect that this effect will level off, making the 64 processor results more representative of the improvements that we would see on larger machines.

As can be seen, our first algorithm does well on programs such as matrix multiplication and the tridiagonal linear system solver which have cyclic communication graphs as shown

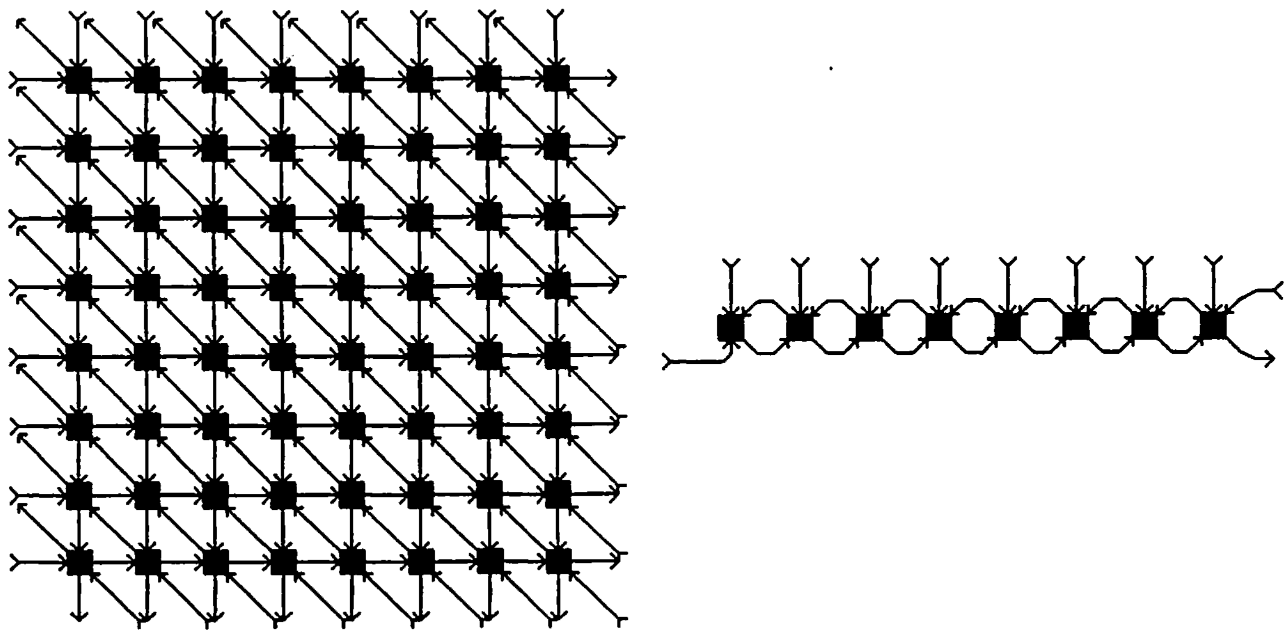


Figure 3. Cyclic communications graphs for Matrix Multiplication (left) and Tridiagonal System Solver (right). Nodes represent processes and edges their interconnections.

in Figure 3. It does not do as well on programs with acyclic communications graphs such as the FFT or the Tree Summation pictured in Figure 4. This is in part because the asynchronous execution of programs with acyclic communication patterns results in the pipelining of iterations. The second of our coordination algorithms, which works only for programs with acyclic communications graphs, incorporates pipelining. The loop

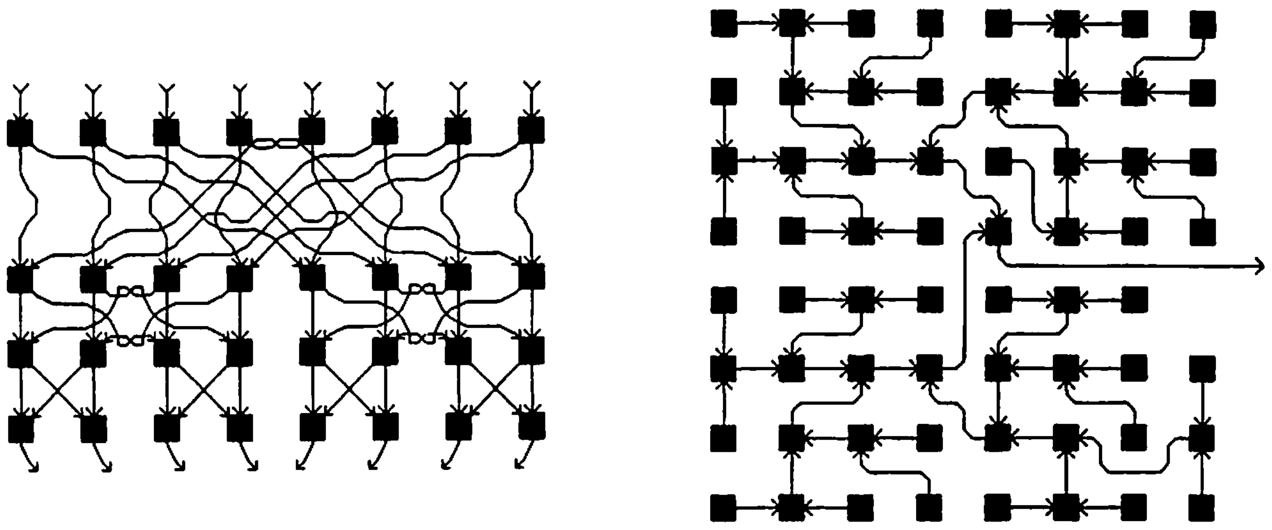


Figure 4. Acyclic communications graphs for FFT (left) and tree summation (right).

iterations are "unrolled" (starting with the code for "source" processes and queuing I/O where necessary), until all processes are executing together. This saturates the pipeline as shown in Figure 5. This second coordination algorithm achieves the performance improvements shown in Table 2. Notice that while the improvements are not uniformly good, they are considerably better than in the case of Algorithm 1.

In either case, the improvements in runtime performance that we have been able to achieve with coordination alone are limited. To some extent this is because of the char-

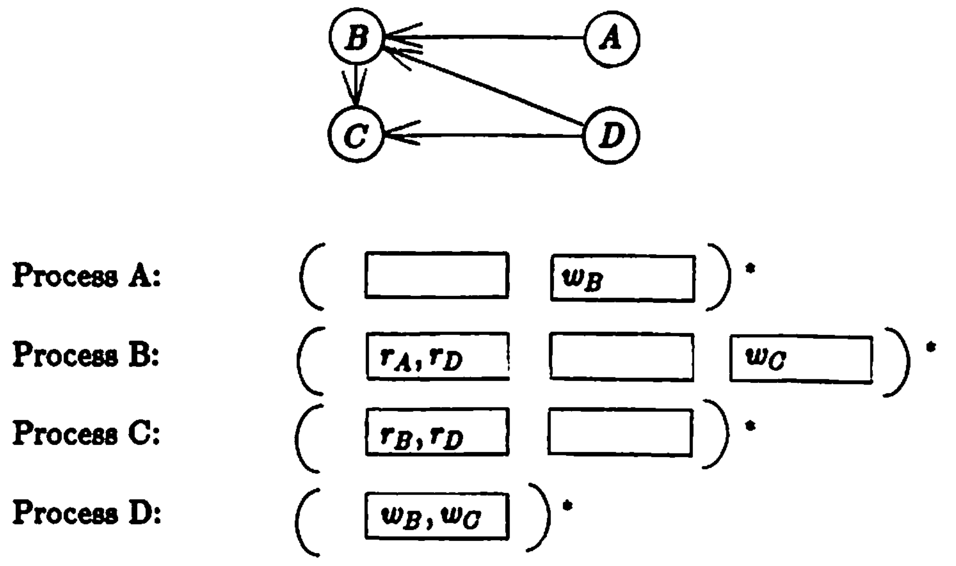


Figure 5a. An asynchronous system.

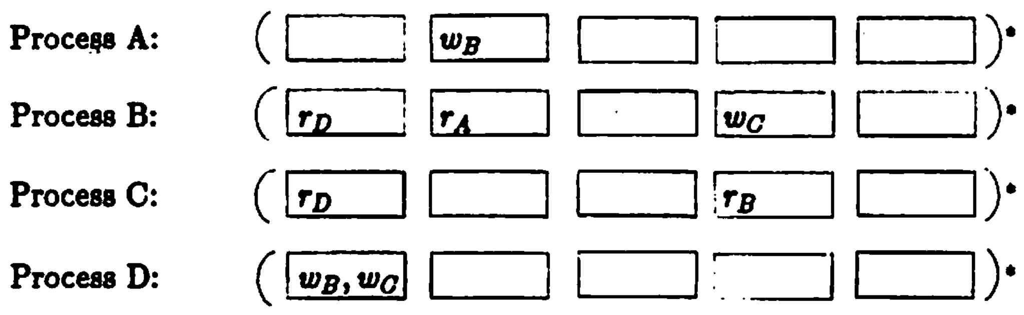


Figure 5b. The synchronous version of the above program as produced by our first coordination algorithm.

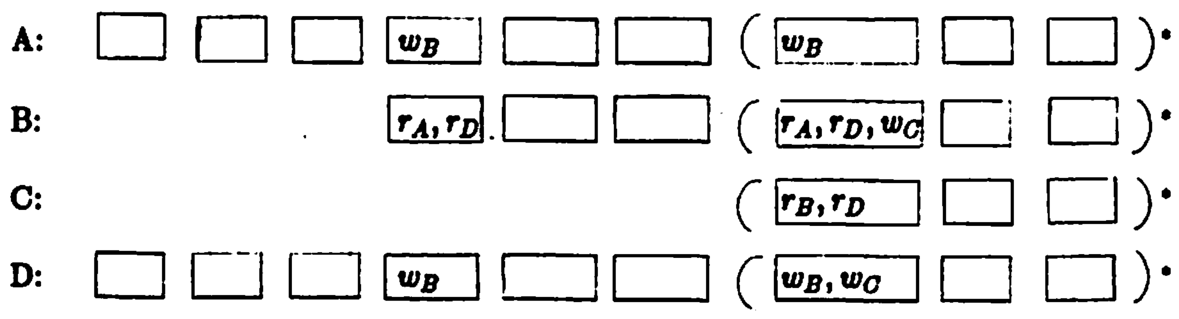


Figure 5c. The synchronous version of the above program as produced by our second coordination algorithm.

Figure 5.

Program	Base Time	Asynchronous Loop Time	Synchronous Loop Time	Overhead Reduction
Dynamic Programming [8]				
4×4	3584	5844	4644	53.1
8×8	8200	15240	11503	53.1
Fast Fourier Transform [10]				
3×4	1708	2483	2590	-13.8
4×8	1708	2507	3213	-88.4
Matrix Multiplication				
(acyclic) 4×4	1325	2799	2662	9.3
(acyclic) 8×8	1325	3021	2728	17.3
Tree Summation				
15 nodes	861	1442	1461	-3.3
63 nodes	861	1434	1504	-12.2

Table 2: Communication overhead reduction achieved by coordination with second algorithm. Columns are as in Table 1.

acteristics of Pringle prototype; in particular, much of its I/O protocol is implemented in software and communication costs are dominated by the time spent moving data pointers rather than by the time spent in data transmission. This may well be different on other hardware. Even on the Pringle, however, the effects of coordination can be improved because the synchronous programs it creates are amenable to a number of parallel program optimizations.

COORDINATION AS THE BASIS FOR PARALLEL CODE OPTIMIZATIONS

Synchronous programs – whether they result from the coordination of asynchronous code or from the direct, hand coding – usually contain explicit idles, inserted to maintain synchronization. For the simple control structures that we allow, this insertion occurs in three places: before I/O operations (reads or writes) whose corresponding operation will not be ready to execute*; at the end of the shorter of the two clauses of a conditional branch, padding them to the same length; and before the final branches of loops, insuring that all processors exit their loops together. We have implemented two optimisations for parallel, synchronous code that remove as much of this explicit idling as possible. The first optimization *bubbles* idles to the bottom of loops where they become candidates for removal and the second optimization *reorients* loops so that synchronisation is maintained without idling.

The bubbler attempts to move idles to the end of loops where they can potentially be eliminated. It scans the synchronous code for all processes from top to bottom; when an idle is encountered it tries to move it toward end of the loop as follows:

- idles occurring before an operation other than an I/O or branch are moved over that operation (Figure 6a)

  and

- idles occurring before a read operation are matched with idles before the corresponding write and equivalent portions of both idles (equal to the smaller of their lengths) are moved over the read and write (Figure 6b).

In this way, the idles appear to "bubble" toward the end of the loop where they accumulate as shown in Figures 7a and 7b. After all possible moves are made, the minimum pre-branch idle time can be skimmed from each loop without changing either the synchronization characteristics or the computation performed by the program, as shown in Figure 7c.

* In fact, our coordinator makes various decisions in scheduling I/O operations – such as whether to enqueue or immediately transmit data that is to be written or whether to begin dequeing data prior to the actual start of its read – that may result in idles being inserted into the code for both partners of an I/O operation.

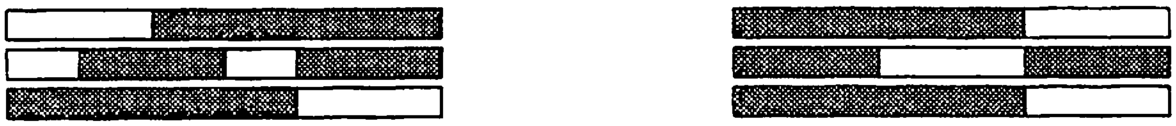


Figure 6a. Profile of two instances of code motion over a non-I/O operation. Code profile on left before move; code profile on right after move.

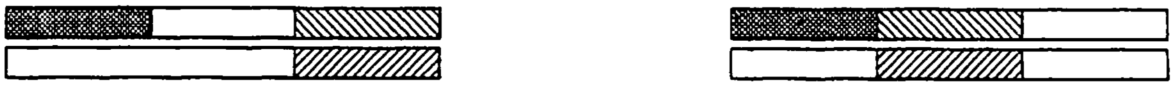


Figure 6b. Profile of a single instance of code motion over matching I/O operations. Code profile on left before move; code profile on right after move.

Figure 6.

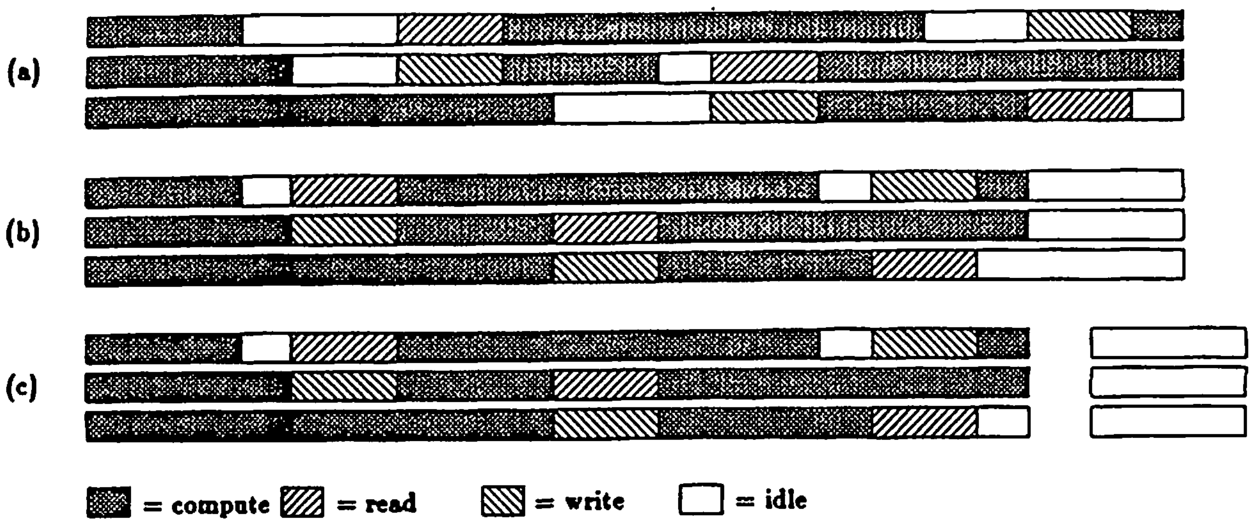


Figure 7. Profiles of loops subject to bubble optimisation; (a) before optimisation, (b) before the "skimming" process, and (c) after excess idles have been removed. Loop boundaries remain synchronized.

Bubbling can not eliminate all idle time but it does guarantee that I/O operations will occur as early as possible within the schedule created by the coordinator.

To maintain synchronous execution, the loop times of all processors must be identical which is usually achieved by forcing all processes to start and finish their iterations together. In reality this is not necessary – it is only necessary to insure that the loops are of the same length. This distinction is important for many programs, especially those with acyclic communications graphs where we can identify sources and sinks for the data. In these cases, the computation of a particular processor is delayed with respect to the

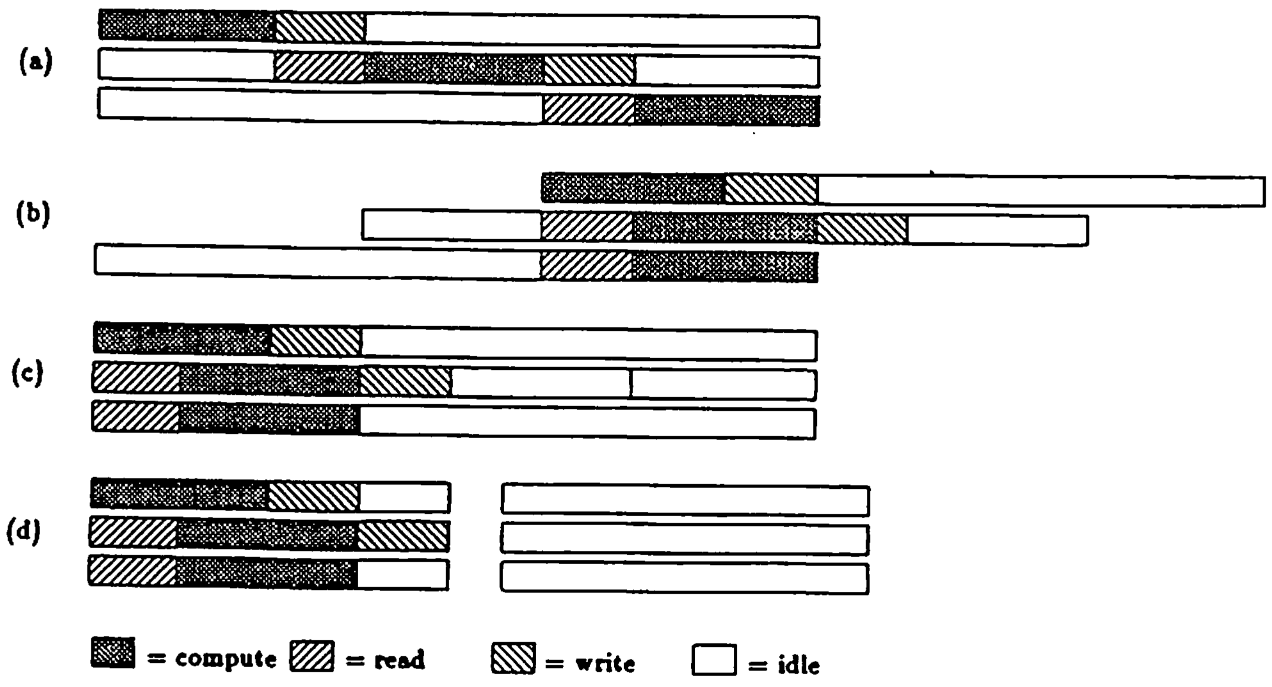


Figure 8. Profiles of loops subject to reorientation; (a) before optimization, (b) before optimization with loop computation lined up, (c) after reorientation before "skimming" of excess idles and (d) after excess idles are skimmed. Loops are reoriented with idles at the end of the loop. Since boundaries are no longer synchronised the loop initialization must also be modified (not shown).

loop start, by a time proportional to its distance from the furthest source. The resulting computation then is clustered between two idles whose total duration over all processors is roughly constant; for some processors a substantial portion of this idling is trapped at the top of the loop. Loop reorientation, as shown in Figure 8, allows the trapped idle time to be relocated to the end of the loop where it again becomes a candidate for removal.

The results of these two optimizations are shown in Table 3 with the final column showing our best runtime performance improvement. As can be seen the optimizations result in further improvements over coordination, doing particularly well on the dynamic programming, FFT, acyclic matrix multiplication, and tree summation algorithms. Code produced by Algorithm 1 improves more with optimization than that produced by Algorithm 2. This is to be expected since Algorithm 1, in forcing all iterations into lock step, tends to add more idles than Algorithm 2. Notice also that the code produced by Algorithm 1 after optimization is almost always better than the code produced by Algorithm 2. This is an artifact of the limitations on the size simulation that we can run; as the machine size increases, Algorithm 2 will win out on programs with acyclic graphs. (This implies that our techniques may not always be beneficial since Algorithm 2 does not do well on the FFT, for example. We are looking at characterizations of the class of algorithms that we can successfully optimize. In the mean time, it is possible to compare our results to sample runs in order to determine whether or not to utilize the synchronous version.)

The optimizations that we have implemented are very simple; in particular, we do not do any analysis of program data dependencies and so we can accomplish only limited code movement. Even our simple techniques, however, often result in impressive performance improvements and we expect that the addition of more sophisticated dependency analysis would allow us to do even better, perhaps reducing the time spent on computation as well as

Program	Algorithm 1 Reduction	Algorithm 1 Reduction Optimized	Algorithm 2 Reduction	Algorithm 2 Reduction Optimized	Best Result
Grid Sort [5]					
4×4	47.1	47.1	N/A	N/A	47.1
8×8	47.1	47.1	N/A	N/A	47.1
LU Decomposition [6]					
4×4	10.2	17.4	N/A	N/A	17.4
8×8	9.5	9.5	N/A	N/A	9.5
Matrix Multiplication [6]					
(cyclic) 4×4	37.2	37.2	N/A	N/A	37.2
(cyclic) 8×8	26.8	26.8	N/A	N/A	26.8
Multigrid [7]					
4×4	0.8	2.5	N/A	N/A	2.5
8×8	2.2	4.4	N/A	N/A	4.4
Ring Maximum					
10 nodes	75.2	75.2	N/A	N/A	75.2
20 nodes	19.3	19.3	N/A	N/A	19.3
30 nodes	19.5	19.5	N/A	N/A	19.5
40 nodes	21.5	21.5	N/A	N/A	21.5
Transitive Closure [8]					
4×4	39.1	45.2	N/A	N/A	45.2
8×8	50.4	53.4	N/A	N/A	53.4
Tridiagonal Linear System Solver [10]					
4 nodes	89.0	89.0	N/A	N/A	89.0
8 nodes	75.0	78.5	N/A	N/A	78.5
Vector/Matrix Multiplication [6]					
4 nodes	48.1	51.5	N/A	N/A	51.5
8 nodes	44.3	47.3	N/A	N/A	47.3
Dynamic Programming [8]					
4×4	-39.1	91.2	53.1	57.7	91.2
8×8	-153.7	17.0	53.1	53.1	53.1
Fast Fourier Transform [10]					
4×3	-85.9	86.2	-13.8	-7.4	86.2
8×4	-247.8	86.6	-88.4	-26.2	86.6
Matrix Multiplication					
(acyclic) 4×4	49.9	49.9	9.3	9.3	49.9
(acyclic) 8×8	50.3	50.5	17.3	17.3	50.5
Tree Summation					
15 nodes	-226.5	48.7	-3.3	32.2	48.7
63 nodes	-501.6	48.0	-12.2	5.8	48.0

Table 3. Results of optimization combined with coordination. All values in percentage reduction of communication overhead.

that spent on communication overhead. It should also be noted that our results are, in some cases, sensitive to the original order of I/O operations; we are investigating the feasibility of automatic re-ordering of independent I/O operations for maximum efficiency.

AREAS FOR FURTHER RESEARCH

We are currently pursuing two areas for further research: the extension of coordination to the scheduling of multiple logical processes on a single processor; and its generalization to asynchronous architectures.

It is rarely the case that the logical process structure of an algorithm exactly matches the physical structure of the target architecture. More often, the two structures differ and frequently this difference is in cardinality [11], that is, the number of logical processes exceeds the number of physical processors. When this happens, the logical structure is mapped onto the physical structure in such a way that multiple processes execute on a processor. In general, these processes must be multiplexed to avoid deadlock and thus incur the substantial overhead of context switching. If the original program can be coordinated, however, all of the processes on a processor can be combined into a single, sequential process without introducing deadlock. These newly created processes can then be run without context switching. As a simple example, Figure 9 shows a four process system after coordination and its mapping onto a two processor system.

Another area for further consideration is the extension of these ideas to systems in which processors do not share a common clock. Our techniques will not be useful for truly distributed systems in which no assumptions can be made about the relative timing of processors but we expect that they will be useful for many of the proposed highly parallel machines which are closely coupled and designed for single users (making it possible to put bounds on the relative differences in execution speeds). We are investigating the use of coordination as the basis for scheduling code across such processors, looking specifically at the distribution of *forall* and *doacross* loops.

PROCESS A:	$write_D$				$read_D$		
PROCESS B:		$write_D$				$read_D$	
PROCESS C:			$write_D$				$read_D$
PROCESS D:	$read_A$	$read_B$	$read_C$	*compute*	$write_A$	$write_B$	$write_C$

Figure 9a. Four Process System.

PROCESSOR 1:	$write_2$	$write_2$			$read_2$	$read_2$	
PROCESSOR 2:	$read_1$	$read_1$	*read*	*compute*	$write_1$	$write_1$	*read*

Figure 9b. Mapping of Four Process System onto Two Processors.

Figure 9. Coordinated four process system and its mapping onto a two processor architecture. Processes A and B have been mapped onto Processor 1 and Processes C and D have been mapped onto Processor 2. Code shown is for a single iteration; blank boxes denote idles (*compute*s are shown explicitly); subscripts indicate the destinations of read and write operations. Note that the I/O between Processes C and D has been removed (replaced by internal, unsubscripted reads).

CONCLUSIONS

We have used coordination – a technique for converting programs from asynchronous to synchronous execution mode – as the basis for several simple parallel code optimisations. We have found that these optimisations achieve a significant reduction in communication overhead and we expect that, as we incorporate more sophisticated dependency analysis, these techniques will become even more valuable. In addition, we expect to be able to extend these techniques to optimise parallel segments within more complex programs for closely coupled (although not necessarily synchronous) architectures.

Acknowledgments. We would like to thank Larry Snyder who was one of the originators of coordination and a source of encouragement for our efforts on optimisation.

REFERENCES

[1] Lawrence Snyder, "Parallel programming and the Poker Programming Environment," *Computer* 17(7), pp. 27-37 (1984).

[2] Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer* 15(1), pp. 47-56 (1982).

[3] Alejandro Kapauan, Ko-Yang Wang, Dennis Gannon, Janice Cuny, and Lawrence Snyder, "The Pringle: An experimental system for parallel algorithm testing and software development," Proceedings of The 1984 International Conference on Parallel Processing, pp. 1-6.

[4] Janice Cuny and Lawrence Snyder, "Compilation of data-driven programs for synchronous execution," Proceedings of the 1983 Annual Symposium on Principles of Programming Languages, January 1983, pp. 197-202.

[5] Charles C. Weems, "Image Processing with a Content Addressable Array Parallel Processor," Ph.D. Thesis, Computer and Information Science Department, University of Massachusetts, Amherst, May 1984.

[6] H. Kung and C. Leiserson, "Systolic Arrays (for VLSI)" in *Introduction to VLSI Systems* by C. Mead and L. Conway, Addison-Wesley, 1980.

[7] Piyush Mehrotra, and John Van Rosendale, The Blaze Language: A Parallel Language for Scientific Programming. ICASE Report # 85-29, Langley Research Center, Hampton, Virginia (May 1985).

[8] L. Guibas, H. Kung and C. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," Caltech Conference on VLSI, California Institute of Technology, pp. 509-525 (1979).

[9] D. Gannon, L. Snyder and J. Van Rosendale, "Programming substructure computations for elliptic problems on a CHiP system," Proc. Sym. on the Impact of New Computing Systems on Computational Mechanics, ACME, pp. 65-80, 1983.

[10] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, MA, pp. 252-264 (1974).

[11] Francine Berman and Lawrence Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," Proceedings of the 1984 International Conference on Parallel Processing, pp. 307-309 (1984).