Non-strict Evaluation of the FFT Algorithm in Distributed Memory Systems*

Alfredo Cristóbal-Salas[1], Andrei Tchernykh[2], Jean-Luc Gaudiot[3]

[1] School of Chemistry Sciences and Engineering, University of Baja California, Tijuana, B.C. Mexico 22390, cristobal@uabc.mx
[2] Computer Science Department, CICESE Research Center Ensenada, BC, Mexico 22830, chernykh@cicese.mx
[3] Electrical Engineering and Computer Science, University of California, Irvine, USA, gaudiot@uci.edu

Abstract. This paper focuses on the partial evaluation of local and remote memory accesses of distributed applications, not only to remove much of the excess overhead of message passing implementations, but also to reduce the number of messages, when some information about the input data set is known. The use of split-phase memory operations, the exploitation of spatial data locality, and non-strict information processing are described. Through a detailed performance analysis, we establish conditions under which the technique is beneficial. We show that by incorporating non-strict information processing to FFT MPI, a significant reduction of the number of messages can be archievd, and the overall system performance can be improved.

1 Introduction

Parallelization of scientific and engineering applications has become essential in modern computer systems. The variety of parallel paradigms, languages, runtime systems, and architectures makes optimization of parallel programs as equally demanding as the design of the parallel algorithm itself. High performance of the Fast Fourier Transform (FFT) is a key issue in many application [20]. There have been several attempts to parallelize FFT. For example, an algorithm suitable for 64-processor nCUBE 3200 hypercube multicomputer was presented in [17] where a speedup of up to 16 with 64 processors was demonstrated. In [10], the binary exchange algorithm for the parallel implementation of FFT was presented. Also, FFT for hypercube multicomputers and vector multiprocessors was discussed in [19]. The implementation of a parallel FFT algorithm on a 64-processor Intel iPSC was

* This work is partly supported by CONACYT (Consejo Nacional de Ciencia y Tecnología de México) under grant #32989-Aand by the National Science Foundation under Grants No. CSA-0073527 and INT-9815742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views neither of the National Science Foundation nor of CONACYT.

Recent Advances in Parallel Virtual Machine and Message Passing Interface. J. Dongarra, D. Laforenza, S. Orlando (eds.), Lectures Notes in Computer Science, Springer-Verlag, Berlin, vol. 2840, pp. 188-195, 2003

described in [3]. Finally, in [2], two FFT implementations (recursive and iterative), written in Id, were presented.

Nevertheless, in spite of its reduction in complexity and time, FFT remains expensive mainly in distributed memory parallel computers where network latency significantly affects the performance of the algorithm. In most cases, the speedup which can be gained by parallelization is limited due to inter-process communication. Because of this, programming for distributed architectures has been somewhat restricted to regular, coarse-grained, and computation-intensive applications. FFT exploits fine grain parallelism, which means that an improvement at the communication level plays an extremely important role. Ideas for improvements include optimization by pipelining of communications such as considered in [6] where a simple communication mechanism named Active Messages was proposed. Under Active Messages, the network is viewed as a pipeline operating at a rate determined by the communication overhead and with a latency which is directly related to the message length and the network depth.

Another approach to speeding up applications is based on Partial Evaluation (PE) [13]. PE [11] is an automatic program transformation technique which allows the partial execution of a program when only some of its input data are available, and it also "specializes" the program with respect to partial knowledge of the data. In [9], PE is incorporated in a library of general-purpose mathematical algorithms in order to allow the automatic generation of fast, special-purpose programs from generic algorithms. The results for the FFT show speedup factors between 1.83 and 5.05 if the size N of the input is available, where N ranges from 16 and 512. Good speedup for larger N is achieved despite the growth in code size which reaches $O(N \log_2 N)$.

In this paper, we present a 1D-FFT for distributed memory systems with an optimization at the communication level which reduces the number of messages by exploiting data locality and by applying a partial evaluation technique. We describe implementations based on multi-assignment and single-assignment data structures in distributed environments. Finally, we also discuss a caching mechanism for single-assignment data structures.

In the next section, we present several approaches to FFT parallelization. Description of our benchmark programs are made in section 3. In section 4, we discuss our experimental results. Lastly, conclusions are presented.

2 FFT Optimization

MPI provides many benefits for scalable parallel computations. However, one of its drawbacks is that it allows unrestricted access to data. The performance of MPI implementations is bounded by the performance of the underlying communication interface. However, an efficient interface does not necessarily guarantee a high performance implementation. One possible way to increase performance is to eliminate synchronization issues by non-strict data access and fully asynchronous operations, and to reduce the number of messages. We use single-assignment I-Structures [1] (ISs) to facilitate asynchronous access when structure production and consumption can be allowed to proceed with a looser synchronization than

Recent Advances in Parallel Virtual Machine and Message Passing Interface. J. Dongarra, D. Laforenza, S. Orlando (eds.), Lectures Notes in Computer Science, Springer-Verlag, Berlin, vol. 2840, pp. 188-195, 2003

conventionally understood. In our implementation, ISs are managed by our Distributed I-Structure (DIS) memory system. We also use a mechanism to cache ISs memory requests; we call it Distributed I-Structures Software Cache (DISSC) to reduce number of messages by spatial and temporal locality exploitation, and by partial evaluation [4,21].

2.1 Non-strict Data Access and Software Caching

Our DIS [14] is a linked list of ISs where multiple updates of a data element are not permitted. In DIS, each element maintains a presence bit which has three possible states: full, empty, or deferred, indicating whether a valid data has been stored in the element. Split-phase operations are used to enable the tolerance of request latencies by decoupling initiators from receivers of communication/synchronization transactions. DISs facilitate the exploitation of parallelism while timing sequences and determinacy issues would otherwise complicate its detection, regain flexibility without losing determinacy, and avoid the cache coherence problem [16]. However, the overhead of DIS management becomes its major drawback [14]. In order to solve this problem, we use a caching mechanism, the Distributed I-Structure Software Cache (DISSC). Several efforts related to the optimization of IS memory systems using a caching mechanism have been presented in [5, 8, 12]. Our DISSC is a further development of the ISSC system designed for non-blocking multithreaded architectures and tested for the EARTH [14]. DISSC provides a software caching mechanism under a distributed address space. It takes advantage of spatial and temporal localities without hardware support. DISSC works as an interface between user applications and network and is implemented as an extension of the MPI library. It makes the cache system portable and provides non-blocking communication facilities. Accesses to DIS elements are naturally mapped to the split-phase transactions of MPI.

In DISSC, due to the long latency and unpredictable characteristics of a network, a second remote access to the data elements in the same data block (cache line) may be issued while the first request is still traveling. Hence, spatial data locality can be exploited. Temporal data exploitation refers to the reuse of data which is already in the cache. Because of the inherent cache coherence feature of DISs, no cache coherence problem exists. This significantly reduces the overhead of the cache system.

2.2 Partial Evaluation

In this section, an optimization technique based on partial evaluation is described. It enables the construction of general highly parameterized software systems without sacrificing efficiency. “Specialization” turns a general system into an efficient one, optimized for a specific parameter setting. It is similar in concept to, but in several ways stronger than, a highly optimizing compiler. Specialization can be done automatically or with human intervention. Partial evaluation may be considered a generalization of the conventional evaluation [7]. The use of partial evaluation for

distributed applications has been considered in the recent past. For instance, in [18], a distributed partial evaluation model that considers distributing the work of creating the specializations to computational agents called specialization servers is presented. Also in [15], the OMPI (Optimizing MPI) system that removes much of the excess overhead by employing partial evaluation and exploiting static information of MPI calls is considered.

The question now becomes, would it be possible to use partial evaluation, not only to remove much of the excess overhead of a program, but also to reduce the number of messages? The answer is effectively yes. The problem is to assess how to partially evaluate of split-phase memory operations under a distributed address space.

In this paper, we focus on non-strict information processing of split-phase memory operations to demonstrate the possibility to optimize distributed applications at the communication level when some program inputs are known. The splitting of data requests on split-phase transactions such as *send-a-request, receive-a-request, send-a-value* and *receive-a-value,* together with the ability of ISs to defer reads, when the values are not available, allow evaluating MPI programs partially without losing determinacy. To completely evaluate a *send-a-request* transaction, the element being requested and the process that owns the element have to be specified. For the FFT, the size N of the input vector determines the control and data structures of the program. Hence, if N is available, an *MPI_Send* instruction can be executed. The *receive-a-request* transaction can also be completely evaluated. The owner of the element executes the *MPI_Receive* instruction, checks the status of the element, and, if it is available, sends a value back to the requester by the *MPI_Send* instruction. Otherwise, it stores the request as a deferred read to this element. Later, when the element is produced and written, the owner of the element finds the list of pending reads (continuation vectors) and sends a value to the requestors by executing *MPI_Send* instructions. A *receive-a-value* transaction executes an *MPI_Receive* instruction and writes the value to the local memory of a requester.

Distributed programs where parallel control structure is completely determined by the size of the problem (data-independent programs) can be partially evaluated even if the data bindings of the input vector are not performed. Residual programs only include *send-a-value* and *received-a-value* transactions. More details about non-strict evaluation and partial evaluation of DIS and DISSC can be found in [4].

3 Experimental Results

In this section, we discuss the performance evaluation of the 1-D FFT algorithm with 2048 double precision complex data on an SGI ORIGIN2000 with 8 MIPS R10000 processors running at 195MHz, with 1280MB of main memory, and a network bandwidth of 800MBs/sec. Six different MPI implementations have been compared:

1. *FFT* is the basic implementation with MPI.
2. *FFT-Residual*. This program differs from *FFT* in that all *send-a-request* and *receive-a-request* transactions are performed at the partial evaluation step. Hence, they are not included in the residual program. Each element of input vector has a

vector of deferred reads. The residual program only binds elements, completes pending requests, and executes *send-a-value* and *receive-a-value* transactions.

3. *FFT-DIS.* Remote requests are managed by the DIS memory system.
4. *FFT-DIS-Residual.* A residual program differs from the original *FFT-DIS* program in that all requests for IS data items, local or remote, and *receive-a-request* operations are performed during the partial evaluation step.
5. *FFT-DISSC.* The DISSC system is used.
6. *FFT-DISSC-Residual.* Each element of the input vector has a vector of deferred reads. The residual program only binds elements, completes pending requests, and executes *send-a-value* and *receive-a-value* transactions. To support a cache line mechanism, the vector has one extra element which counts how many elements in a requested cache block have been produced.

3.1 Message Reduction by Caching Remote Memory Requests and Partial Evaluation

Table I shows the number of messages with a varying numbers of processors. DISSC does not reduce the number of messages when caching a single IS data item (CB=1). With caching 4 and 8 IS data items, the reduction of messages obtained by DISSC is respectively 4 and 8. This demonstrates that the FFT algorithm has no significant temporal data locality and re-use of data, and that only spatial locality is exploited. In the residual programs, the number of messages is reduced by a factor of two as compared to the original ones, irrespective of the number of PEs. Table 1 also shows how the DISSC contributes to the messages reduction. Increasing the size of the cache block proportionally decreases the number of messages, for example, the total reduction in the *FFT-DISSC*–Residual (CB=8) is 16 times comparing with the *FFT*.

It is important to note that a reduction in the number of messages not only diminishes the execution time of the program, but also improve the system behavior by reducing the saturation of the communication system.

Table I. Number of messages varying the number of processors.

MPI programs		2 PEs	4 PEs	8 PEs
FFT		16,384	40,960	81,920
FFT-Residual		8,192	20,480	40,960
FFT-DIS		16,384	40,960	81,920
FFT-DIS Residual		8,192	20,480	40,960
FFT-DISSC	CB=1	16,384	40,960	81,920
FFT-DISSC-Residual		8,192	20,480	40,960
FFT-DISSC	CB=4	4,096	10,240	20,480
FFT-DISSC-Residual		2,048	5,120	10,240
FFT-DISSC	CB=8	2,048	5,120	10,240
FFT-DISSC-Residual		1,024	2,560	5,120

3.2 Time Reduction by Caching Remote Memory Requests

Figure 1 shows the speedup for varying numbers of PEs. *FFT-DIS* has a lower speedup than *FFT* because of the DISs management overhead. *FFT-DISSC* with

CB=1 has a lower speedup than *FFT-DIS* because of the overhead and the lack of temporal data locality. Nevertheless, the spatial data locality exploited by the DISSC mechanism contributes to the acceleration of the *FFT* program. For instance, for eight PEs, the speedup varies from 1.19 to 4.39, varying CB from 1 to 8.

Figure 2 presents the relative time reduction of *FFT-DIS* and *FFT-DISSC*, with different cache block sizes over the original *FFT* program, varying the number of PEs. *FFT-DIS* and *FFT-DISSC* with CB=1 are not faster than *FFT*. The speedup is increased when CB=4, 8. For PEs=8, *FFT-DISSC* has a relative time reduction of 1.46 and 2.01, respectively. The time reduction is higher than the overhead of DISs and DISSC.

3.3 Time Optimization by Partial Evaluation

The degree of parallelizability of residual programs is presented in Figure 3. It shows speedups of residual programs with varying number of PEs. It shows that the speedup of *FFT-Residual, FFT-DIS-Residual* is relatively small, between 2 and 3 for 8 PEs. For the same number of PEs, the speedup of *FFT-DISSC-Residual* is increased from 2.1 to 5.35 varying CB from 1 to 8. To evaluate the impact of partial evaluation on the performance, a time optimization coefficient $S^o_p = T_o / T_r$ is calculated. S^o_p is the ratio of the execution time T_o taken by the original program over the time T_r taken by the residual one. Figure 4 shows the S^o_p for benchmark programs with and without cache system versus the number of processors.

FFT-Residual is 20-50% faster (depending number of PEs) than *FFT, FFT-DIS-Residual* is about 70% faster than the original *FFT-DIS* program. The time reduction for *FFT-DISSC-Residual* program is slightly larger. It is about 90% when CB=1 and 46% for CB=8. Increasing the CB reduces the number of messages and, hence, fewer messages are removed by partial evaluation.

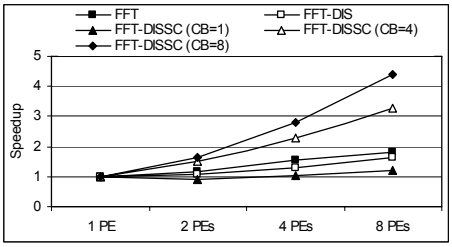

Fig. 1. Speedup of FFT, FFT-DIS and FFT-DISSC (with different cache block sizes 1, 4 and 8) varying number of Pes.

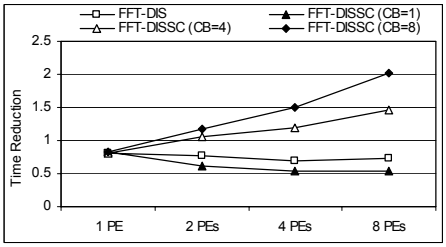

Fig. 2. Time reduction of *FFT-DIS* and *FFT-DISSC* (with different cache block sizes 1, 4 and 8) programs over *FFT*, when varying the number of PEs.

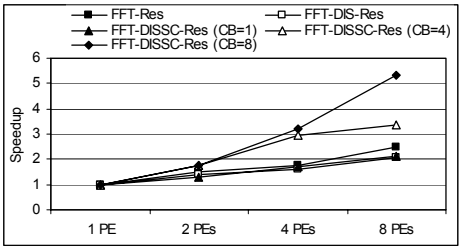

Fig. 3. Speedup of *FFT-Residual*, *FFT-DIS-Residua*l, and *FFT-DISSC-Residual* (with cache block sizes equal 1, 4 and 8) programs with different number of PEs.

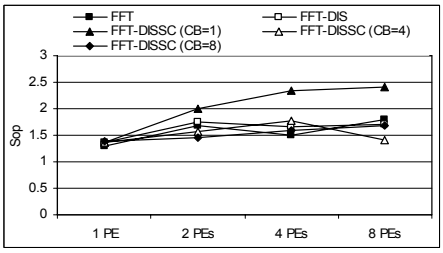

Fig. 4. S_{op} for benchmark programs with and without cache system versus the number of processors.

4. Conclusions

Many non-strict structures are known and have been experimentally evaluated for a variety of multithreaded shared memory systems. The problem is to assess how suited they are for the exploitation of parallelism in distributed memory systems which use the latency tolerance properties of MPI. In this paper, the design and experimental evaluation of parallel implementations of the FFT algorithm in MPI with DISs and DISSC have been presented. We have shown that the split-phase memory access scheme of MPI and DISs allows not only an overlap of long communication latencies with useful computations, but also lifts the main restriction which conventional (and sequential) information processing usually implies: complete production of data before its consumption. It makes the concept of partial evaluation of distributed programs on the communication level feasible. Partial evaluation allows a reduction in the number of messages in data-independent applications. It can also be applied to program optimization by constant propagation, loop unrolling, and polyvariant specialization in order to get fully advantage of the static information available.

We have shown that MPI programs using DIS and DISSC can take advantage of data locality and can allow complete asynchronous memory accesses. Although the management of D-IS has a cost, the DISSC overcomes this cost and improves the program performance by eliminating messages. Although ISs help with the synchronization issues, there are some algorithms where re-assignment is a key issue. In this paper, we have also presented experimental results which show the gain of partial evaluation of MPI programs without ISs.

We have shown that DISSC and partial evaluation are a good programming mechanism which optimize both distributed programs and the use of the parallel systems by avoiding the saturation of the interconnection network, thereby leaving the resources free for other applications. Experiments have shown that if one were to take the total number of messages in the original MPI program as 100%, then introducing DISSC, it would reduce up to 88% of messages. The number of messages that are left (12%) can be reduced twice by partial evaluation, which means that only 6% of the original messages are left. Comparing the execution time of all benchmark programs versus the execution time of an FFT program running in a single process, the total

execution time can then be reduced by a factor of 1.68, when CB=4, PEs=2; 2.38 when CB=8, PEs=4; and 3.37 when CB=8, PEs=8; with DISSC and partial evaluation together.

References

1. Arvind, Nikhil R.S., Pingali, K-K.: I-Structures: Data Structures for Parallel Computing. ACM Transaction on Programming Languages and Systems, Vol. 11 No. 4 (1989) 598-632
2. Böhm, A-P-W., Hiromoto, R-E.: The Data Flow Parallelism of FFT in Gao, G-R., Bic, L., Gaudiot, J-L.: Advanced topics in dataflow computing and multithreading ISBN: 0-8186-6542-4 (1995) 393-404
3. Chamberlain, R-M.: Gray codes, Fast Fourier Transforms and hypercubes. Parallel computing, vol. 6, (1988) 225-233
4. Cristobal A., Tchernykh A., Gaudiot J-L., Lin WY. Non-Strict Execution in Parallel and Distributed Computing, *International Journal of Parallel Programming*, Kluwer Academic Publishers, New York, U.S.A., vol. 31, 2, p. 77-105, 2003
5. Dennis, J-B., Gao, G-R.: On memory models and cache management for shared-memory multiprocessors. CSG MEMO 363, CSL, MIT. (1995)
6. Eicken, T., Culler, D-E., Goldstein, S-C., Schauser, K-E.: Active Messages: a Mechnisim for Integrated Communication and Computation. In Proceedings of the 19th International Symposium on Computer Architecture, 1992, 256-266
7. Ershov, A.P.: Mixed computation: potential applications and problems for study. Theoretical Computer Science, vol. 18 (1982)
8. Govindarajan R., Nemawarkar S, LeNir P: Design and performance evaluation of a multithreaded architecture. In proceedings of the 1st international symposium on High-Performance Computer Architecture, Raliegh, 1995, 298-307
9. Gluck R., Nakashige R., Zochling R.: Binding-time analysis applied to mathematical algorithms. In Dolezal J., Fidler, J. (eds.) 17th IFIP Conference on System Modelling and Optimization; Prague, Czech Republic. (1995)
10. Gupta S-A,: A typed approach to layered programming language design. Thesis proposal, Laboratory of computer science, Department of EE&CS, MIT (1993)
11. Jones, N-D.: An introduction to Partial Evaluation. ACM computing surveys, Vol 28, No 3 (1996)
12. Kavi, K-M., Hurson, A-R., Patadia P., Abraham E., Shanmugam, P.: Design of cache memories for multithreaded dataflow architecture. In ISCA 1995, 253-264
13. Lawall, J-L.: Faster Fourier Transforms via automatic program specialization. IRISA research reports 1998; 28 pp.
14. Lin, W-Y., Gaudiot, J-L.: I-Structure Software Cache – A split-Phase Transaction runtime cache system. In: Proceedings of PACT '96 Boston, MA, 1996, 20-23
15. Ogawa, H., Matsuoka, S.: OMPI: Optimizing MPI programs using Partial Evaluation. In Proceedings IEEE/ACM Supercomputing Conference (1996)
16. Osamu, T., Yuetsu, K., Santoshi, S., Yoshinori, Y.: Highly efficient implementation of MPI point-to-point communication using remote memory operations. In: Proceedings of 12th ACM ICS98, Melbourne, Australia. (1998) 267-273

17. Quinn, M-J.: Parallel computing theory and practice. McGraw-Hill Inc. (1994)
18. Sperber, M., Klaeren, H., Thiemann P.: Distributed partial evaluation. In: Kaltofen, Erich (ed.): PASCO'97, Maui, Hawaii. (1997) 80-87
19. Swarztrauber, P-N.: Multiprocessor FFTs. Parallel computing, vol. 5. (1987) 197-210.
20. E. Oran Brigham. Fast Fourier Transform and Its Applications, Prentice-Hall, 1988
21. J.N. Amaral, W-Y. Lin, J-L. Gaudiot, and G.R. Gao, "Exploiting Locality in Single Assignment Data Structures Updated Through Split-Phase Transactions," *International Journal of Cluster Computing, Special Issue on Internet Scalability: Advances in Parallel, Distributed, and Mobile Systems*, Vol. 4, Issue 4, 2001.