Fortran D

- Final Report -

Compilation Techniques for Parallel Processors
ECE 1754

Professor Abdelrahman

Friday May 12, 2000.

David Tam

1 INTRODUCTION

Fortran D is an extension of Fortran 77 to allow for efficient data-parallel programming. It has language extensions to allow the programmer to specify how data should be divided among nodes of a distributed-memory machine (DMM). This task is known as user specified data partitioning. Once the data partitioning is specified, the Fortran D compiler automates the rest of the process to create efficient parallel executable code. It handles the partitioning of the computation based on the data partitioning that was specified by the programmer. Explicit communication, in the form of message passing, is generated by the compiler. The compiler handles the tasks of optimizing this communication and exploiting available parallelism by efficiently dividing computation among the nodes. Finally, the data is distributed according to the specified data partitioning and a single SPMD (single program multiple data) executable is distributed to all nodes of the DMM.

Although the Fortran D compiler system is nearly a decade old, it has contributed important research on compiler optimizations for distributed-memory machines (DMMs). The optimization techniques developed are applicable in general to parallel machines and languages. For instance, the work done on the Fortran D compiler has been used in conjunction with automatic data partitioning schemes [15]. The automatic data partitioning

algorithms can be added to the front-end of the Fortran D compiler system, while the back-end remains essentially unchanged. This capability shows the modularity and applicability of the compiler system. As well, the compiler system has been influential in the development of the High Performance Fortran (HPF) specification [8] [19]. HPF also leaves data partitioning to the programmer and performs many of the same back-end tasks and optimizations as Fortran D.

Fortran D has a heavy emphasis on compile-time optimizations, as opposed to slower run-time optimizations. As well, optimizations are based around loops rather than procedure calls.

1.1 Relation to Other Areas

Fortran D relies heavily on analysis techniques such as dependence, control flow, and symbolic analysis. Loop transformations are relied upon to maximize the exposure of parallelism to the compiler. The general ideas of loop scheduling are incorporated into the automatic computation partitioning phase of the compiler. It performs a static scheduling based on the "owner computes" rule. Fortran D uses the "owner computes" convention in dealing with data locality. However, loop scheduling, as studied in the lectures, was targeted for shared-memory systems.

1.2 Main Problems

The main problems that the Fortran D compiler system deals with are the creation of a machine-independent programming language, data partitioning, computation partitioning, static and dynamic partitionings depending on the nature of the code, communication optimization, and exploiting parallelism.

In designing a language that is machine-independent, the programming model is as follows. The programmer writes code under the traditional sequential programming model. Next, the programmer determines how the data should be divided in general based on insight gained from first-hand experience in designing the algorithm and implementing code. Partitioning the data is specified by using the **`DECOMPOSITION`**, **`ALIGN`**, and **`DISTRIBUTE`** operatives. The programmer does not need to worry about the number of nodes on the DMM or the details of the DMM architecture.

1.3 Report Outline

Section 2 of the report explores a number of issues in the design of Fortran D. The collection of issues presented is fairly eclectic and loosely-coupled. After gathering a large stack of related papers and attempting to read (and understand) all of them (a nearly impossible task), these were the

issues that came to mind: critical design decisions, review of optimization results, scalability of optimizations, analysis of optimizations, additional optimizations, training sets, dynamic partitioning, storage management, pipelining, owner-computes, distribution flexibility, load balancing, operating system interaction, programming model, compile-time emphasis, classes of scientific problems, new technologies, and comparison to HPF.

2 MAIN

2.1 Critical Design Decisions

A critical decision made by the designers was that data partitioning be specified first. Computation partitioning is automatically derived from the first step. Another critical decision was that the owner computes convention be the basis for computation partitioning. The designers decided to compromise on flexibility rather than performance by heavily emphasizing compile-time optimizations rather than relying more on run-time support. This design decision was made in [11] after observing the drawbacks of the first generation of data parallel programming languages, such as the Superb system [2].

2.2 Review of Optimization Results

Various communication optimizations have been developed to hide startup, copy, and transit times in the message-passing system [12]. Reducing startup times relies upon the techniques of message vectorization, message coalescing, and message aggregation. Hiding copy times relies upon nonblocking message mechanisms. Hiding transit times relies upon message pipelining, vector message pipelining, and iteration reordering. Pipelined computations are handled effectively using fine-grain or coarse-grain pipelining. Collective communication, reductions, and scans are applied as well. The benefits of reductions and scans increase with increased problem and system size.

The applicability of each optimization depends upon various factors. However, message vectorization has been shown to be the most important optimization [12]. In effect, this technique gathers a series of individual messages destined for a particular distributed array and a particular node, and makes a single request instead. This drastically reduces the startup overhead. When pipelined computations are present, coarse-grain pipelining optimizations work best compared to fine-grain pipelining. Again, this result is mainly due to communication overhead factors.

As a general guideline as to which optimizations should be applied, the following observations have been made [12]. When communication

startup costs are high, message vectorization is more effective than message pipelining. Since message pipelining attempts to communicate results as soon as they are produced, it will naturally ignore buffering attempts that are inherent in message vectorization. Buffering amortizes the startup cost of sending one message among several messages.

Message vectorization can sometimes lead to unnecessary communications. Similar to the problem experienced in loop-invariant code motion, loops or certain iterations of a loop may not be executed. Due to message vectorization, the remote data has already been retrieved but is no longer needed.

Message aggregation is more effective than vectorization when messages come from a series of different arrays but are destined for the same node. Naturally, choosing to use aggregation will exploit the phenomenon and result in better optimizations. Using vectorization would require buffering for each of the different arrays and may prove to be slower.

In the process of transmitting a message to another node, the message must be copied from the application space to the operating system kernel space before sent out onto the network. Traditional messaging causes the application to block until the copying from application space to

kernel space is complete. On the other hand, nonblocking messages return to the application as soon as the copying begins, performing the message copying concurrently with application execution. The use of nonblocking messages is effective only when the increase in startup time is less than the copy time. As well, there must be sufficient local computation to hide the copy time. If the increase in startup time is equal to or greater than the copy cost, the program might as well use the traditional blocked message mechanism and eliminate the overly expensive startup time.

Iteration reordering requires the difficult task of predicting the amount of increased computation due to reordering. Under blocked message implementations, this optimization can hide any remaining transit time that is not hidden by the vector message pipelining technique. Under nonblocking message implementations, it can hide copy times as well.

In general, reductions and scans should always be applied. Pipelining is an effective and scalable optimization. Pipelining requires the running of training sets to obtain parameter values that are used in calculating optimal block size in coarse-grain pipelining. Dynamic decomposition is suitable only for problems that contain small amounts of distributed data, since redistribution of this data can be very expensive. Message vectorization remains effective even when the amount of communication increases.

2.3 Scalability of Optimizations

Results from Fortran D papers [12] [13] indicate that, for a fixed number of nodes, as the amount of communication increases, message vectorization scales better than message pipelining. There is relatively less local computation available to hide the increase in communication under the message pipelining scheme. Message vectorization eliminates entire messages in an effective manner. However, as the number of nodes increases, collective communication such as broadcasting scale well while others such as message vectorization do not. With broadcasting, the startup and copy costs of communicating a segment of data are amortized over a larger and larger number of nodes.

In terms of reducing execution times in comparison against sequential execution, the effectiveness of fine-grain pipelining, coarse-grain pipelining, and regular send/receive mechanisms increase as problem size increases. However, communication optimizations become less effective.

The general rule is that the computation to communication ratio will indicate whether communication or parallelization optimizations are the most effective. The ratio of problem size (in terms of the number of elements) to number of nodes will determine whether communication or computation dominates. For a fixed problem size, as the number of nodes increases, communication will eventually dominate. This is because each

node has less and less local computation to perform since computation is divided among more nodes. For a fixed number of nodes, as the problem size increases, computation will dominate. Each node must perform more and more local computations since computation is divided among less and less nodes.

2.4 Analysis of Optimizations

Most communication optimizations attempt to simply move time that must be spent on an operation from one location to another. Communication operations require a certain minimal amount of time that cannot be reduced. These optimizations attempt to overlap these segments of time. The underlying time segments are from computations performed locally on each node. However, when these time segments are shorter than other time segments which are trying to overlap them, the communication optimizations become less effective. The length of these underlying time segments depend upon the nature of the code. Since the Fortran D compiler system was initially targeted for a certain class of scientific computations, these communication optimizations have proven to be effective. This class of programs usually have an alternating pattern of local computation phase followed by a global synchronization and communication phase. Programs that do not fit this class of behaviour may not be optimized effectively.

The whole process of deriving optimizations must be done carefully.

Some optimizations may only be patch solutions and not widely applicable. Since the optimizations are based on studying well-known benchmarks [12] [23], and these benchmarks are a representative sample of the behaviour of a wide class of programs, these optimizations should have wide applicability. However, a number of papers have been written that introduce new optimizations for additional classes of problems that were not initially considered [9] [13] [18].

2.5 Additional Optimizations

Additional analysis such as interprocedural and interloop analysis can expose further opportunities to apply the various communication optimizations [7]. Array kill analysis can eliminate unnecessary communication and computations. For instance, code in the form of:

```
A[1] = ...
...
..
.
A[1] = ...
```

can be optimized if, for a particular node, the reference **A[1]** is remote and there are no reads of **A[1]** in between by any nodes, then the first line can be eliminated. This also eliminates the communication of rhs references to the owner of **A[1]** due to the owner computes rule.

Statement grouping can aid in the partitioning of computation [13]. It basically groups statements that have common conditions and reduces the

number of explicit guards inserted to perform run-time checking.

Interloop analysis can allow for message coalescing and aggregation to be applied across loop nests, reducing communication overhead.

For multi-dimensional arrays which reduction is applicable, there are opportunities to improve efficiency by applying the technique of multi-reductions [13]. This technique imposes a direction upon the reduction of multi-dimensional arrays. This leads to data transfers in a predetermined direction across a predetermined dimension. This allows the compiler to determine when communication is necessary and apply optimizations. It also allows a problem to be partitioned in other dimensions so that no communication is required in some cases.

2.6 Training Sets

Training sets are typically run on a system to determine values for various parameters such as message startup, transit, and copy times [1]. These training sets may not be an accurate reflection of parameters during the execution of the target code. The training sets may not account for multiprogramming workload interference, program self-interference, and various operating system factors. As well, since averages are used, variance may need to be examined as well. Ideally, the parameters should be random variables with distributions that reflect those obtained by the training sets.

Using training sets provides a way of obtaining static performance parameters that are important in deciding which optimizations to implement. In general there are issues in using static versus dynamic performance estimates. The use of static methods means that there is no feedback mechanism in the system to compensate for bad optimization decisions, or unanticipated situations. The use of run-time checks may account for some of the changes but cannot do very much to correct the problems. The heavy emphasis on compile-time analysis and optimizations naturally leads to the use of static performance estimation. Dynamic estimates may have a lot of overhead, deteriorating performance.

Dynamic information could be useful in coarse-grain pipelining to adjust block sizes dynamically. However, this would require a number of run-time checks to be inserted in various locations to trigger the new block size implementation.

New developments in technology show that a number of built-in performance monitoring facilities are becoming available. For instance, many processors now include high-resolution counters that can be used for a variety of purposes. The overhead of dynamic performance estimation may be less than previously thought.

2.7 Dynamic Partitioning

The applicability of the optimizations to dynamic partitioning schemes is fairly good. As long as the nature of the dynamic partition is known at compile-time, the Fortran D system will handle it fairly well. However, this feature usually means the compiler will be inserting run-time checks to determine which data partitioning scheme is in effect. These run-time checks have an detrimental effect on performance.

The support of dynamic partitioning causes a number of complications and problems. Reaching decomposition analysis must be performed and run-time checks may need to be added if the partitioning remains ambiguous after compile-time analysis. These checks add overhead to the executable and decrease the performance of the DMM system. An alternate approach to run-time checks is node splitting, which restores compile-time analysis. The original basic block containing the accessed array is split into two. This causes changes to the control flow and data flow graphs which are used in various analysis stages.

Avoiding dynamic partitioning is a wise decision for all but the smallest programs [12]. This is because the cost of redistribution of data among the nodes is very high. In some very rare cases, it may be possible to apply the ideas of the message pipelining technique to overlap data redistribution with other activities well ahead of the actual switch-over

point in a dynamic partitioning scenario. In most cases, this optimization can prove to be difficult, as data can be altered up until the switch-over point. In another possibility, which may be quite rare, immediately before the cross-over point, all variables on the right hand side (rhs) may exist on a particular node A. The left hand side (lhs) belongs to a remote node B. Immediately after the cross-over point, the lhs variable is redistributed to node A. This scenario presents an obvious opportunity for optimization, however, it is a severely contrived example.

2.8 Storage Management

There are three schemes of storage management [11] [10]. They are overlaps, buffers, and hash tables. Overlaps expand the local array sections to hold additional elements. The advantage of this technique is the clean code it generates. The disadvantage is that it is specific for each array and may require more storage than available.

For instance, the local section of a distributed array A[] holds elements in indices 10 to 20. If the local node requires elements in indices 9 and 21 to perform computation, then overlaps are added. The local array is expanded on the left and the right so that it will range from indices to 9 to 21.

Buffers avoid contiguous and permanent storage. It is useful when

storage space must be reused (such as due to memory constraints), and when the remote array section is not near the local array section. For instance, array A requires elements from remote indices 100 and 200 in order to perform computation instead of 9 and 21.

Hash tables are used when the set of remote elements accessed is quite sparse, such as in irregular computations. The main advantage of this method is its quick look-up mechanism.

2.9 Pipelining

In previous papers, pipelining was shown to be very effective in parallelizing computational wavefront programs [12] [23]. These types of programs cause a dependence-based sequential access of a distributed array. This linear sweeping motion of dependencies can be transformed into a collection of parallel sweeps along a number of segments, with a single initial dependency at the beginning of each segment. Strip mining and loop interchange are used to transform the code.

In general, course-grain pipelining was considered most applicable, compared against fine-grain pipelining. Choosing the optimal block size is based on the formula sqrt(block communication cost / element computation cost). For NUMA systems, determining block communication costs can be fairly difficult. It may also lead to variable block sizes since communication

costs from one node to another will vary, which causes further complications. For heterogeneous nodes, determining computation costs will be difficult since each node may have a different computation cost.

2.10 Owner-Computes

A fundamental characteristic that affected the applicability of the optimizations was the use of the owner computes convention. This convention determined how the computation was to be partitioned. This convention allowed some important assumptions to be made. The owner computes rule solved the problem of the owner having the most up to date copy of a variable. However, it was seen that this convention was sometimes too restrictive and prevented some optimizations. Relaxation of this convention was applied in a number of situations, such as on private variables.

An alternative convention to the owner-computes rule is the almost-owner-computes rule [21]. In this alternative, all right--hand side (rhs) variables are examined. The node that owns the most variables performs the computation. This convention could relieve some of the redistribution tendencies experienced by the compiler. A programmer who is aware of this convention would know to leave the distributed data alone under some situations rather than specify a redistribution of data.

Fortran D provides the "**ON**" clause to optionally specify which processor should perform the computation [21]. However, this seems to be a very primitive and tedious way of optimizing code. The programmer may need to specify **ON** clauses in a number of locations in the code and must also determine which node is best to perform the computation, contradicting the goal of machine-independent code. These issues should be abstracted away from the programmer and automatically handled.

2.11 Distribution Flexibility

The three basic distribution concepts of **BLOCK**, **CYCLIC**, and **BLOCK_CYCLIC** are supported along with a number of variations. These variations are specified in the form such as **BLOCK(4)**, where the parameter **4** in a block distribution specifies the number of nodes involved. For instance, if this distribution is specified on an 8 node DMM, only the first 4 nodes will be utilized. For cyclic distributions such as **CYCLIC(5)**, the parameter **5** specifies how many times to cycle through all nodes during distribution. On a 4 node DMM, for an array that has 40 elements with a **CYCLIC(5)** distribution, each node is given 2 elements at a time. For block cyclic distributions such as **BLOCK_CYCLIC(3)**, the parameter **3** specifies that 3 elements are to be grouped for each block. Various possible distributions, as described in [6] are shown in Figure 1.

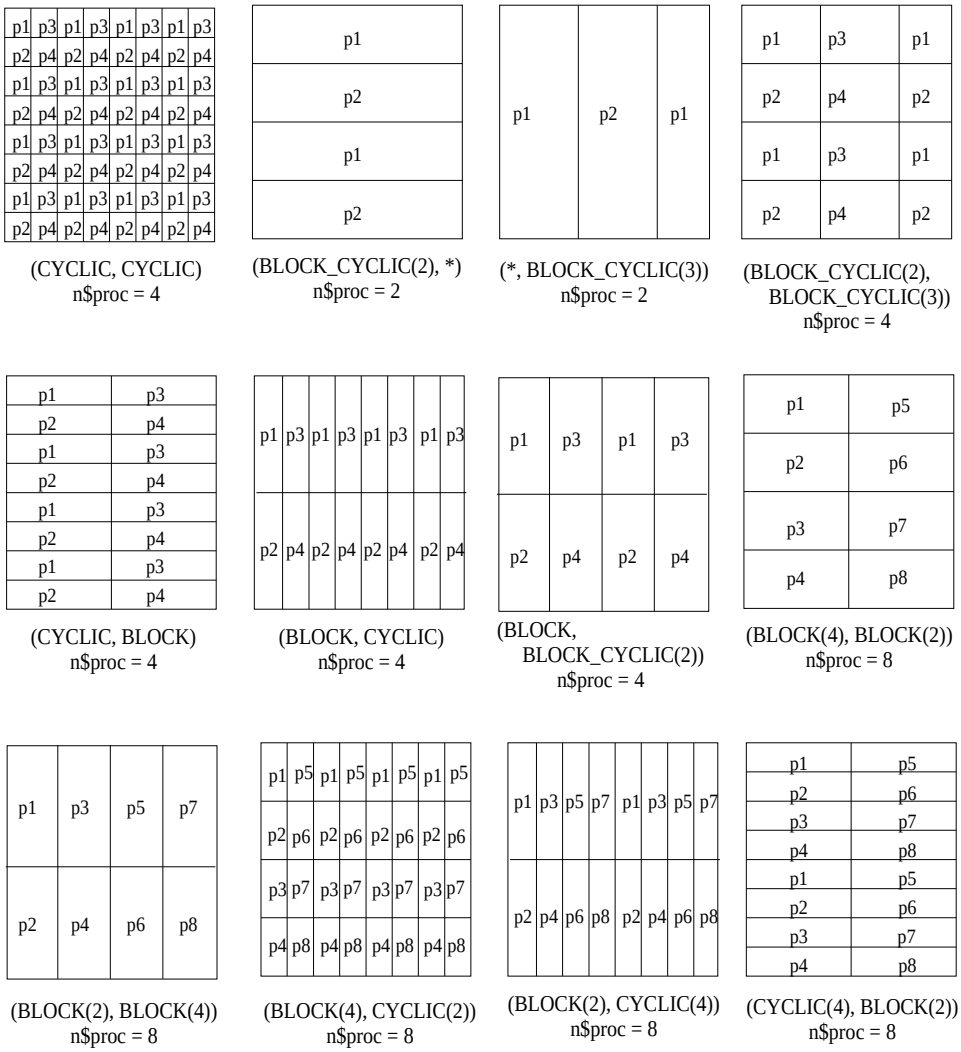

Figure 1 - Examples of distributions

2.12 Load Balancing

From the phases of compilation system as presented in various Fortran D papers [11] [12] 23], there appears to be no handling of load balancing. Data partitioning will lead to the initial distribution of load. However, there are no feedback mechanisms to handle imbalances that could occur. The owner computes convention is a fundamental inhibitor of load balancing since computation cannot simply migrate to an idle node.

At first glance, data-parallel problems may appear to exhibit a natural load balancing tendency. It appears that the same general operation is applied uniformly on all data elements. However, when this is actually implemented, some locations of the distributed data may require more resources to obtain a result [20]. Fortran D does not appear to handle such cases.

2.13 Operating System Interaction

The role of the operating system (OS) has been largely ignored in developing the optimizations. It is only partially accounted for by the estimation of startup and copy times in message transmission. Useful hardware facilities have also been ignored. Perhaps certain facilities provide a drastic improvement in performance, or make certain optimizations extremely effective. Ideally, the role of the compiler,

operating system, and hardware should all be considered in order to achieve optimal performance. Most compiler systems seem to ignore these other aspects and believe all problems can be solved by the compiler rather than from the cooperation of hardware, operating system, and compiler. They exhibit the behaviour best described by the saying, "given a hammer, all problems appear to be nails". Accounting for useful facilities, as well as identifying interference situations could be helpful. Additional barriers to performance may be identified during this process.

In general, the Fortran D compiler does not address operating system issues such as the effect of context switches and pollution of the cache by the OS or other processes. It appears to assume a batch processing OS. It ignores multiprogramming environments. It ignores the effects of I/O and paging. For instance, if secondary storage or retrieval is required, locality of disks should be accounted for as well.

2.14 Programming Model

The applicability of the programming model adopted by Fortran D is debatable. It is based on the assumption that the programmer knows best. The programmer knows how to partition the data most efficiently. However, there have been a number of attempts to automate this data partitioning process. As well, data partitioning assistant tools have been developed and integrated in the Parascope Editor and D Editor [4] [14]. These

programming environments allow the programmer to obtain rough performance estimates based on the partitioning scheme chosen.

The need for such tools indicates some kind of underlying weakness in the programming model. Determining the best data partitioning is perhaps more difficult than expected. Development of these partially automated tools seems to foreshadow the usefulness of completely automatic data partitioning. It also contributes fundamental research into the automation of data partitioning. Automatic data partition is beyond the scope of this report.

The expressiveness of this model is also a debatable issue [6]. The high level abstractions presented by Fortran D could potentially be an inhibitor of performance. Certain problems and operations may not be efficiently expressed leading to systemic bottlenecks. Fortran D does provide irregular data partitioning through alignment maps. This fine-grained data partitioning mechanism provides a lot of flexibility and can handle more problems that have irregular data access patterns.

Parallel sections of code cannot be explicitly specified in the Fortran D language. This may be a disadvantage for code implemented directly in Fortran D from the start. A programmer who knows for certain that a section code can be executed concurrently cannot directly specify it. There

is an implied parallelism based on the user-specified data partitioning. However, this lack of an explicit parallel directive is advantageous when porting an existing sequential program to the Fortran D language and DMM parallel system since only a few additions need to be made to efficiently parallelize the code. Actual alteration of existing lines in the code is not needed.

2.15 Compile-Time Emphasis

A scenario where the emphasis of compile-time analysis and optimization can be problematic is as follows. A large DMM system requires users to reserve time-slots in order to run their computations and simulations and are charged for this usage. However, the number of nodes that are available at any particular time is unpredictable. Rather than spend time during the allotted time slot to compile code, users would prefer to perform the compilation ahead of time on their own computers and simply transfer the executable to the DMM. Users of such a system would prefer the number of nodes to be a run-time specification rather than compile-time specification.

This issue has been addressed in [9]. However, the solution in general limits the effectiveness of compile-analysis and introduces run-time overhead. There are drawbacks to such run-time support. Run-time guards must be inserted at many locations of code, and certain optimizations

cannot be performed. These problems will cause a decrease in performance. Naturally, the increase in time must be compared against the compilation time to determine which choice to select.

The trade-off between compile-time and run-time optimizations is the issue of speed versus flexibility. Compile-time optimizations give less flexibility but performs much deeper analysis and lead to better optimizations. Fortran D has very few language extensions and thereby has fewer opportunities to insert ambiguous code that requires run-time support.

A disadvantage of compile-time analysis is that there is typically difficulty in determining indirect variable references. However, run-time support can easily determine these references. Perhaps more run-time optimizations should be added. However, these optimizations will increase overhead so they must be weighted carefully against the benefits.

Additional run-time support can be provided in areas that are important to added program flexibility. These sections of code should encounter infrequent execution and should not be on the critical path of the program. Again, there is a trade-off between performance and flexibility that must be balanced.

2.16 Classes of Scientific Problems

In general, there are roughly three classes of scientific problems. One class exhibits the behaviour of alternating local computation with global synchronization and communication. These cycles of compute, communicate, compute are ideal problems to be handled by compiler systems. Another class is loop-carried cross-processor data dependent computations. These usually traverse a distributed array in a serial fashion and exhibit dependency characteristics that make it appear impossible to parallelize. However, various loop transformation techniques can eliminate these dependencies and expose parallelization opportunities. Fine-grain or coarse-grain pipelining techniques can be applied to achieve parallelization. A final class of computations are those that are irregular in nature. They access data in irregular patterns. These are more difficult to optimize. A number of papers have dealt with these types of problems [2] [21] [22].

A potential solution relies on the ability to specify data partition at a fine-grain level. If the programmer understands the access irregularities, these could be accounted for during data partitioning.

The optimizations do not take into account that for some classes of problems, the amount of data may start off small, but slowly grow. For instance, dynamic programming problems exhibit this behaviour. As well, programs that retain and alter historic values may exhibit this behaviour.

Optimizations may not take these factors into account. Since most of the optimizations are compile-time based, they have a tendency to fix a particular optimization to a particular region of code. In some cases, run-time guards could be used to trigger a different optimization at a particular point in the program. For instance, when the amount of data in a distributed array is small, message aggregation may be chosen. When the amount of data in a distributed array grows larger, message vectorization or message coalescing may be chosen instead.

2.17 New Technologies

It has been close to a decade since the Fortran D compiler system was developed. Since that time, new technologies have been developed. New hardware and operating system technologies have been developed and are in use. The Fortran D system may be less applicable to these new systems. For instance, shared-memory systems are not addressed by the Fortran D system even though they existed before the compiler system was conceived. NUMA systems have not been accounted for in the optimizations. This characteristic may make it very difficult to determine which communication optimizations to apply. The transit times will vary greatly depending on which node is local and which is remote. Transit time is no longer uniform. Case by case optimizations for each partition and each node may be required. An explosion in dimensionality is evident. A partial solution relies on the fine-grain specifications in data distribution provide

by Fortran D. A programmer would first need to account for the underlying system architecture before carefully specifying the data partition. This is only a partial solution since the programmer does not have full control over exactly which node data will be placed. As well, optimizations performed by the compiler will still not be aware of the NUMAness of the system. Unfortunately, this solution leads to architecture dependent code and defeats one of the main goals of Fortran D.

Another new trend is the use of heterogeneous nodes, such as in clusters, or the internet in general. This makes the calculation of computation costs fairly difficult, similar to NUMA systems.

The use of nodes that are symmetric multiprocessors (SMPs) is not addressed by the Fortran D compiler system. A simple approach would be to treat the entire SMP node as a single processor node. However, this approach would under-utilize the SMP node and would not achieve the full potential of the system. Another possibility is to account for the SMPs as a more powerful node. Effectively, this would could be seen as the treatment of a heterogeneous system as mentioned previously. An even better approach would be to treat the system as a NUMA system. Some nodes are closer together than others. For instance, each processor on the SMP node is treated as a separate node. When analyzing for optimization, the transit times between nodes on the same SMP are relatively small.

2.18 Comparison to HPF

HPF uses the align and distribute statements that are common with Fortran D. However, the extensions also provide a way to specify that data be stored in multiple locations to improve read access [8] [19]. Updates become more difficult under this scheme. HPF also supports explicit parallel directives unlike Fortran D.

3 CONCLUSIONS

The back-end optimization techniques developed in the Fortran D compiler system have been valuable to parallelizing compiler research. Fortran D was designed mainly for problems that have inherent data parallelism. There is a heavy emphasis on compile-time rather and run-time optimizations. Such an emphasis allows deeper analysis and highly effective optimizations to be performed. However, it also means there is less flexibility in the execution environment.

Fortran D, like most other compiler systems, fails to seriously consider the effects of the operating system and additional hardware support. Accounting for all three areas may lead to further optimizations.

Fortran D was developed for a DMM hardware nearly a decade ago

and therefore does not accommodate heterogeneous DMMs, NUMAness of nodes, and SMP nodes. Support could be provided, however focus has shifted to the HPF specification, which incorporates much of Fortran D. Any future research will be implemented on HPF compiler systems instead.

REFERENCES

[1] V. Balasundaram, G. Fox, K. Kennedy, U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, April 1991, pp. 213-223.

[2] P. Brezany, M. Gerndt, V. Sipkova, H.P. Zima, "SUPERB Support for Irregular Scientific Computations," IEEE Proceedings to the Scalable High Performance Computing Conference, April 1992, pp. 314-312.

[3] S. Chatterjee, G.E. Blelloch, M. Zagha, "Scan Primitives for Vector Computers," IEEE Proceedings of Supercomputing 1990, pp. 666-675.

[4] K.D. Cooper, M.W. Hall, R.T. Hood, K. Kennedy, K.S. McKinley, J.M. Mellor-Crummey, L. Torczon, S.K. Warren, "The ParaScope Parallel Programming Environment," Proceedings of the IEEE, Vol. 81, No. 2, February 1993, pp. 244-263.

[5] A.L. Fisher and A.M. Ghuloum, "Parallelizing Complex Scans and Reductions," Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, June 1995, pp. 135-146.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu, "Fortran D Language Specification," Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[7] M.W. Hall, S. Hiranandani, K. Kennedy, C.-W. Tseng, "Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines," IEEE Proceedings of Supercomputing 1992, pp. 522-534.

[8] High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 2.0, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1997.

[9] S. Hiranandani, K. Kennedy, J.M. Crummy, A. Sethi, "Advanced Compilation Techniques for Fortran D," Technical Report CRPC-TR93338, Center for Research on Parallel Computation, Rice University, October 1993.

[10] S. Hiranandani, K. Kennedy, C-.W. Tseng, "Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," ACM Proceedings of Supercomputing 1991, pp. 86-100.

[11] S. Hiranandani, K. Kennedy, C.-W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," Communications of the ACM, August 1992, Vol. 35, No. 8, pp. 66-80.

[12] S. Hiranandani, K. Kennedy, C.-W. Tseng, "Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," ACM Proceedings of Supercomputing 1992, pp. 1-14.

[13] S. Hiranandani, K. Kennedy, C-.W. Tseng, "Preliminary Experiences With the Fortran D Compiler," ACM Proceedings to Supercomputing 1993, pp. 338-350.

[14] S. Hiranandani, K. Kennedy, C.-W. Tseng, S. Warren, "The D Editor: A New Interactive Parallel Programming Tool," IEEE Proceedings of Supercomputing 1994, pp. 733-742.

[15] K. Kennedy and U. Kremer, "Initial Framework for Automatic Data Layout in Fortran D: A Short Update on a Case Study," Technical Report CRPC-TR93324-S, Center for Research on Parallel Computation, Rice University, July 1993.

[16] C. Koelbel and P. Mehrotra, "Programming Data Parallel Algorithms on Distributed Memory Machines Using Kali," ACM Proceedings of Supercomputing 1991, pp. 414-423.

[17] J. Li and M. Chen, "Compiling Communication-Efficient Programs for Massively Parallel Machines," IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 3, July 1991, pp. 361-376.

[18] L.M. Liebrock and K. Kennedy, "Parallelization of Linearized

Applications in Fortran D," IEEE Proceedings to the 8th International Parallel Processing Symposium, April 1994, pp. 51-60.

[19] D.B. Loveman, "High Performance Fortran," IEEE Parallel & Distributed Technology: Systems & Applications, Vol. 1, No. 1, February 1993, pp. 25-42.

[20] C.M. Pancake and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?" IEEE Computer, Vol. 23, No. 12, December 1990, pp. 13-23.

[21] R. Ponnusamy, Y-.S. Hwang, R. Das, J.H. Saltz, A. Choudhary, G. Fox, "Supporting Irregular Distributions Using Data-Parallel Languages," IEEE Parallel & Distributed Technology: Systems & Applications, Vol. 3, No. 1, Spring 1995, pp. 12-24.

[22] S.D. Sharma, R. Ponnusamy, B. Moon, Y-.S Hwang, R. Das, J. Saltz, "Run-time and Compile-time Support for Adaptive Irregular Problems," IEEE Proceedings of Supercomputing 1994, pp. 97-106.

[23] C. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines," Ph.D. Thesis, Department of Computer Science, Rice University, January 1993.

[24] H.P. Zima, "High-Performance Languages for Parallel Computing," IEEE Computational Science and Engineering, Fall 1996, pp. 63 - 65.