ハイパフォーマンス
コンピューティング 52－5
（1994．7．22）

VPP Fortranを用いたNAS Parallel Benchmarkの並列化とAP1000を用いた評価

金城ショーン　進藤達也
(株)富士通研究所 並列処理研究センター

本稿では，NAS Parallel Benchmarkを並列化した経験を基に，AP1000版VPP Fortranの記述能力と性能に関する評価を行なう．今回行なった並列化方式について概略を述べ，ベンチマークプログラムの実行時間結果を報告する．まとめとして，VPP Fortranの特徴について考察し，その記述能力柔軟性を大きく改善するための言語の改善案について述べる．

NAS Parallel Benchmark Implementation and Evaluation Using VPP-Fortran on the AP1000

Shaun Kaneshiro　Tatsuya Shindo
Fujitsu Parallel Computing Research Center
Fujitsu Laboratories Ltd., Kawasaki, Japan

This study evaluates the usability and performance of the VPP-Fortran language–Fujitsu's parallel Fortran dialect–based on our experience in parallelizing and tuning the NAS Parallel Benchmark programs for the AP1000 parallel computer. We outline the parallelization scheme used in this study, and report the execution time results for the benchmark programs. We conclude the paper by summarizing our observations on the VPP-Fortran features, as well as, language enhancements which would greatly improve its usability and flexibility.

1 Introduction

The VPP-Fortran language was first made available for the AP1000 parallel computer in January 1994. This study evaluates the usability and performance of the VPP-Fortran language based on our experience in parallelizing and tuning the NAS Parallel Benchmark programs for the AP1000 parallel computer.

The paper is organized as follows. First, an overview of the VPP-Fortran language, the AP1000 parallel computer, and the NAS Parallel Benchmark is presented. Next, we outline the parallelization process applied in this study, and report the benchmark execution time results. Finally, we summarize the lessons learned from this study from a programmer's point of view followed by a discussion of enhancements to the language.

2 Background

2.1 VPP-Fortran Language

In this study we port the NAS benchmarks to VPP-Fortran, Fujitsu's parallel Fortran dialect, which is based on the FORTRAN77 standard [2][3]. The language is designed for data parallel computation, providing the user with directives for partitioning and distributing loops and arrays in block or cyclic manner, and performing barrier synchronization and global operations across nodes within a region. VPP-Fortran also includes several features for performance tuning to aggregate data communication, and to eliminate unnecessary global address calculation and barrier synchronization [4]. We take advantage of these features in the parallelization scheme applied in this study.

2.2 AP1000 Parallel Computer

The AP1000 is a massively-parallel message-passing MIMD computer. Each node is based on a 25MHz SPARC processor with custom-designed modules for memory and network controllers, and 16MB of memory. The nodes, arranged in a two-dimensional torus configuration, can communicate via three different networks: the T-net, a two-dimensional point-to-point torus network, B-net, a hierarchical broadcast network, and S-net, a binary tree synchronization network. A single configuration can have up to 1024 processors.

2.3 NAS Parallel Benchmark

The Numerical Aerodynamic Simulation (NAS) Parallel Benchmark [1] is a collection of five kernels and three applications common to computational fluid dynamic (CFD) applications. The benchmark was primarily developed to compare the performance of highly-parallel, distributed-memory computers. This study includes the parallelization of five kernel programs and two of the CFD applications in the version 4.2 benchmark:

Embarrassingly Parallel (EP) generates 2^{28} pseudo-random numbers, and characterizes the generated numbers.

Multigrid (MG) computes approximations to a discreet Poisson problem on a 256 × 256 × 256 grid using four iterations of the V-cycle multigrid method with periodic boundary conditions.

Conjugate Gradient Method (CG) calculates the smallest eigenvalues of a symmetric, positive definite sparse matrix using the power method to solve the system of linear equations. The 14000 × 14000 input matrix contains 1853104 pseudo-randomly generated non-zero elements.

3-D FFT (FT) solves the given partial differential equation using forward and inverse FFT on a 256 × 256 × 128 imaginary matrix.

Block Tridiagonal CFD (BT) computes the solution to independent systems of non-diagonally dominant, 5 × 5 block tridiagonal equations.

Scalar Pentadiagonal CFD (SP) computes the solution to independent systems of non-diagonally dominant, scalar pentadiagonal equations.

3 Parallelization process

In this section we outline the general parallelization strategy, and briefly summarize the special cases requiring transformations other than the default ones described here. The steps basically include parallelization and optimization steps which integrates all data and control flow information available. For more details on the parallelization process, see [5].

3.1 Pre-parallelization transformations

The process of porting the NAS Benchmark to the AP1000 required several syntactic changes in order to meet VPP-Fortran specifications, and to accommodate array partitioning in the

next phase. This section summarizes four modifications, some of which lead to a tremendous blowup in the source code size, hindering the readability of the resulting VPP-Fortran code.

- The shape and partitioning for arrays must be known at compile time. Because the shape of partitioned arrays cannot be parameterized as an argument, a procedure was defined for each array shape.
- The shape of a partitioned array must remain constant throughout the entire program. In FT, for example, instances where the array shape is modified was eliminated by defining additional arrays for each different array shape.
- Only one index partitioning can be defined for a partitioned array. In cases where portions of an array are used for different computations and each portion requires a different partitioning, the original array was separated into distinct arrays for each respective computation.
- Subarrays of global partitioned arrays cannot be passed as real arguments. Those instances were decoupled into two separate arguments—global partitioned array name and array index offset.

3.2 Partitioning phase

The array and loop partitioning and distribution were selected so that the number of remote data accesses is minimized. The analysis was performed manually taking into consideration the optimizations that are done during the next phase, described in the next section. In the cases where conflicts between partitionings occur, preference was given to partitionings in the heavily executed areas within the timed region. In order to determine which partitioning yields the best running time, preliminary versions of the benchmarks were written varying the partitioning.

3.3 Optimization phase

The optimization phase is divided into four steps: removing unnecessary global address calculation, aggregating data communication and removing unnecessary barriers synchronizations. These optimizations were incorporated into the parallelization process based on the tradeoff between the time invested in implementation and execution time improvement reported in Section 5.

1. This optimization step effectively remove unnecessary global address calculation by referencing a partitioned array using its local name (as opposed to its global name) when the reference is known to be local.
2. Aggregating data communication is one of the most effective optimizations. Because an overhead is associated with each remote data transfer, reducing the number of transfers, reduces the execution time. In this step, the `SPREAD MOVE` directive was used to aggregate individual remote data accesses into larger blocks.
3. Overlap areas were incorporated into the data partitioning and the `OVERLAPFIX` directive was inserted to aggregate data communication. This optimization is a more specialized data aggregation for a more specific, but common data access pattern.
4. Unnecessary barrier synchronizations are removed. By default, synchronizations occur before and after a parallel loop, and after a remote data transfer completion.

3.4 Conflict resolution

This section describes three of the problems encountered when parallelizing the benchmark programs using the described parallelization scheme. The first problem, due to conflicts in the partitioning in the FT, BT, and SP benchmarks was resolved using data redistribution. Since the amount of computation performed between each data redistribution is large, redistribution proved to be effective for those cases. Secondly, the partitioning for SP and BT benchmarks could have been done across either one or two dimensions of the arrays. The tradeoffs in partitioning over two dimensions are performance improvement for larger machine sizes versus more memory and more time spent in redistribution, if performed. For SP and BT, even with data redistribution, the best performance of the 2-D partitioned version still exceeded that of the 1-D partitioned version. Lastly, for the CG and MG benchmarks, changes to the data representation were necessary in order to improve the runtime performance. The changes basically minimized the amount of remote data accesses.

Table 1: NAS Benchmark problem size A execution time for the AP1000

Benchmark	Mach. size	Runtime (sec)	Total Mem. (64-bit words)
EP	512	30.4	67.1
	256	60.7	33.6
	128	121.4	16.8
	64	242.7	8.4
FT	256	38.0	92.7
	128	54.3	92.5
CG	512	159.3	38.6
	256	146.3	26.0
	128	136.4	19.7
BT	512	571.5	91.8
	256	976.3	81.3
	128	1631.7	76.0
	64	2874.7	73.4
SP	512	688.3	36.7
	256	1128.3	26.2
	128	1624.2	21.0
	64	2790.6	18.3

4 Benchmark Performance

After applying the parallelization steps described in the previous section, the execution time for each NAS benchmark were measured for different machine configurations shown in Table 1. [1] [2] [3]

5 Hints to the VPP-Fortran User

In this section we summarize the lessons learned from this experience which will be beneficial to the VPP-Fortran programmer in making partitioning and implementation decisions. We first comment on methods of improving the program performance, and then describe the relative costs of basic VPP-Fortran operations.

5.1 General Comments

5.1.1 Communication aggregation

In the current language implementation, when using the SPREAD MOVE directive, the degree to which communication aggregation is performed is solely dependent on the user-specified ordering of the loop iterators and the partitioning of the contained arrays. As a result, the difference in runtime performance between the worst case and optimal ordering and partitioning combination may be larger than 200%, due to the larger number of remote data requests, and the overhead associated with each request. Because the execution time can be significantly affected by the amount of communication aggregation, we warn the user to give special attention to the ordering of the SPREAD MOVE loop iterators. To maximize the amount of aggregation, the general rule of thumb is the outermost loops should iterate over the partitioned dimensions of the global array in the SPREAD MOVE, and the innermost loops should iterate over the other dimensions.

5.1.2 Data redistribution

As discussed in Section 3, data redistribution was applied to resolve array distribution conflicts. Because no sophisticated parallelizing and performance tools were available at the time of this study, resolving such conflicts were manually done on a trial-and-error basis. For the SP, BT, and FT benchmarks, because the amount of calculation performed between redistributions was relatively large, the version with redistribution yielded a better runtime performance than the version without redistribution. We recommend to the user to consider data redistribution to resolve partitioning conflicts.

5.2 Basic operation costs

5.2.1 Barrier synchronization

We first consider the cost of a hardware and software barrier synchronizations implemented in the current runtime system. If the barrier is across all nodes, the synchronization is done in hardware using the S-net. Otherwise, it is done in software using the T-net. The table below shows the relative cost of the two types of barrier synchronization with respect to number of nodes involved in the barrier. Basically, hardware synchronizations require the same amount of time regardless of the machine size, whereas the software synchronizations depends on the number of nodes involved in the synchronization, $\log_2(node) \times 32 \times 10^{-5}$ seconds.

Based on these measurements, the user should keep in mind that when parallelizing

[1] The runtime system version rel.940530, and operating system version 1.4.1 were used to collect this data.

[2] These timings have not been officially submitted to NASA.

[3] The MG timings are currently not available.

Table 2: Barrier synchronization timings

Barrier type	Num. of nodes	Time (10^{-5} sec)
Hardware (across all nodes)	512	1.0626
	256	1.0615
	64	1.0640
Software (across a subregion of nodes)	256	266.9067
	64	224.2551
	16	150.1888
	4	64.5072
	2	32.6328

a program which requires many barrier synchronizations before and after parallel loops, the synchronization across a subregion of processors is slightly more expensive than one performed across all nodes. In other words, the default synchronizations associated with a parallel loop which is partitioned over a single dimension will be faster than one partitioned over multiple dimensions.

5.2.2 Global and local array name references

As a parallelization step, the global array names which access a local region of a partitioned array were replaced by their corresponding local array names. The difference between the two types of references is the global reference must first calculate the node on which the array element resides and the local array offset prior to the actual array access. This optimization basically eliminates the cost of the node and offset calculation.

Table 3 summarizes the time required for node and address calculation based on the time for a local array access using both the global and local array name. The global address calculation requires 1.37×10^{-4} to 1.86×10^{-4} seconds. Like the barrier synchronization optimization, this one can potentially reduce the execution only by a small fraction of the running time.

5.2.3 Block-cyclic distribution

When deciding whether an array should be distributed in a block or cyclic manner, the programmer should keep in mind the features available for the different distribution schemes. Of the two, the cyclic scheme has the most restrictions: overlap area cannot be used with the cyclic distribution, and the cyclic width can only be of unit length. In terms of performance, the address calculation for elements in a cyclically distributed array is slightly higher than the block distribution, about 101.4%-108.3% times slower according to the timings in Table 3.

Table 3: Global array address calculation timings

Array oper.	Global ref.	Local ref.	Global addr. calc.
Block (10^{-4} sec)			
Read	1.7213	0.0065	1.7148
Write	1.3808	0.0049	1.3759
Cyclic (10^{-4} sec)			
Read	1.8655	0.0081	1.8574
Write	1.4020	0.0065	1.3955

6 Language improvements

We present a list of VPP-Fortran language improvements, based on our experiences in parallelizing the NAS Benchmark, which would greatly enhance the flexibility and usability of the language. In this section, we describe the limitations in the VPP-Fortran `OVERLAPFIX` operation, cyclic distribution constraints, communication aggregation, and global vector operations.

6.1 Updating overlap regions

In the current language specification, the `OVERLAPFIX` directive updates the overlap regions over the entire corresponding array. In some cases, the overlap area only within a specified region was necessary, for example, along a particular array dimension, within a selected region, or along either the positive or negative side of the overlap region. To reduce the amount of data transfer to those which are necessary, the `OVERLAPFIX` directive should be augmented to allow a more detailed specification of the region to update.

Due to a requirement by the `OVERLAPFIX` statement and a FORTRAN77 restriction, it is not possible to specify the array to be `OVERLAPFIX`-ed as a procedure dummy argument. As a result, a copy of the procedure containing the `OVERLAPFIX` must be defined for each array argument, resulting in a severe increase in code size. The requirement that the global and local array to be fixed should be `EQUIVALENCE`d in the procedure invoking

OVERLAPFIX should be removed. The information from the EQUIVLANCE should be obtain by some other means or instruction.

6.2 Periodic overlap

Based on our experience in parallelizing the MG benchmark, we propose a periodic option with the overlap area partitioning. With this option, the the opposite boundary elements will also be included in the overlap region update. Using the current language specification, it is possible to explicitly implement a periodic boundary, but because the boundary elements must be treated as special cases, the code increases in size and becomes unreadable. The user should be able to specify the periodic option in the index partitioning declaration, and reference the opposite boundary overlap data by using *lowerbound-1* and *upperbound+1* indices.

6.3 Cyclic distribution

Cyclic distribution has several restrictions which makes it not as flexible as block distribution. The language should be augmented so that a cyclic width can be specified by the user, and overlap region can be defined with this distribution type.

6.4 Communication aggregation

As explained in Section 5, the degree to which communication aggregation is performed in the SPREAD MOVE construct is dependent on the ordering of the loop iterators, and the partitioned array dimensions. Ideally the user should not have to know details of the compiler implementation in order to attain the optimal performance. The compiler should include an optimizing module which, based on some heuristic, would attempt to reorder the loop iterators which best matches the array partitioning.

6.5 Global vector operation limitation

In the current implementation, global vector operations use the B-net when transmitting the result to the nodes. Because the ring buffer length for the B-net is limited to 512KB, vectors of up to 512KB in length can be involved in a vector global operation. If the vector exceeds this length, the user must manually divide the array and DO-loop into smaller chunks. Such implementation-dependent restrictions should be eliminated by modifying the current implementation so that the T-net is used instead, even at the expense of increasing the basic cost of a global vector operation.

7 Conclusion

This study attempted to evaluate the VPP-Fortran language by parallelizing and tuning the NAS Parallel Benchmark using VPP-Fortran. The execution time for the NAS Benchmark programs on the AP1000 was presented in this paper. Based on our experience in parallelizing the benchmark, we presented advice to the VPP-Fortran user in regard to performance tuning, and recommendations to the language designer to improve the VPP-Fortran's usability and flexibility.

8 Acknowledgments

The authors would like to thank Mr. T. Doi, Mr. H. Iwashita, and Mr. K. Hayashi for their advice on the VPP-Fortran language and implementation, and Dr. M. Ishii, Mr. H. Shiraishi, and Mr. M. Ikesaka for their continued support on this project.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, eds., "The NAS Parallel Benchmark," Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, Moffett Field. CA 94035, August 1991.

[2] Fujitsu Limited. "AP1000 VPP Fortran Users Guide," March 1994.

[3] H. Ishihata, T. Horie, and T. Shimizu. "Architecture for the AP1000 Highly Parallel Computer," *Fujitsu Science Technical Journal*, v. 29, no. 1, March 1993, pp 6-14.

[4] 岩下 英俊, 進藤 達也, and 岡田 信. "VPP Fortran : 分散メモリ型並列計算機言語," *JSPP 1994*, May 1994, pp. 153-160.

[5] S. Kaneshiro, and T. Shindo. "The NAS Parallel Benchmark on the AP1000," Technical Report LTM-94-0918-07, Fujitsu Parallel Computing Research Center, 1994.