Experiments with HPF Compilation for a Network of Workstations [⋆]

Fabien COELHO (coelho@cri.ensmp.fr)

Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, F-77305 Fontainebleau Cedex, FRANCE.
phone : +33 1 64 69 48 52, fax : +33 1 64 69 47 09

Abstract. HIGH PERFORMANCE FORTRAN (HPF) is a data-parallel Fortran for Distributed Memory Multiprocessors. HPF provides an interesting programming model but compilers are yet to come. An early implementation of a prototype HPF optimizing compiler is described. Experiments of PVM 3-based generated code on a network of workstations are analyzed and discussed. It is shown that if such systems can provide very good speedups at low cost, they cannot allow scalable performance without specific communication hardware. Other early implementations of HPF compilers from academic and commercial groups are presented and compared to our work.

Introduction

The most promising parallel machines seem to be the Distributed-Memory Multiprocessors (DMM), such as the Intel's Paragon or the Thinking Machine's CM5. They are scalable, flexible, and offer good price/performance ratio. However their efficient programming using the message-passing paradigm is a complex and error-prone task, which makes coding hard and expensive. To ease the programmer's burden, new languages have been designed to provide a uniform name space to the user. The compiler must handle the communications, and the machine can be used as a Shared-Memory Multicomputer.

High Performance Fortran (HPF) is a new standard to support data-parallel programming for DMMs. The HPF forum, composed of vendors, academics and users, specified HPF [6] in 1992. It is intended as a *de facto* standard, supported by many companies. The language is based on Fortran 90 with additions to specify the mapping of data onto the processors of a DMM (namely directives `align` and `distribute`), parallel computations with independent and data-parallel loops (`independent` directive and `forall` instruction), reductions, plus some other extensions like new intrinsics to query about the system at run-time. The specification effort has restarted in January 1994 to address problems left aside in 1992 such as parallel I/O and irregular computations.

Section 1 gives a brief overview of the implementation and the optimizations performed by our compiler. Section 2 describes results of runs of Jacobi iterations on a network of workstations, and Section 3 analyzes what can be expected from this kind of applications on such systems. Section 4 describes the current status of other HPF compilers and compares the shortcomings of the different implementations with respect to our experiments.

[⋆] Published in proceedings of HPCN Europe'94 (Munich, Germany)

1 Compiler implementation

A prototype HPF compiler [5] has been developed within PIPS [10] (Scientific Programs Interprocedural Parallelizer) at CRI. It is a 15,000 lines project that generates SPMD (Single Program Multiple Data) distributed code. Fortran 77 and various HPF constructs such as independent loops, static mapping directives and reductions are taken as input. The full semantics of the HPF mapping is supported, e.g. replicated dimensions and arbitrary cyclic distributions are handled. From this input, a 2PMD (2 Programs, Multiple Data, see Figure 3) Fortran 77 message passing code is generated. The first program mainly deals with I/O to be performed by the host processor, and the other is an SPMD code for the nodes. The code uses a library of PVM 3-based [7] run-time support functions. Thus the compiler output is as portable as PVM. Two files are also generated to initialize run-time data structures describing the distributed arrays.

The compilation is divided into 3 phases:

First, the input code is parsed and normalized (e.g. temporaries are inserted to avoid indirections...), the directives are analyzed and new declarations are computed for the distributed arrays. These new declarations reduce when possible the amount of allocated memory on each node.

The second phase generates the run-time resolution code [4] for the host and nodes as a double rewriting scheme on the abstract syntax tree of the program. The *owner computes rule* is used to define the processor that performs the computations. Accesses to remote data are guarded by tests and communications if necessary. This rewriting scheme has been formalized, which enabled us to prove parts of the correctness of the compilation process. For instance, the balance of communications can easily be proved by checking that each rewriting rule generates as many sends as receives with complementary guards and destination.

The third phase performs optimizations when necessary conditions are met. An overlap analysis for multi-block-distributed arrays [8] is implemented. Alignment shifts at the template level are used to evaluate the overlap width in each directions. Guards are generated for non contributing nodes. Messages are vectorized by the process, and moved outside of the loop body. Reductions are compiled through calls to dedicated run-time functions.

2 Experiments

Experiments have been performed on an unloaded Ethernet network, with up to 8 Sparc Stations 1. The application (see Figure 1 with HPF directives) computes 100 Jacobi iterations on a 2D plate. The computation models heat transfers in an homogeneous square plate when the temperature is controlled on the edges. This application is fully parallel and few communications are needed. The kernel comes from [12], with a few I/O and initializations added. The communication pattern induced by the computation kernel, as well as the different initializations of some areas within the plate, are representative of stencil computations such as wave propagation in geophysics. The arrays are aligned and block-block distributed to minimize the amount of data exchange. Figure 2 shows extracts from the generated code for the machine nodes: It is an SPMD code parametrized by the processor's identity. Declarations are reduced to what is necessary on one processor, and extended to store remote data in overlap areas. Run-time support functions are prefixed by `hpfc`. The `north` vector initialization introduces a guard so that only the appropriate processors execute the loop. The local loop bounds are computed by each processor before the loop. The last part of the code shows the first send within the kernel of the computation: Each selected processor first computes to which neighbor it has to send the data, then packs and sends them. The last line is the comment that introduces the corresponding receive. All codes were compiled with all compilers' optimizations set on.

The measures are given for 1, 4 and 8 processors, and for plate width of 20 to 1,000 points (up to 2,000 points for 8 processors). They are based on the best of at least 5 runs. Figure 4 shows the Mflop/s achieved by the sequential runs when the plate width is scaled up. From these measures, 1.45 Mflop/s is taken as the Sparc 1 performance on this application. It matches most experimental points, and cache effects explain the other results. This reference is scaled to the number of processors to provide a realistic peak performance for the network of workstations used. Other measures are displayed as the percentage of this peak performance reached by the best run. This metric (e) is called *relative realistic efficiency*. Real speedups were rejected because of the artefacts linked to cache effects: A good speedup may only mean a bad sequential performance for a given plate width. Moreover sequential experiments were stopped because of memory limitations on the tested configuration. An absolute measure (e.g. Mflop/s) would not have clearly shown the relative efficiency achieved. Note that $p.e$ (p processors, efficiency e) is a speedup, but not a measured one.

Figures 5 and 6 show the experimental data using this metric and theoretical curves derived from realistic assumptions in the next section. Large computations lead to up to 80–90% efficiency. The complementary part evaluates the loss due to latencies and communications. These results are used to test the formulas derived in the next section. These formulas are then used to precisely analyze the performance which can be expected from such applications run on networks of workstations when the parameters change.

3 Analyses

The theoretical curves (doted lines on Figures 5 and 6) are based on assumptions about the network bandwidth ($\gamma = 700$ KB/s through PVM), the realistic peak performance of one Sparc 1 ($\pi = 1.45$ Mflop/s) and an offset ($\alpha p + \delta$) to model the initialization times. It is assumed (a) that floats are 4 bytes long, (b) that the application is block-block distributed, (c) that communications and computations do not overlap, and (d) that the communications are sequentialized on the medium. If p is the number of processors, n the plate width and t the number of iterations, then the total time complexity $\Theta_t(n,p)$ to run the program is:

$$\Theta_t(n,p) \simeq \frac{4n^2t}{\pi p} + \frac{16n(\sqrt{p}-1)t}{\gamma} + \alpha p + \delta \qquad (1)$$

The first term is the computation time and the second the communication time. If $\mu = \frac{\pi}{\gamma}$, the continuous regime relative realistic efficiency is:

```
      program jacobi
      parameter (n=500)
      real tc(n,n), ts(n,n), north(n)
chpf$ template t(n,n)
chpf$ align tc(i,j), ts(i,j) with t(i,j)
chpf$ align north(i) with t(1,i)
chpf$ processors p(2,4)
chpf$ distribute t(block,block) onto p
chpf$ independent(i)
      do i=1,n
         north(i) = 100.0
      enddo
      ...
      do k=1,time
c     kernel of the computation
chpf$ independent(j,i)
      do j=2,n-1
        do i=2,n-1
          ts(i,j) = 0.25 *
     $      (tc(i-1,j) + tc(i+1,j)
     $     + tc(i,j-1) + tc(i,j+1))
         enddo
      enddo
      ...
```

Fig. 1. Jacobi iterations kernel

```
      program node
      include 'fpvm3.h'
      include 'parameters.h'
      ...
      real*4 north(1:125),
     $    ts(1:250,1:125), tc(0:251,0:126)
      call hpfc_init_node
      ...
      if (mypos(1,1).eq.1) then
        call hpfc_loop_bounds(i_5,i_6,1,500..
        do i_4 = i_5, i_6
          north(i_4) = 100.0
        enddo
      endif
      ...
      do k = 1, time
c p(1:2,2:4) send tc(1:250,1) to (-1)
      if (mypos(2,1).ge.2) then
        call hpfc_cmpneighbour(-1)
        call hpfc_pack_real4_2(tc, ...)
        call hpfc_sndto_n
      endif
      ...
c p(1:2,1:3) receive tc(1:250,126) from (+1)
```

Fig. 2. Node code extract

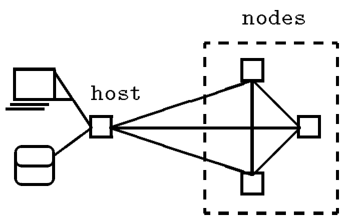

Fig. 3. Machine model

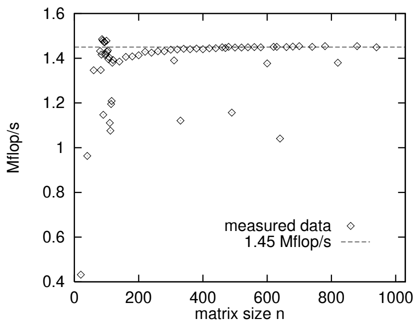

Fig. 4. Sequential Mflop/s

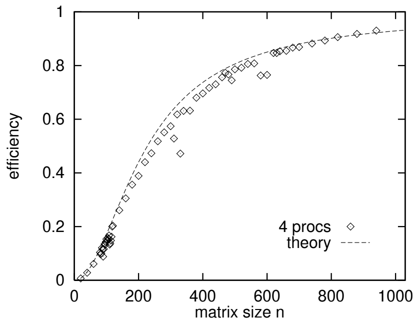

Fig. 5. Efficiency with 4 processors

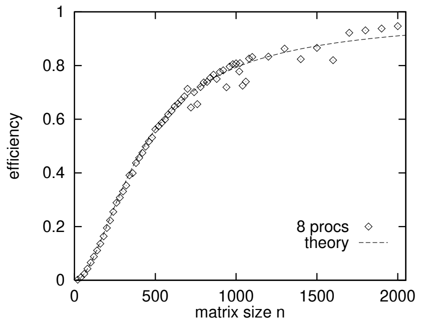

Fig. 6. Efficiency with 8 processors

$$e_\infty(n,p) = \lim_{t\to\infty} \left(\frac{4n^2t}{\pi p}\right) \Theta_t(n,p)^{-1} = \left(1 + \mu\frac{4p(\sqrt{p}-1)}{n}\right)^{-1} \qquad (2)$$

These functions match very well the experimental points as can be seen on the Figures. They can be used to see what can be expected from such applications on networks of workstations. The interesting point is not to scale up the problem size for better efficiency, but to analyze what happens when the number of processors is scaled with a constant problem size: The efficiency quickly decreases. The minimal size to get efficiency e with p processors can be computed (3). An optimal number of processors is also easily derived (4), for which 2/3 of the network bandwidth is used.

$$n \geq \left(\frac{4e}{1-e}\right) \mu p(\sqrt{p}-1) \quad (3) \qquad\qquad p \simeq \left(\frac{n}{2\mu}\right)^{\frac{2}{3}} \quad (4)$$

The μ parameter is expected to belong to a large range of values as processor and network performances change. For a network of RISC 6000 on Ethernet, $\mu \simeq 100$[2], while for SS1 $\mu \simeq 2$. For a realistic application with $n = 1000$, $\mu \simeq 100$ gives the optimal number of processors $p = 3$, a 1.5 speedup and an overloaded network! This clearly shows that a set of fast workstations linked by Ethernet is definitely not an underused massively parallel machine, even for an embarrassingly parallel program like Jacobi iterations. Scaling the problem size to show better results is a fake, because the convergence speed and other requirements of the algorithms [11] would dramatically slow down the resolution.

The first problem is the low bandwidth of the network, especially when compared to processor speed. The local networks should be upgraded as much as the workstations to get good performance. The second problem is the complexity in p of the functions since the communications are sequentialized on the medium. The network is the bottleneck of the system.

4 Related work

A lot of work about compilation and prototype compilers for DMMs was published during the last few years. The Vienna Fortran Compiling System [13] and Fortran D [12] are early developments of data-parallel languages on which HPF is based. The Fortran D prototype allows only one dimension to be distributed, and no I/O. Adaptor [2] is a pragmatic and efficient public domain software which implements some features of HPF and which was initially based on CM-Fortran. Many vendors have announced commercial compilers: Applied Parallel Research, Digital Equipment, Pacific-Sierra Research, the Portland group...These products are not yet available, or do not yet implement all HPF features. Moreover the performances of these systems will depend on the implemented optimizations, and many points are still an open research area. Performance figures for some of these prototypes are available [9, 3], but comparisons are difficult because the applications and the machines are different.

[2] This figure is based on the peak performance of a RS 6000 running at 62.5 MHz

Conclusion

Our optimizing compiler supports the full HPF mapping semantics and produces portable code. Experiments with Jacobi iterations on a network of workstations were performed with the compiler output. Very good efficiencies were measured for large problem sizes. These results have been analyzed and discussed, and it has been shown that networks of fast workstations cannot provide good performance without specific communication hardware.

Some HPF features, such as procedure interfaces and the `forall` instruction, are missing. Moreover I/O are not yet efficiently compiled. Future work will include a better handling of I/O, experiments using more realistic programs and real parallel machines (Paragon, CM5, Alpha farm...), as well as implementation of new optimization techniques [1].

References

1. C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, Dec. 1993. Also available as TR EMP A/250/CRI.
2. T. Brandes. Adaptor: A compilation system for data parallel fortran programs. Technical report, High Performance Computing Center, German National Research Institute for Computer Science, Aug. 1993.
3. T. Brandes. Results of Adaptor with the Purdue Set. Internal Report AHR-93 3, High Performance Computing Center, German National Research Institute for Computer Science, Aug. 1993.
4. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.
5. F. Coelho. Étude de la Compilation du *high performance fortran*. Master's thesis, Université Paris VI, Sept. 1993. Rapport de DEA Systèmes Informatiques. TR EMP E/178/CRI.
6. H. P. F. Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, May 1993. *Version 1.0*.
7. A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.
8. H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Vienna, 1989.
9. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD Distributed-Memory machines. In *ACM International Conference on Supercomputing*, 1992.
10. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, June 1991.
11. J. Pal Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, pages 42–50, July 1993.
12. C.-W. Tseng. *An Optimising Fortran D Compiler for MIMD Distributed Memory Machines*. PhD thesis, Rice University, Houston, Texas, Jan. 1993.
13. H. Zima and B. M. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, Feb. 1993.