Performance assessment of parallel techniques

T. Grytsenko & A. Peratta
Wessex Institute of Technology, UK

Abstract

The goal of this work is to evaluate and compare the computational performance of the most common parallel libraries such as Message Passing Interface (MPI), High Performance Fortran (HPF), OpenMP and DVM for further implementations. Evaluation is based on NAS Parallel benchmark suite (NPB) which includes simulated applications BT, SP, LU and kernel benchmarks FT, CG and MG. A brief introduction of the four parallel techniques under study: MPI, HPF, OpenMP and DVM, as well as their models is provided together with benchmarks used and the test results. Finally, corresponding recommendations are given for the different approaches depending on the number of processors.
Keywords: MPI, HPL, DVM, OpenMP, parallel programming, performance, parallel calculations.

1 Introduction

This section provides a brief introduction of the four parallel techniques under study: MPI, HPF, OpenMP and DVM, as well as their models.

1.1 The Message-Passing Information programming model (MPI)

Programming models are generally categorised according to the way in which how memory is used. In the shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes, each with its own local memory. In the message passing model processes communicate with other processes by sending and receiving messages (see Figure 1). When data is passed in a message, the sending and receiving processes must work to transfer the data from the local memory of one to the local memory of the other.

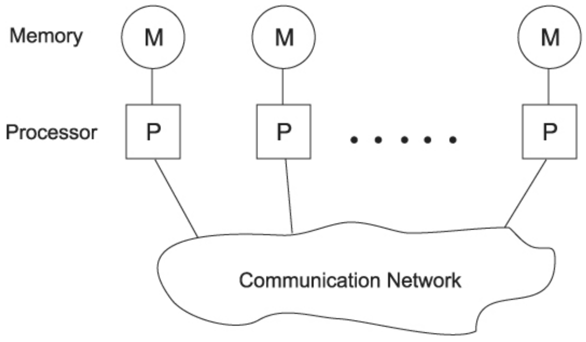

Figure 1: The message-passing programming paradigm.

Message passing is used widely on parallel computers with distributed memory, and on clusters of servers. The advantages of using message passing are [9, 10]: i- Portability; ii- Universality, i.e. the model makes minimal assumptions about underlying parallel hardware; and iii- Simplicity, in the sense that the model supports explicit control of memory references for easier debugging.

The primary goals of MPI are efficient communication and portability. Although several message-passing libraries exist on different systems, MPI is popular due to: i- support for full asynchronous communication, i.e. process communication can overlap process computation; ii- group membership, that is, processes may be grouped based on context; iii- Synchronization variables which protect process messaging. When sending and receiving messages, synchronisation is enforced by source and destination information, message labelling, and context information; iv- Portability, in the sense that all MPI implementations are based on a published standard which specifies the semantics for usage. An MPI program consists of a set of processes and a logical communication medium connecting those processes. An MPI process cannot directly access memory in another MPI process. Inter-process communication requires calling MPI routines in both processes. MPI defines a library of routines through which MPI processes communicate. The MPI library routines provide a set of functions that support: point-to-point communications, collective operations, process groups and communication contexts, Process topologies, and Data type manipulation. MPI includes more than 300 different procedures to provide necessary functionality and provides interfaces for C/C++ and Fortran languages.

1.2 HPF

In the data parallel model of HPF, calculations are performed concurrently over data distributed across processors. Each processor operates on the segment of data it owns. In many cases HPF compiler can detect concurrent calculations with distributed data. HPF advises a two-level strategy for data distribution. First, arrays should be co-aligned with the ALIGN directive. Then each group of co-aligned arrays should be distributed onto abstract processors with the DISTRIBUTE directive. There are several ways to express parallelism in HPF:F90 style of array expressions, FORALL and WHERE constructs, the INDEPENDENT directive and HPF librarybintrinsics [5]. In array expressions, operations are performed concurrently on segments of data owned by a processor. The compiler takes care of communicating data between processors if necessary. The INDEPENDENT directive asserts that there are no dependences between different iterations of a loop and the iterations can be performed concurrently. In particular it asserts that Bernstein's conditions are satisfied: sets of read and written memory locations on different loop iterations don't overlap and no memory location is written twice on different loop iterations [6]. All loop variables which do not satisfy the condition should be declared as NEW and are replicated by the compiler in order for the loop to be executed in parallel. The concurrency provided by HPF does not come for free. The compiler introduces overhead related to processing of distributed arrays. There are several types of overhead: (1) creating communication calls, (2) implementing independent loops, and (3) creating temporaries, and accessing distributed arrays. The communication overhead is associated with requests of elements residing on different processors when they are necessary for evaluation of an expression with distributed arrays or executing an iteration of an independent loop. Some communications can be determined at compile time while others can be determined only at run time causing extra copying and scheduling of communications. As an extreme case, the calculations can be scalarised resulting in a significant slowdown. HPF standard was developed in 1993 as an extension of Fortran 90. Later on such extensions were been proposed for C/C++.

1.3 OpenMP

OpenMP [7] is designed to support portable implementation of parallel programs for shared memory multiprocessor architectures. OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran, C and C++ to express shared memory parallelism. It provides an incremental path for parallel conversion of any existing software, as well as targeting at scalability and performance for a complete rewrite or entirely new software. A *fork-join* execution model is employed in OpenMP. A program written with OpenMP begins execution as a single process, called the *master thread*. The master thread executes sequentially until the first parallel construct is encountered (such as a "PARALLEL" and "END PARALLEL" pair). The master thread, then, creates a team of threads, including itself as part of the team. The statements enclosed in the parallel construct are executed in parallel by each thread in the team until a worksharing construct is encountered. The "PARALLEL DO" or "DO" directive is such a worksharing construct which distributes the workload of a DO loop among the members of the current team. An implied synchronisation occurs at the end of the DO loop unless an "END DO NOWAIT" is specified. Data sharing of variables is specified at the start of parallel or worksharing constructs using the SHARED and PRIVATE clauses. In addition, reduction operations (such as summation) can be specified by the "REDUCTION" clause. Upon completion of the parallel construct, the threads in the team synchronize and only the master thread continues execution. OpenMP introduces a powerful concept of orphan directives that greatly simplify the task of implementing coarse grain parallel algorithms. Orphan directives are directives encountered outside the lexical extent of the parallel region. The concept allows the user to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained region.

1.4 DVM

DVM is an extension of ANSI-C and Fortran languages with annotations named DVM-directives. DVM-directives may be conditionally divided on three subsets: data distribution directives, computation distribution directives, and remote data specifications. DVM model of parallelism is based on specific form of data parallelism called SPMD (Single Program, Multiple Data). In this model all the processors concerned execute the same program, but each processor performs its own subset of statements in accordance with the data distribution. In DVM model at first a user defines multidimensional arrangement of virtual processors, which sections data and computations will be mapped on. The section can be varied from the whole processor arrangement up to a single processor. Then the arrays to be distributed over processors (*distributed data*) are determined. These arrays are specified by data mapping directives. The other variables (distributed by default) are mapped by one copy per each processor (*replicated data*). A value of replicated variable must be the same on all the processors concerned. Single exception of this rule is variables in parallel loop. Data mapping defines a set of local or *own variables* for each processor. A set of own variables determine the *rule of own computations*: the processor assigns the values to its own variables only. DVM model defines two levels of parallelism: data and task parallelism. Data parallelism is implemented by distribution of tightly enclosed loop iterations over the processors of the processor arrangement (or the arrangement sections). The loop iteration is executed on one processor entirely. The statements located outside of the parallel loop are executed according to the rule of own computations. Task parallelism is implemented by the distribution of data and computations over disjoined sections of processor arrangement. When calculating the value of own variable, the processor may need in values of both own and remote variables. All remote variables must be specified in remote data access directives.

2 NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB's) [4] were derived from Computational Fluid Dynamics (CFD) codes. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. NPB consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. These five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. The benchmarks are specified only algorithmically and referred to as NPB. Details of the NPB suite can be found in [4]. In this paper we study only six benchmarks (excluding IS and EP):

- **BT** is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible equation $Ku = r$ (1) where *u* and *r* are 5x1 vectors defined at the points of a 3D rectangular grid and K is a 7 diagonal block matrix of 5x5 blocks. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the *x*, *y* and *z* dimensions: $K \cong BT_x \cdot BT_y \cdot BT_z$, where BT_x, BT_y and BT_z are block tridiagonal matrices of 5x5 blocks if grid points are enumerated in an appropriate direction. The resulting system is then solved by solving the block tridiagonal systems in *x*-, *y*- and *z*-directions successively.
- **SP** is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization and Pulliam-Chaussee diagonalisation of the operator of equation (1) and adds fourth-order artificial dissipation:

$$K \cong T_x \cdot P_x \cdot T_x^{-1} \cdot T_y \cdot P_y \cdot T_y^{-1} \cdot T_z \cdot P_z \cdot T_z^{-1}$$

where *Tx*, *Ty* and *Tz* are block diagonal matrices of 5x5 blocks, *Px*, *Py* and *Pz* are scalar pentadiagonal matrices. The resulting system is solved by inverting the block diagonal matrices $T_x, T_x^{-1} \cdot T_y, T_y^{-1} \cdot T_z$ and then solving the scalar pentadiagonal systems.

- **LU** is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretisation of the Navier-Stokes equations in 3-D by splitting it into block **L**ower and **U**pper triangular systems:

$$K \cong \omega(2-\omega)(D+\omega Y)(I+\omega D^{-1}Z)$$

where ω is a relaxation parameter, *D* is the main block diagonal of *K*, *Y* consists of three sub block diagonals and *Z* consists of three super block diagonals. The problem is solved by computing elements of the triangular matrices and solving the lower and the upper triangular system.

- **FT** contains the computational kernel of a 3-D fast **F**ourier **T**ransform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension. The trans-formation can be formulated as a matrix vector multiplication:

$$v = (F_m \otimes F_n \otimes F_k)u$$

where u and v are 3D arrays of dimensions *(m,n,k)* represented as vectors of dimensions *m*x*n*x*k*. $A \otimes B$ is a block matrix with blocks $a_{ij}B$ and is called tensor product of A and B. The algorithm is based on representation of the FFT matrix as a product of three matrices performing several FFT in one direction. Henceforth FT performs FFTs in *x*-, *y*- and *z*- directions successively. The core FFT is implemented as a Swarztrauber's vectorisation of Stockham autosorting algorithm performing independent FFTs over sets of vectors.

• **MG** performs iterations of V-cycle multigrid algorithm for solving a discrete Poisson problem $\nabla u = v$ on a 3D grid with periodic boundary conditions [4]. Each iteration consists of evaluation of the residual $r = v - Au$, and of the application of the correction: $u = u + Mr$, where M is the V-cycle multigrid operator.

• **CG** uses a **C**onjugate **G**radient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries. A single iteration can be written as follows:

$$q = Ap,\ d = p^T q,\ \alpha = \frac{\rho}{d},$$
$$z = z + \alpha p,\ r = r - \alpha q,\ \rho_0 = \rho,$$
$$\rho = r^T r,\ \beta = \frac{\rho}{\rho_0},\ p = r + \beta p$$

The most time consuming operation is the sparse matrix vector multiplication $q = Ap$ which is carried out in parallel.

3 Test results

The NPB implementation is based on message passing standard (MPI). So, it is possible to compare original MPI implementation with the implementations of NPB by means of HPF, OpenMP and DVM techniques. MPI, OpenMP and HPF versions are tested in Origin 2000 hardware platform and DVM on RCC-cluster [3]. Results shown in diagrams below are based on information derived from sources [1–3,8]. Diagrams illustrate an execution time of MPI, OpenMP, HPF and DVM versions of six tests from NPB set as well as speedup of different versions for every test.

$$Speedup = {T_n}/{T_s}$$

where T_n is execution time on multiprocessor computer (*n=2,4,8,16,32*) and T_s is execution time on a single processor.

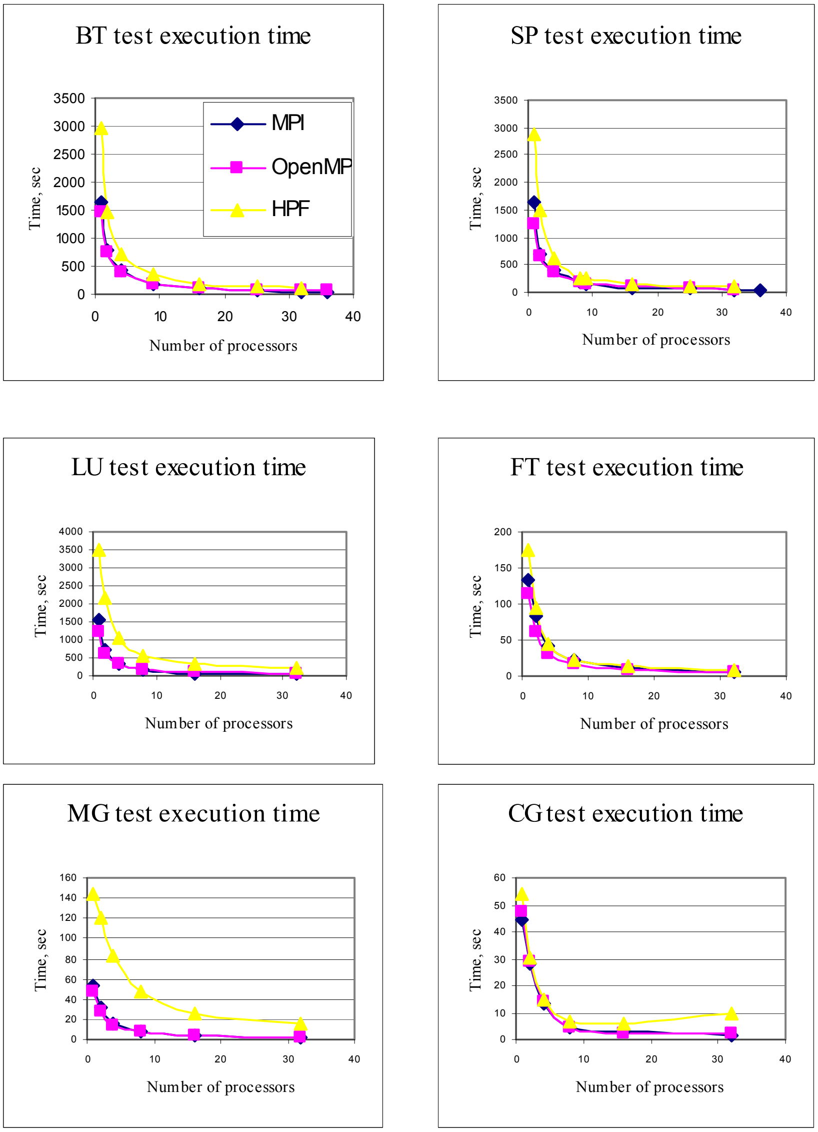

Figure 2: Test execution time of MPI-, OpenMP- and HPF-version on Origin 2000.

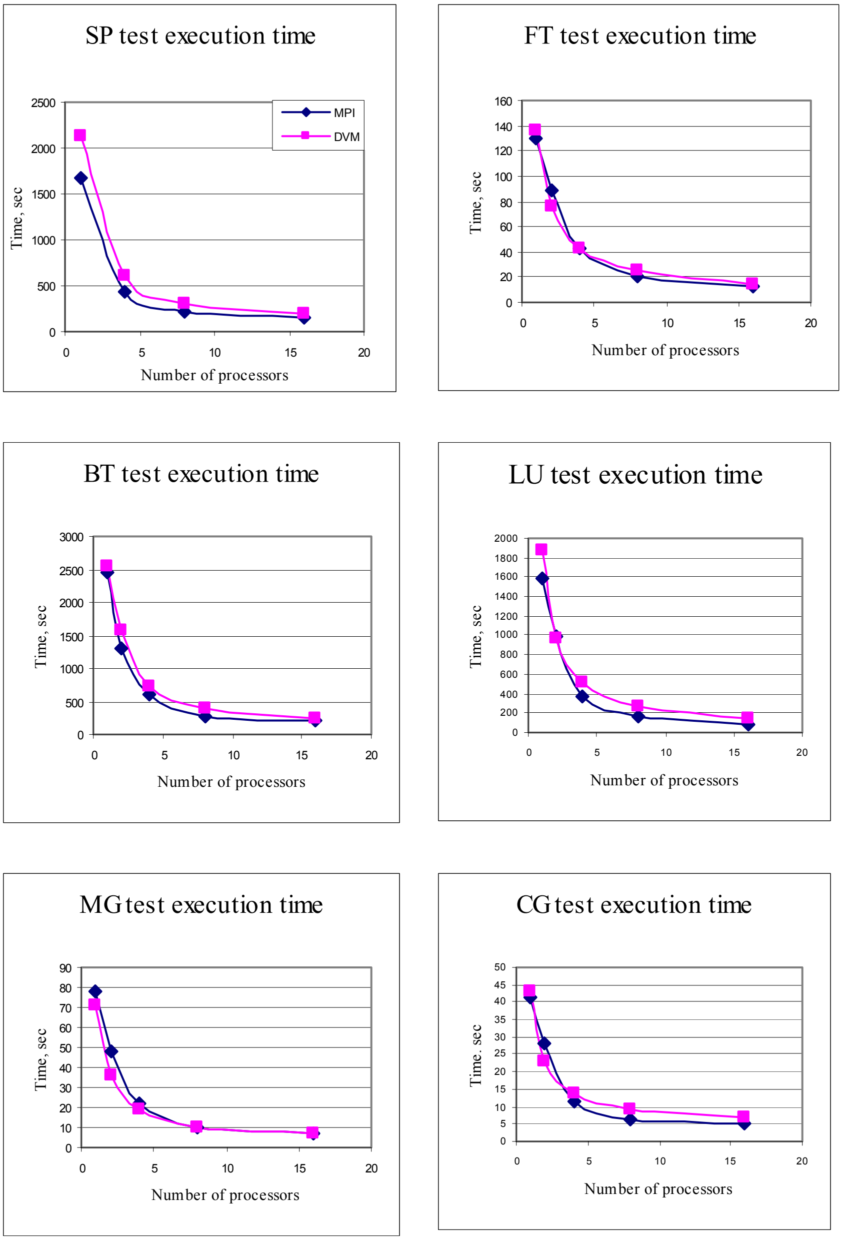

Figure 3: Test execution time of MPI- and DVM-version on RCC-cluster.

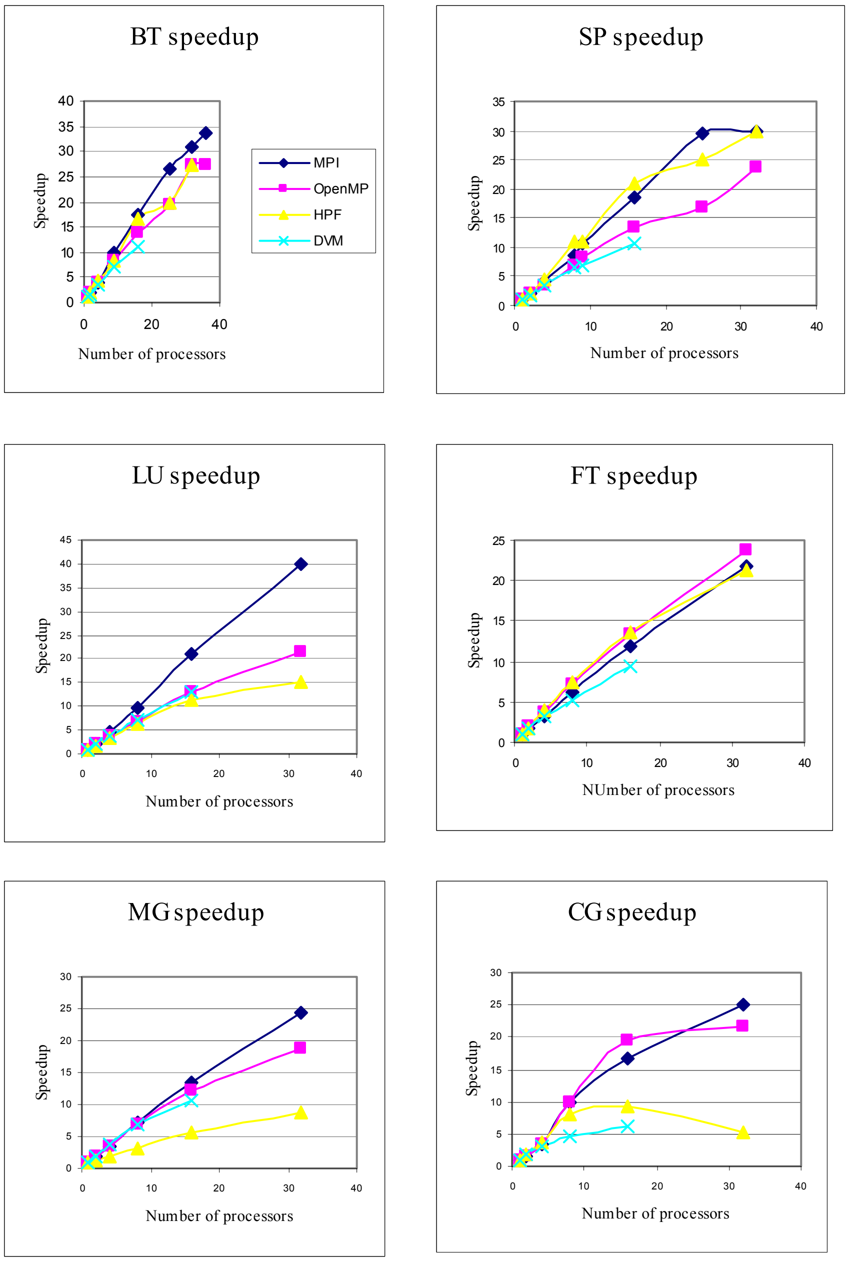

Figure 4: Speedup of MPI-, OpenMP-, HPF- and DVM-version for BT, SP, LU, FT, MG and CG.

4 Conclusions

In most cases execution time of MPI-version is lower, in comparison to other approaches. OpenMP-version is about 10% slower than MPI-version, whereas DVM is 20% and HPF is 30%. This difference increases with the number of processors. This is because of the efficient model of memory distribution in MPI which employs cache memory in a more efficient way. Speedup of MPI-version is also higher than speedup of other approaches. As a conclusion, if a given task is distributed over more than 16 processors then MPI offers the best solution for this suite of benchmarks. However, MPI is a low-level programming language in terms of parallel programming. This is “assembler” for parallel programming especially at data distribution stage and building of communication scheme between processes. In some cases when the number is lower than 16 it is advisable to use another technique such as OpenMP or DVM, which although are not as fast as MPI, they are easier to implement. Comparison on NPB 2.3 test cannot be comprehensive because these tests are developed on a very high level by team of experts but it can give a general imagination about possible performance of involved parallel techniques. Finally, in general parallel programmes require a considerable amount of modifications in order to optimise its computational performance. Hence, in parallel applications, the developer should have special knowledge not only in his/her scientific area but also in the specific parallel technique that leads to the optimum performance.

References

[1] Michael Frumkin, Haoqiang Jin and Jerry Yan (1998). *Implementation of NAS Parallel Benchmarks in High Performance Fortran*. NAS Technical Report NAS-98-009, NASA Ames Research Center.

[2] H. Jin, M. Frumkin and J. Yan (1998). *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance.* NASA Ames Research Center.

[3] V. Krukov. *Development of Parallel Programmes for Clusters and Networks* (2002). Keldysh Institute of Applied Mathematics.

[4] D. Bailey, T. Harris, W. Sahpir, R. van der Wijngaart, A. Woo, M. Yarrow (December 1995). *The NAS Parallel Benchmarks 2.0*. Report NAS-95-020.

[5] C.H. Koelbel (November 1997). *An Introduction to HPF 2.0. High Performance Fortran - Practice and Experience.* Supercomputing 97.

[6] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., M.E. Zosel (1994). *The High Performance Fortran Handbook.* MIT Press.

[7] *OpenMP Fortran Application Program Interface*, http://www.openmp.org

[8] *DVM. Execution performance of NAS tests*, http://www.keldysh.ru/dvm/

[9] *Writing Message-Passing Parallel Programs with MPI.* http://www.epcc.ed.ac.uk/ computing/ training/ document_archive/ mpi-course/ mpi-course.book_1.html

[10] *HP MPI User's Guide. Fourth Edition* http://www.docu.sd.id.ethz.ch/comp/stardust/SW/mpi/title.html