Computing and Informatics, Vol. 33, 2014, 1377–1399

OPTIMIZATION OF A PARALLEL CFD CODE AND ITS PERFORMANCE EVALUATION ON TIANHE-1A

Yonggang Che, Lilun Zhang, Chuanfu Xu, Yongxian Wang
Wei Liu, Zhenghua Wang

Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha 410073, Hunan, P. R. China
e-mail: ygche@nudt.edu.cn, zll0434@sina.com, xuchuanfu@nudt.edu.cn, yxwang@nudt.edu.cn, liuweinudt@126.com, zhhwang@nudt.edu.cn

Abstract. This paper describes performance tuning experiences with a parallel CFD code to enhance its performance and flexibility on large scale parallel computers. The code solves the incompressible Navier-Stokes equations based on the novel Slightly Compressible Model on three-dimensional structure grids. High level loop transformations and argument based code specialization are utilized to optimize its uniprocessor performance. Static arrays are converted into dynamically allocated arrays to improve the flexibility. The grid generator is coupled with the flow solver so that they can exchange grid data in the memory. A detailed performance evaluation is performed. The results show that our uniprocessor optimizations improve the performance of the flow solver for 1.38× to 3.93× on Tianhe-1A supercomputer. In memory grid data exchange optimization speeds up the application startup time by nearly two magnitudes. The optimized code exhibits an excellent parallel scalability running realistic test cases. On 4 096 CPU cores, it achieves a strong scaling parallel efficiency of 77.39 % and a maximum performance of 4.01 Tflops.

Keywords: Computational fluid dynamics, slightly compressible model, large-scale parallel computing, uniprocessor optimizations, in memory grid exchange, scalability, efficiency

Mathematics Subject Classification 2010: 65Y05

1 INTRODUCTION

LM3D [1, 2] is a CFD (Computational Fluid Dynamics) code for three dimensional low Mach number flows initially developed by Xiaogang Deng et al. of China Aerodynamics Research and Development Center (CARDC). It solves the incompressible Navier-Stokes equations with a finite volume method. For low Mach number perfect gas flow, Xiaogang Deng et al. suggested a novel Slightly Compressible Model (SCM) [1], in which the governing equations are simplified by neglecting the change of temperature in the state equation and the pressure change results from the density change. Thus the governing equations of LM3D are as follows:

$$Continuity: \quad \frac{D\rho}{\rho Dt} = -\partial_i \upsilon_i \tag{1}$$

$$Momentum: \quad \rho\frac{D\upsilon_i}{Dt} = -\partial p_i + \partial_j \tau_{ji} \tag{2}$$

$$State\ equation: \quad p = \text{constant} \cdot \rho^\sigma \tag{3}$$

where σ is a constant with a suggested value in the range of $[1.0, 1.56]$ in the numerical calculation. The governing equations can be written in non-dimensional variables as

$$\rho\frac{D\upsilon_i}{Dt} = -\kappa\partial p_i + \frac{1}{Re}\partial_j \tau_{ji} \tag{4}$$

$$p = \rho^\sigma \tag{5}$$

where Re is the Reynolds number, M_∞ is the free stream Mach number, and κ is a nondimensional number. $Re = \frac{\rho_\infty V_\infty L}{\mu}$. $M_\infty = \frac{\sqrt{u_\infty^2+v_\infty^2+w_\infty^2}}{a_\infty}$. $\kappa = \frac{1}{\gamma M_\infty^2}$.

The governing Equations (4)–(5) can be combined into one of conservative forms in generalized coordinates (τ, ξ, η, ζ)

$$\frac{\partial Q}{\partial \tau} + \frac{\partial E}{\partial \xi} + \frac{\partial F}{\partial \eta} + \frac{\partial G}{\partial \zeta} = \frac{1}{Re}\left(\frac{\partial E_v}{\partial \xi} + \frac{\partial F_v}{\partial \eta} + \frac{\partial G_v}{\partial \zeta}\right) \tag{6}$$

where

$$Q = J\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\end{bmatrix} \quad H = J\begin{bmatrix}\rho U\\ \rho U u + \Psi k_x p\\ \rho U v + \Psi k_y p\\ \rho U w + \Psi k_z p\end{bmatrix} \quad H_v = J\left(k_x^2 + k_y^2 + k_z^2\right)\begin{bmatrix}0\\ \partial u/\partial k\\ \partial v/\partial k\\ \partial w/\partial k\end{bmatrix}$$

where $U = k_t + uk_x + vk_y + wk_z$. When k equals to ξ, η, ζ, H equals to E, F, G, and H_v equals to E_v, F_v, G_v. J is the Jacobian matrix of coordinates transformations. k_x, k_y, k_z are grid derivatives of coordinates transformations.

The governing Equations (6) are discretized by a cell-centered finite-volume scheme and the numerical interface fluxes are calculated by upwind biased flux

differences. The LU-SGS (Lower-Upper Symmetric Gauss-Seidel) scheme is used to solve the nonlinear system of equations. The numerical efficiency and accuracy of this method have been validated by simulation performed for some steady and unsteady flow problems [1, 2].

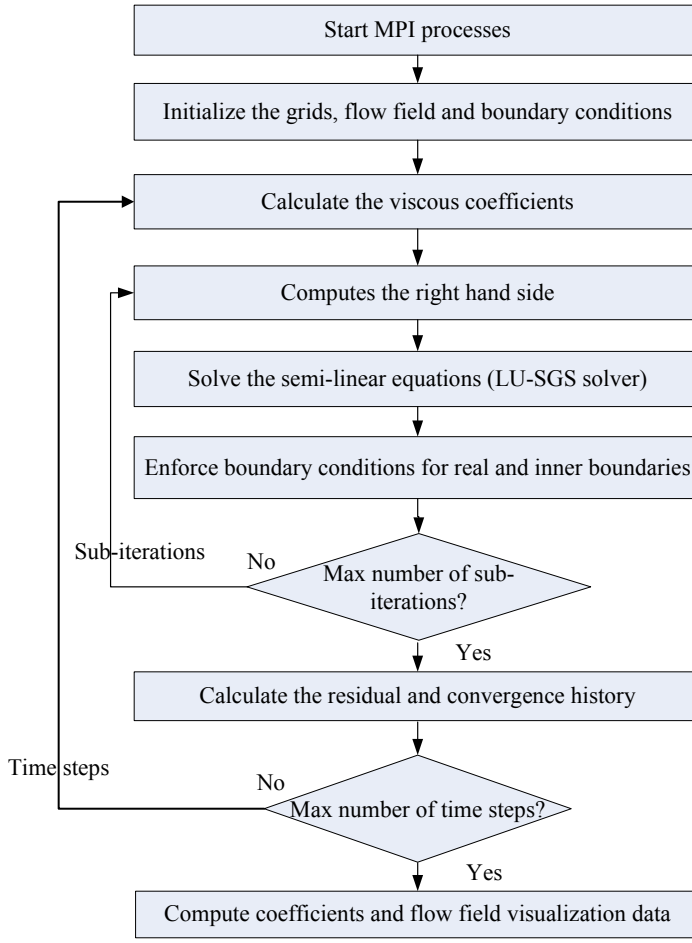


Figure 1. The high-level flow chart of LM3D

LM3D code is originally written in FORTRAN 77. Its flow data is stored at the volume center, with the interfacial fluxes determined through interpolation of cell-centered values. It is paralleled by domain decomposition method [3, 4] and MPI (Message Passing Interface) is used for inter-process communications. The grids of the whole computation domain are partitioned into multiple grid blocks. These grid blocks are then distributed to the MPI processes. Each process (including the

root) is assigned one or more distinct grid blocks. As LM3D is a CFD solver of second order accuracy, the ghost cell width between neighboring grid blocks is 2 in LM3D. Figure 1 shows the high-level flow chart of LM3D. The program first starts MPI processes, and initializes the grid, flow field and boundary conditions. Then it enters a time step loop, in which it spends most of its execution time. In each time step, it executes the following procedures:

1. Calculating the viscous coefficients. It first initializes the field variables for laminar problems, then computes the vorticity. If turbulent flow is involved, it computes the eddy viscous coefficient.
2. A number of sub-iterations. The iterative algorithm is based on Newton implicit schemes with line relaxation Gauss-Seidel sub-iterations. In each sub-iteration, it first computes the right hand side, then calls the LU-SGS solver to solve the semi-linear equations, and then exchanges boundary data to impose boundary conditions for real boundaries and inner boundaries. Exchanging of boundary data involves MPI communications among neighboring processes.
3. Calculating the residual and the convergency history of drag coefficient, lift coefficient, etc. For residual calculation, each process first calculates the local residuals of density, divergence, etc., then each non-root process sends the local residuals to the root process, which calculates the global residuals. MPI communications between the root process and other processes are involved. For convergency history calculation, similar computation and communication patterns are involved.

After the time step loop, LM3D first computes the coefficient of surface pressure distribution, vorticity distribution, etc. Then it computes the flow field visualization data, outputs them in PLOT3D file format, then exits.

CFD is known to be one of the grand challenge application areas of HPC because it is mainly limited by the amount of time to compute a solution. For example, applying CFD simulations to wing shape optimization requires a large number of similar simulations with different input parameters. Therefore, beside the progress in CFD models and algorithms, further progress in CFD is closely related with efficient usage of modern HPC systems. Many prior efforts [5, 6, 7, 8, 9, 10, 11] have been paid on the efficient implementation of CFD applications on parallel computers. The performance is also crucial for LM3D since it requires days of time to achieve a usable solution. When LM3D was parallelized with MPI, several techniques to improve the parallel performance were considered. These techniques include load balancing, data localization, communication minimization and combination. This code has shown good parallel scalability in our preliminary performance evaluation. However, its computation efficiency is not satisfactory. The floating point efficiency achieved is below 4 % of the peak even for a single process test on an Intel Xeon X5670 processor. Furthermore, in the startup phase of LM3D, the data are serially read from the disks. So the startup speed is very slow as compared to the computation speed, as the latter has been accelerated by using large number of CPU cores. The mismatch between

the startup speed and the computation speed also limits the parallel scalability for large-scale simulations. When scaling to larger problem sizes (and consequently more CPU cores), the startup will become the performance bottleneck. This work aims to tackle these problems and optimize LM3D's performance and scalability for large-scale parallel computers. High level loop transformations, argument based code specialization and value profile based optimization are applied to optimize its uniprocessor performance. Static arrays are converted into dynamically allocated arrays to improve the flexibility. The grid generator is coupled with the solver to enable online grid generation and in memory data exchange. Performance evaluation is performed on the Tianhe-1A supercomputer [14, 15]. The performance results demonstrate the effectiveness of our optimization methods.

2 OPTIMIZATION STRATEGIES

Application performance optimizations should based on the characteristics of the application code. Although there are many prior performance optimization efforts for CFD applications, they are not directly applicable to LM3D. Based on the in-depth performance measurement and analysis of the original LM3D code, three classes of optimization strategies are applied to improve its performance as well as flexibility.

2.1 Uniprocessor Performance Optimization

Uniprocessor performance is crucial for a parallel code to attain high performance on a parallel computer [16]. Current uniprocessor performance optimization methods involve many code transformations that improve the data access locality and parallelism such as ILP (Instruction Level Parallelism) to better utilize the processor's architecture characteristics. To maximize the performance gains, we first profile the program with Gprof to locate those computation intensive subroutines. As a typical Navier-Stokes equations based CFD code, LM3D spends most of its time in two phases: flux computations (to evaluate conservation law residuals) and sparse linear algebraic kernels. Ten kernel subroutines take over 90 % of total runtime. Then we examine the source code of these kernel subroutines to find optimization opportunities. The following optimization transformations are found to be feasible for LM3D code and are applied.

Loop permutation. LM3D stores its main data in several four dimensional arrays, with the last three dimensions describe three coordinate indexes and the first dimension describes different physical variables. For example, the array Q, which stores the primitive variables, has the form Q(4, -1:jmax+1, -1:kmax+1, -1:lmax+1), with Q(1, :, :, :), Q(2, :, :, :), Q(3, :, :, :) and Q(4, :, :, :) store ρ, u, v and w, respectively. Array QBAR (stores the values of ΔQ) and array Q1 (stores the newly calculated values of the primitive variables) are of the same form as array Q. Many computation intensive loop nests access these arrays

and for better cache performance, they must access array elements along the fast changing dimensions. Based on dependency analysis, loop permutation is applied to those loop nests that do not access array element along the fast changing dimensions. Figure 2 presents an example of loop permutation in LM3D. Figure 2 a) is the loop nest in (J, K, L) order. Figure 2 b) is the transformed loop nest in (L, K, J) order, in which the loop nest accesses array elements with much smaller strides, and hence exhibits better spatial locality of memory access. This transformation also introduces additional compiler optimization opportunities (e.g, inner loop vectorization) for the compiler. We should note that current compilers can perform loop permutation automatically. However, due to the inability of dependency analysis, the compilers are conservative in doing so. Our manual loop permutation of LM3D improves the performance, even when an aggressive compiler optimization switch (-O3) is used.

```
DO 100 J = 1, JM
DO 100 K = 1, KM
DO 100 L = 1, LM
  ...QBAR(*, J, K, L)...
  ...Q(*, J, K, L) ...
  ...Q1(*, J, K, L) ...
100 CONTINUE
```

a) The original loop nest (J, K, L)

```
DO 100 L = 1, LM
DO 100 K = 1, KM
DO 100 J = 1, JM
  ...QBAR(*, J, K, L)...
  ...Q(*, J, K, L) ...
  ...Q1(*, J, K, L)...
100 CONTINUE
```

b) The permuted loop nest (L, K, J)

Figure 2. An example of loop permutation in LM3D

Full loop unrolling. LM3D uses four primitive flow variables (ρ, u, v, w), so most of its arrays have a leading dimension of 4. Figure 3 shows the definition of these arrays.

```
Q(4, -1:JMAX+1, -1:KMAX+1, -1:LMAX+1)
Q1(4, -1:JMAX+1, -1:KMAX+1, -1:LMAX+1)
QBAR(4, -1:JMAX+1, -1:KMAX+1, -1:LMAX+1)
S(4, JMAX, KMAX, LMAX)
```

Figure 3. The definitions of main arrays, with the leading dimension of 4

Hence, many inner loops that access these arrays have a constant loop count of 4. For these inner loops, full loop unrolling is applied. Figure 4 presents an example of full loop unrolling in LM3D. Figure 4 a) is the original loop, where array DSPR stores the upper limit of the sum of spectra radius of Jacobian matrix. In Figure 4 b), the loop is unrolled for four times and the statements that control the loop are totally eliminated. Such transformation not only increases the ILP, but also eliminates the loop related branch penalties.

Argument-based code specialization. Some kernel subroutines are programmed in a general way that can accept arbitrary argument values but are called with some fixed argument values. These subroutines can be specialized and

```
DO N = 1, 4
  QBAR(N, J, K, L) = DSPR(J, K, L)*QBAR(N, J, K, L)
ENDDO
```

a) The original loop

```
QBAR(1, J, K, L) = DSPR(J, K, L)*QBAR(1, J, K, L)
QBAR(2, J, K, L) = DSPR(J, K, L)*QBAR(2, J, K, L)
QBAR(3, J, K, L) = DSPR(J, K, L)*QBAR(3, J, K, L)
QBAR(4, J, K, L) = DSPR(J, K, L)*QBAR(4, J, K, L)
```

b) The code lines after full loop unrolling

Figure 4. An example of full loop unrolling in LM3D

optimized for the actual argument values. Figure 5 presents an example of argument-based code specialization in LM3D, where MVMULTI is a subroutine that multiplies a matrix by a vector. It takes an argument ND, which is the dimension size of the matrix. In the whole program of LM3D, MVMULTI is called with ND equals 3 or 4. So we first modify the subroutine with two versions of codes, one for ND = 3 and another for ND = 4. Then for the two versions of MVMULTI, additional optimizations (e.g., full loop unrolling) are applied because the loop count variable ND is replaced by known constant values (3 and 4) now. We should note that while current compilers may automatically apply the loop unrolling shown in Figure 4, they can not automatically apply the full loop unrolling for the loops shown in Figure 5 a).

Value profile based optimization. In LM3D code, intrinsic math functions POW (returns the value of a base expression taken to a specified power) and SQRT (derives the square root of its argument) are time consuming. These functions take arguments whose values depend on one variable ESIGMA. Figure 6 lists several expressions that use ESIGMA in the calculation of intrinsic math functions.

ESIGMA is the variable corresponding to σ in Equation (3) and is read from a name list stored in a configuration file. ESIGMA does not change in a simulation run, that is, it has a value profile of a constant. So we can optimize the performance by embedding its value to the source code. This is done by defining ESIGMA as constant variable and assigning its value with the value stored in the name list. This enables the compiler to perform some optimizations, such as constant propagation and pre-computing. Such optimization can eliminates some calculations that are related to these math functions.

2.2 Dynamic Memory Allocation

The original LM3D code is written in FORTRAN 77 format. It uses static arrays, which means that the codes must be recompiled for different problem sizes or different numbers of processors in new runs. We modify the code into FORTRAN 90

```
SUBROUTINE MVMULTI(MATR, MVE0, MVE1, ND)
DIMENSION MATR(ND, ND), MVE0(ND), MVE1(ND)
DO N = 1, ND
  MVE1(N) = 0
  DO M = 1, ND
    MVE1(N) = MVE1(N) + MATR(N,M)*MVE0(M)
  ENDDO
ENDDO
```

a The original subroutine

```
SUBROUTINE MVMULTI(MATR, MVE0, MVE1, ND)
DIMENSION MATR(ND, ND), MVE0(ND), MVE1(ND)
IF(ND .EQ. 3)THEN
  DO N = 1, 3
    MVE1(N) = 0
    DO M = 1, 3
    MVE1(N) = MVE1(N)+ MATR(N, M)*MVE0(M)
    ENDDO
  ENDDO
ELSE
  DO N = 1, 4
    MVE1(N) = 0
    DO M = 1, 4
    MVE1(N) = MVE1(N)+ MATR(N, M)*MVE0(M)
    ENDDO
  ENDDO
ENDIF
```

b) The subroutine after argument-based specialization

Figure 5. An example of argument-based code specialization in LM3D

```
CCA2 = AIRINF2*EXXX*ZO**(ESIGMA-1.0)
CCA2 =AIRINF*SQRT(EXXX*ZO**(ESIGMA-1.0))
BIR = AIRINF*SQRT(EXXX*ZA**(ESIGMA-1.0))
PLL = AIRINF2*ZLL**ESIGMA
```

Figure 6. Expressions that use variable ESIGMA

format with dynamic memory allocation. All major arrays are identified and defined in data modules as allocatable arrays with deferred shapes. These arrays are grouped based on their life cycles and corresponding memory allocation subroutines are designed for each group. The memory allocation subroutines determine the size of each array based on the input arguments, such as the number of grid points in three dimensions and the parameters of the three dimensional processor topology. Accordingly, memory deallocation subroutines are also designed. Memory allocation and deallocation subroutines are called at appropriate place to minimize the

number of memory allocation operations. Figure 7 shows the new definition and allocation sentences of arrays listed in Figure 3. Dynamic memory allocation makes it flexible to change the test configurations at runtime. Once the code is compiled into executable for a specific platform, the executable can be used for different test cases.

```
! Define Q, Q1, QBAR, S as ALLOCATABLE arrays
REAL, DIMENSION(:, :, :, :), ALLOCATABLE::Q, Q1,
QBAR, S

! Allocate memory space for Q, Q1, QBAR, S
ALLOCATE(Q(4, -1:jmax+1, -1:kmax+1, -1:lmax+1),
stat=nerror(1))
ALLOCATE(Q1(4, -1:jmax+1, -1:kmax+1, -1:lmax+1),
stat=nerror(2))
ALLOCATE(QBAR(4, -1:jmax+1, -1:kmax+1, -1:lmax+1),
stat=nerror(3))
ALLOCATE(S(4, jmax, kmax, lmax), stat=nerror(4))
```

Figure 7. New definition and allocation sentences of arrays in Figure 3

2.3 In Memory Data Exchange

The original LM3D code has a grid generator, which is a stand alone application. The grid generator generates the grid data and writes them to the grid file. Then the solver reads the grid data from the disk to the memory, initializes the flow field data and performs the simulation. This arouses performance issues because the grid data are written to and read from the much slower disk. For large scale parallel computing, grid I/O may potentially grow into performance bottleneck because of the disparity in the rate of improvement in computational power compared to storage throughput [12]. To alleviate this problem, we couple the grid generator with the solver as a single executable and enable them to directly exchange data in the memory. The grid generator is encapsulated into a subroutine that is called by the main program of the solver. It takes the grid shape and the number of sub-blocks as input arguments. As we have declared major array in data modules, it is convenient for the grid generation routine to pass grid data to the solver routines via the global array module. This avoids writing the grid files to the disk and reading them from the disk, which greatly accelerates the simulation startup process. It can also be configured to output the grid files to the disk if needed.

Coupling the grid generator with the solver may have a negative effect on the flexibility of the problem that can be solved by the code because the capabilities of the grid generator are rather limited. Another issue related to this method is that the grid generation may become the bottleneck that limits the problem size. If the grid generation task is executed by only one process, it generates the grid data of

the whole flow field and sends them to other processes. However, the memory on a compute node may not fit all the grid data of the whole flow field. This issue can be settled by generating and sending grid data for each process one by one, or designing parallel grid generation algorithm to distribute the grid generation task to more processes. For the test case used in this paper, a parallel grid generation algorithm is designed and each process generates its own grid data.

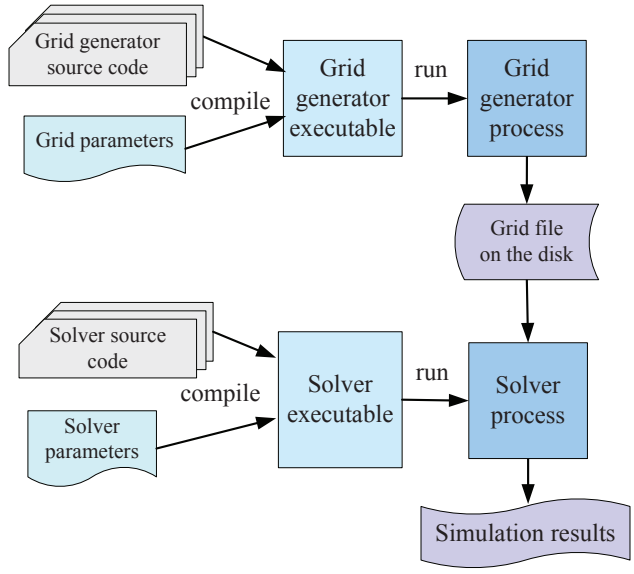
By dynamic memory allocation, coupling of the grid generator with the solver and in memory grid exchange, we have significantly reformed the work flow of LM3D. Figure 8 illustrates the work flow before and after the optimizations. Originally, the grid generator and the solver are compiled separately along with their parameters defined in common parameter files. Then they execute separately: the grid generator process generates the grid file and stores it on the disk; the solver process reads the grid file from the disk and performs the simulation. In the optimized version, the grid generator and the solver code are compiled into one executable. They run in the same process, accept runtime parameters and exchange grid data in the memory. It is clear that the optimized version is more efficient by eliminating the serial and time consuming grid file I/O operations. To our knowledge, the TAU (Parallel Architecture of TAU-Code) [13] employs a similar in memory data access method between modules.

3 PERFORMANCE EVALUATION

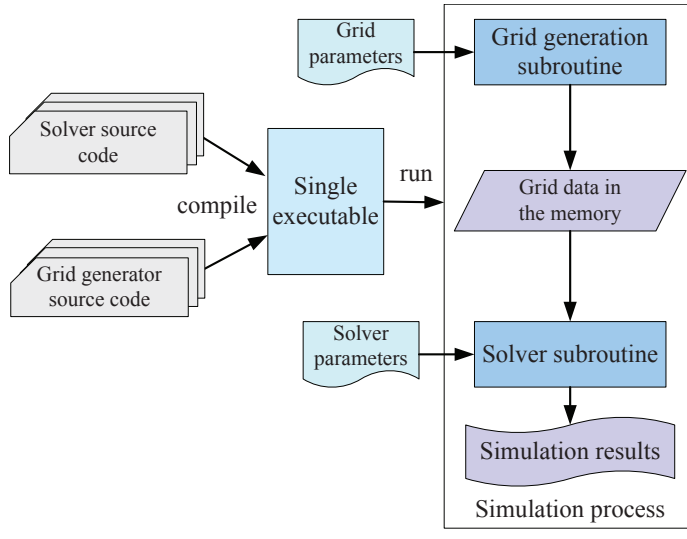
3.1 Test Setups

Since the accuracy of LM3D has been validated by previous works [1, 2], we focus our attentions on the performance evaluation. We mainly evaluate the performance of LM3D on the Tianhe-1A supercomputer installed at the National Supercomputing Center in Tianjin, China [15]. Tianhe-1A has a theoretical peak performance of 4.70 Pflops and a measured Linpack performance of 2.57 Pflops. It ranks No. 1 on November 2010 and No. 10 on June 2013 in the top500 list. Table 1 presents the hardware and software setup of the Tianhe-1A [17]. In our test, the NVIDIA GPUs and the FT-1000 CPUs are not used. We select the aggressive optimization switch -O3 (Maximize Speed plus Higher Level Optimizations). This optimization switch will enable most compiler optimizations, including memory access optimizations and automatic vectorization to utilize the 128-bit SSE instructions on the Intel Xeon X5670 CPUs.

Most modern processors include special hardware units called Performance Monitoring Units (PMUs) to detect and count certain microarchitectural events from several hardware sources such as the pipeline, system bus or memory hierarchy. Such events offer facilities to characterize the interaction between the application and the hardware, hence provide a more precise picture of the hardware resource utilization. We also measure the micro-architecture level performance data based on PMU to see the effects our uniprocessor optimizations. As Tianhe-1A is a heavily loaded supercomputer shared by many concurrent users, it is not convenient to measure



a)



b)

Figure 8. Work flow before and after the optimization: a) the original work flow, b) the optimized work flow

Items	**Configuration**
Nodes	7168 computing nodes, each has two Intel CPUs and one NVIDIA GPU 1 024 service nodes, each has two FT-1000 CPUs
Processors	14 336 Intel Xeon X5670 CPUs (2.93 GHz, six-core) 7 168 NVIDIA M2050 GPUs (1.15 GHz, 14 cores) 2 048 FT-1000 CPUs (1.0 GHz, eight-core)
Interconnect	Custom-built interconnect organized in an fat-tree structure with the bi-directional bandwidth of 160 Gbps, the latency of 1.57μs
Memory	Each compute node has 32 GB memory. The total memory size is 262 TB.
I/O system	Luster file system with 6 I/O management nodes, 128 I/O storage nodes, and 2 PB disk capacity from above 16 000 SATA drives.
OS	Kylin Linux
Compiler	Intel FORTRAN Compiler version 11.1 (Optimization switch: -O3)
MPI	MPICH2 version 1.2.1p1-g16

Table 1. The hardware and software setup of the Tianhe-1A supercomputer

the micro-architecture level performance data on its X5670 CPU. So we choose to measure the micro-architecture level performance data on a stand alone desktop computer. This desktop computer is equipped with an Intel Core i3 550 3.2 GHz CPU, which is also based on the Intel Nehalem micro-architecture, like the Intel X5670. This CPU has two cores. Each core has a 32 KB L1 Instruction Cache, a 32 KB L1 Data Cache and a unified 256 KB L2 Cache. A total of 4 MB Last Level Cache (LLC) is shared by the two cores. The desktop has 2GB memory. The program is compiled with Intel Fortran Compiler 11.1, and the optimization switch used is -O3. We use the Intel Vtune [18] to measure the microarchitecture level performance data. Intel Vtune is a powerful tool that can collect microarchitecture level performance data of the Intel CPUs through the on chip PMUs. The version used is the Intel Vtune Amplifier XE 2013 update 2. The Hyper-Threading feature of Intel Core i3 550 CPU is disabled through the BIOS to avoid its impact to the performance of a single threaded application.

The different versions of the LM3D code tested are designated as follows. The original version is designated as ORI. OPT1 is the version with uniprocessor optimizations and dynamic array allocation. In the OPT1 version the grid generator and the solver are enclosed in one program but they exchange grid data by grid file on the disk. OPT2 is the version after the OPT1 version is modified that the grid generator and the solver exchange grid data in the memory. The three versions of the codes differ only in the implementations that do not affect the numerical algorithms. So they produce the same results and their convergence characteristics are strictly identical.

The test case used is the viscous flow around a prolate spheroid. Many flow related physical variables, such as the speed at three directions, the surface vorticity

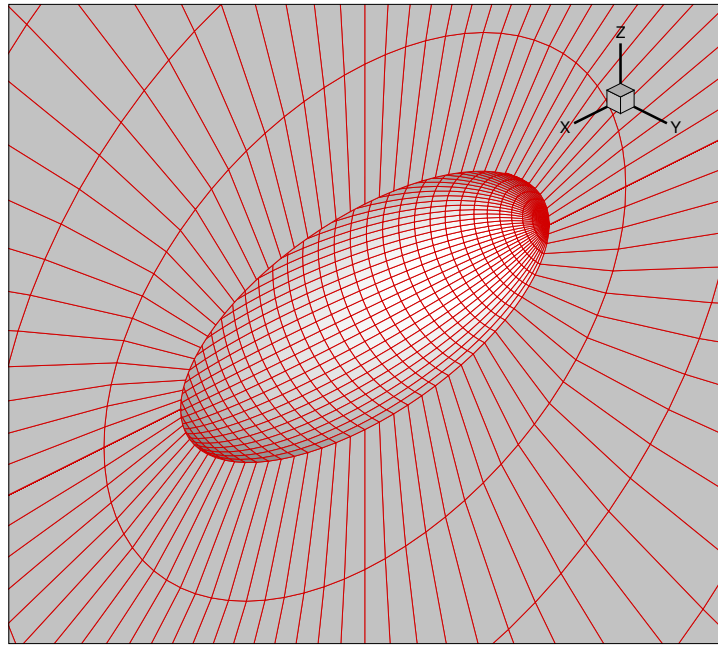


Figure 9. The small scale grids near the surface of the test case

distribution at symmetric plane and the surface pressure distribution at symmetric plane, are calculated. In the test, the ratio of long axis to short axis of this prolate spheroid is 2.0. The radius of far field boundary is 19.5. The grid stretching ratio along normal direction is 1.1678. Figure 9 shows the small scale grids near the surface of this test case. The simple geometry allows for straightforward partitioning, simplifying the evaluation of performance as the grid density and the number of CPU cores are varied. As for the flow field parameters, the free stream Mach number is 0.2, the angle of attack is 10°.

3.2 PMU Based Performance Test on the Desktop Computer

We first perform the PMU based test to measure the micro-architecture level performance data on the desktop computer. We compare the ORI version to the OPT1 version to see the effects of our uniprocessor optimizations. The two versions of codes run serially in the test. The grid resolution used is $241 \times 241 \times 81$ (4.49 M grid points). For this grid resolution, the ORI version consumes about 735 MB memory, and the OPT1 version consumes about 760 MB memory. The number of sub-steps in each time step is 6. We use a two-phase measurement method. In the first phase, we measure the whole application that executes 5 time steps. In the second phase, we measure the whole application that executes 15 time steps. Then we subtract the

data measured in the second phase with the data measured in the first phase and get the data of 10 "pure" time steps. We select several key PMU events that exhibit the instruction execution, the branch prediction and the memory access characteristics of the program. The descriptions of the PMU events can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual [19]. Figure 10 shows the number of PMU events measured on the desktop computer for the two versions of codes. It can be seen that the number of branch instructions executed and the number of mispredicted branches of the OPT1 version is much less than that of the ORI version. This indicates that our loop unrolling optimizations are effective, and these optimizations are not applied automatically by the compiler for the original code. The number of L1 data cache accesses, the number of L2 cache data accesses, the number of L2 cache misses, the number of retired loads that miss the LLC cache and number of Data Translation Look-aside Buffer (DTLB) misses of the OPT1 version are significantly less than that of the ORI version, indicating the effectiveness of our memory optimization methods. The number of instructions executed of the OPT1 version is also slightly reduced as a result of less branch instructions executed and less number of mispredicted branches. Overall, our optimizations have greatly reduced the execution time. The number of clock cycles executed of the ORI version in the 10 "pure" time steps is about 102.58 billions, whereas the number of clock cycles executed of the OPT1 version in the 10 "pure" time steps is only about 31.44 billions.

3.3 Solver Performance Test on the Tianhe-1A Supercomputer

The solver performance test on the Tianhe-1A aims to evaluate the performance improvement of our uniprocessor optimizations to the parallel LM3D codes, and to evaluate the parallel scalability feature of the optimized LM3D code. We first compare the performance of the ORI version and the OPT1 version to see the performance improvement of the uniprocessor optimizations to the parallel code. Two grid resolutions are used in the test: $481 \times 481 \times 81$ (17.87 M grid points), $961 \times 961 \times 81$ (71.34 M grid points). For each grid resolution, a set of runs with the number of CPU cores ranging from 1 to 4 096 are examined. We run each test case only 100 time steps instead of running it until convergence to a steady-state. Only the runtime of 100 time steps is measured and the time spent on grid generation and flow field initialization is not counted. The number of sub-steps in each time step is 6. Figure 11 shows the performance improvement of the uniprocessor optimizations for the parallel code on Tianhe-1A. The performance improvement ratio is computed by dividing the runtime of the ORI version with the runtime of the OPT1 version. This is also the ratio of the solution speed improvement as the convergence characteristics of the ORI version and the OPT1 version are the same. We see that our uniprocessor optimizations are effective for the parallel code even in the presence of the highly optimized Intel compiler with aggressive compiler optimization level. The OPT1 version outperforms the original ORI version by 1.38× to 3.93×. The ratio of performance improvement drops with the number of CPU cores because

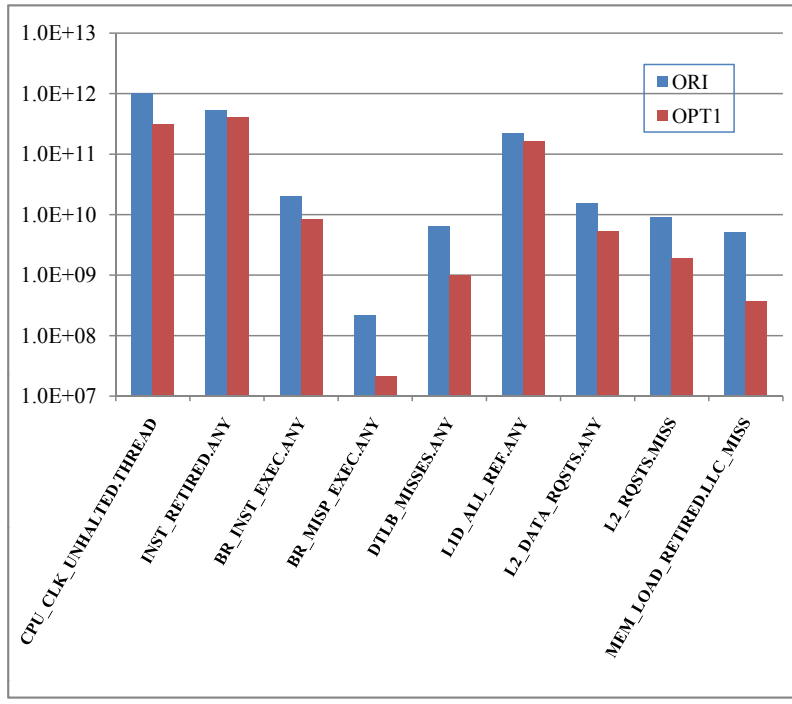


Figure 10. The number of PMU events measured on an Intel Core i3 processor based desktop computer for a grid size of 4.70 M for the ORI version and the OPT1 version

when the number of CPU cores increases, the cost of communication increases. The performance improvement is more significant for larger grid size because when the grid size increases, the computation to communication ratio also increases.

For the parallel scalability test we use the OPT2 version as this version has a fast startup with the in memory grid data exchange optimization. Besides the two previous grid resolutions, an additional and larger grid resolution, $1\,921 \times 1\,921 \times 81$ (286.06 M grid points), is also used in the test. We have performed test runs with grid sizes that exceed 1 giga grid point and the number of CPU cores reaches 16384. However, we are not able to perform comprehensive tests because the Tianhe-1A system is heavily loaded. Here we only report the strong scaling data of tests with the maximum number of CPU cores of 4 096. Figure 12 shows the floating-point performance in Gflops of the OPT2 version when using 2^n CPU Cores. When the number of CPU cores is large, the floating-point performance achieved for the case of 286.04 M grid points is higher than that achieved for the other two cases. It achieves a sustained floating-point rate of 4 101.77 Gflops (4.01 Tflops) for the case of 286.04 M grid points on 4 096 CPU cores, which is proximally 8.54 % of the peak floating point performance of these CPU cores. To our knowledge, achieving such

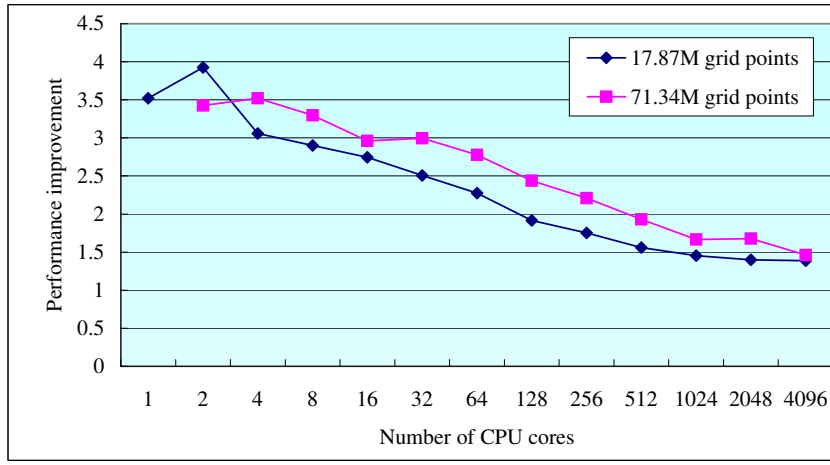


Figure 11. The performance improvement of the uniprocessor optimizations on Tianhe-1A (the OPT1 version versus the ORI version)

a high floating point efficiency at this scale is encouraging for CFD codes that solve Navier-Stokes equations on structured grids.

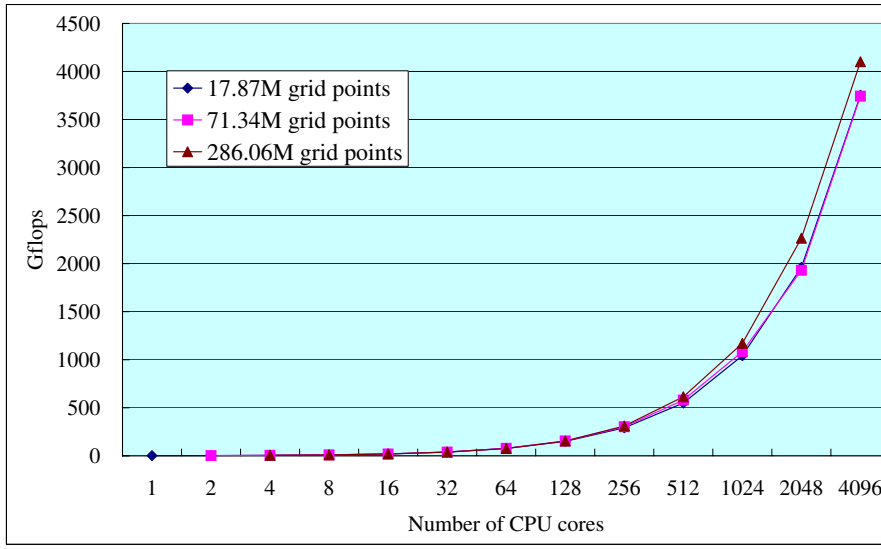


Figure 12. The floating-point performance of the OPT2 version on Tianhe-1A

Figure 13 shows the parallel efficiency of the OPT2 version computed based on data in Figure 12. We see that the code scales excellently in these strong scaling

test cases. The parallel efficiency decreases slowly with the number of CPU cores and reaches 69.13 %, 68.85 % and 77.39 %, respectively for the cases of 17.87 M grid points, 71.34 M grid points and 286.04 M grid points when the number of CPU cores is 4 096. The excellent parallel scaling behaviors can be attributed to the high degree of parallelism, the high computation to the communication ratio and the high speed of the interconnect. The decrease of parallel efficiency for larger number of CPU cores can be attributed to the increase of communication to computation ratio as well as the communication resource contentions.

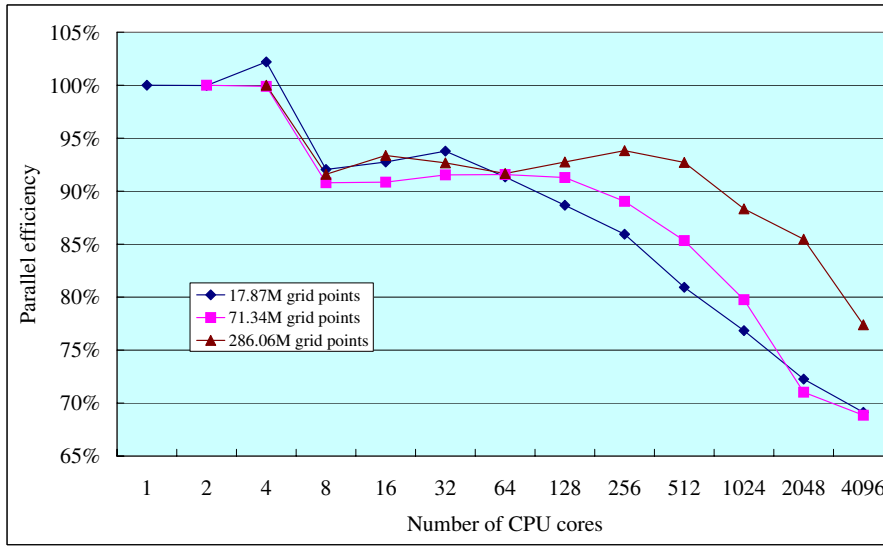


Figure 13. The parallel efficiency of the OPT2 version on Tianhe-1A

3.4 Startup Performance Test on Tianhe-1A

The startup performance test aims to investigate the effect of in memory grid exchange method to the startup speed. We measure the startup time of the OPT2 version and the OPT1 version. Then we calculate the speedup of the OPT2 version over the OPT1 version. For the OPT1 version, the startup time includes the time spent on writing and reading the grid file. The data for the case of 286.06 M grid points are not measured because for this grid resolution, the OPT1 version spends too long startup time while a large number of processes are idle waiting for the grid data. The size of the grid file is approximately 0.83 GB for the case of 17.87 M grid and 3.36 GB for the case of 71.34 M grid points. The startup time of the two versions of codes and the corresponding speedup are shown in Table 2. We see that the OPT2 version spends much less time on startup than the OPT1 version. The speedup of the OPT2 version over the OPT1 version ranges from 58.99 to 95.00,

as considered for the startup time. This indicates that the application startup performance is improved by nearly two magnitudes, which will greatly facilitates large scale parallel runs.

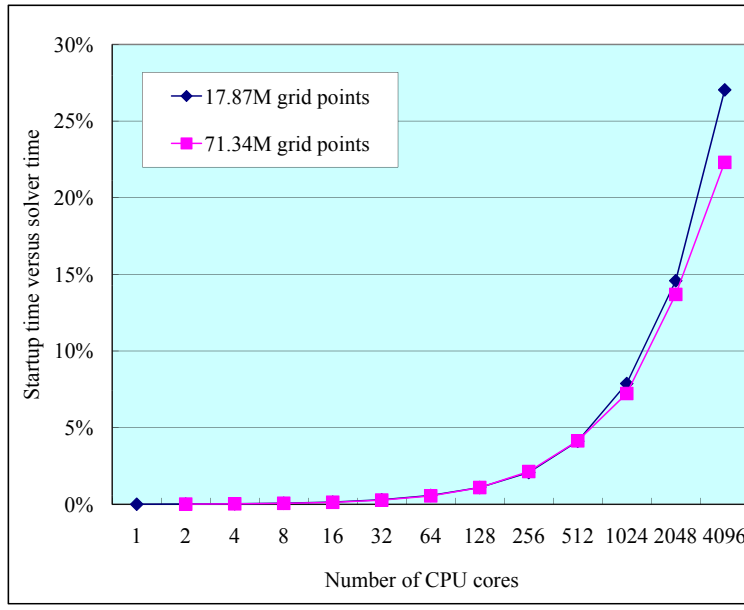
	17.87 **M grid points**			71.34 **M grid points**		
# Cores	OPT1	OPT2	Speedup	OPT1	OPT2	Speedup
1	142.5	1.50	95.00	–	–	–
2	146.1	1.58	92.47	553.1	8.31	66.56
4	146.3	1.59	92.01	555.6	8.45	65.75
8	145.9	1.65	88.42	551.6	8.52	64.74
16	146.8	1.69	86.86	551.9	7.56	73.00
32	147.3	1.66	88.73	558.0	7.59	73.52
64	146.8	1.64	89.51	552.8	8.00	69.10
128	142.5	1.75	81.43	551.6	7.86	70.18
256	141.3	1.88	75.16	553.3	8.46	65.40
512	145.0	1.98	73.23	562.0	8.83	63.65
1 024	146.8	2.05	71.61	557.6	9.01	61.89
2 048	146.5	2.05	71.46	557.4	9.23	60.39
4 096	146.9	2.09	70.29	557.5	9.45	58.99

Table 2. The startup time in seconds and the corresponding speedup (the OPT2 version over the OPT1 version) on Tianhe-1A

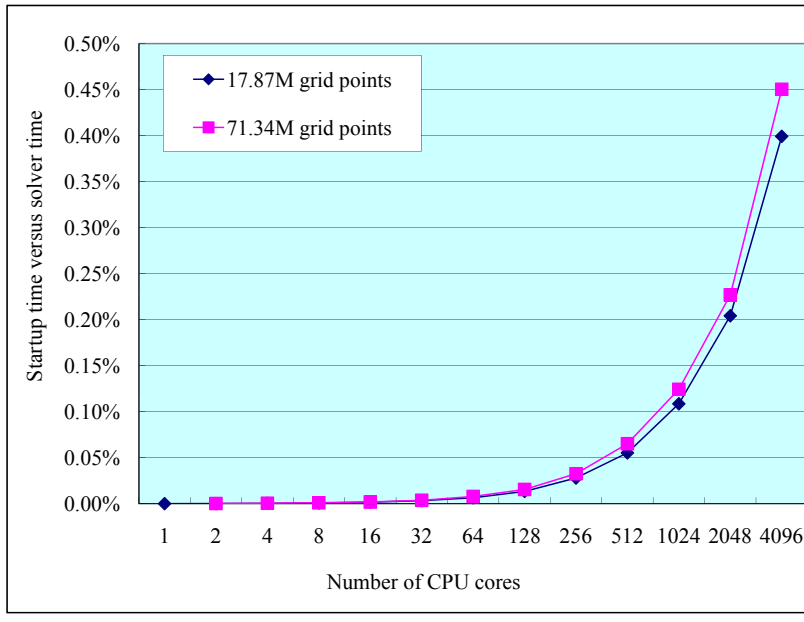
Figure 14 shows the ratio of the startup time versus the solver time. Figure 14 a) is the ratio for the OPT1 version. Figure 14 b) is the ratio for the OPT2 version. We know from previous experiments that the simulation will converge within 10 000 time steps. We also observed that each time step costs nearly the same runtime. So we measure runtime of 100 time steps, scale it by 100 times and use the scaled runtime as the solver time. Hence, the solver time used is approximately the runtime of 10 000 time steps. For the OPT1 version, the ratio of the startup time versus the solver time grows rapidly with the number of CPU cores and exceeds 25 % for 4 096 CPU cores. It is clear that the startup time has grown into a main part of the total runtime when a large number of CPU cores are used. For the OPT2 version, with in memory grid exchange, this ratio is greatly reduced to below 0.5 % for all cases. This suggests that enabling in memory data exchange between the grid generator and the solver may greatly boost the performance of large-scale parallel CFD simulation flow. This is especially useful in the industrial design optimization process.

4 CONCLUSION

In this work, we have optimized LM3D, a parallel three dimensional finite volume CFD code based on the Slightly Compressible Model. High level loop transformations, argument based code specialization and value profile based optimization are used to improve its uniprocessor performance. Dynamic array allocation is used to improve its flexibility. The grid generator is coupled with the solver to eliminate



a)



b)

Figure 14. The ratio of startup time versus the solver time for the OPT1 version and the OPT2 version on Tianhe-1A: a) the OPT1 version, b) the OPT2 version

grid I/O related overhead. Performance evaluation is performed on the Tianhe-1A supercomputer. Experimental results demonstrate the importance and effectiveness of these optimization methods. The uniprocessor optimizations improve the parallel simulation performance for 1.38× to 3.93× on Tianhe-1A. The optimized code scales excellently on Tianhe-1A. It achieves a maximum floating-point performance of 4.01 Tflops when use of 4 096 CPU cores on Tianhe-1A. Furthermore, the application startup performance is improved by nearly two magnitudes.

Acknowledgements

We thank Xiaogang Deng and Daping Xiang of CARDC for the original LM3D code and technical help. We also thank the National Supercomputing Center in Tianjin, China for providing the Tianhe-1A supercomputer for the performance evaluation. This work was partially supported by the Major State Basic Research Development Program of China (973 Program) under Grant No. 2009CB723803 and the National Natural Science Foundation of China under Grant Nos. 60603055, 11272352, 61103014 and 61379056.

REFERENCES

[1] Deng, X.—Zhuang, F.—Mao, M.: On Low Mach Number Perfect gas Flow Calculations. 14th AIAA Computational Fluid Dynamics Conference, 1999, pp. 595–604.

[2] Xiang, D.—Deng, X.—Mao, M.: Study on a Novel Method for Low Mach Number Flows Computation. Acta Aerodynamica Sinica, Vol.20, 2002, No. 4, pp. 373–378 (In Chinese with English abstract).

[3] Xiang, D.—Deng, X.—Mao, M.: Parallel Computation for Low Mach Number Flows. Acta Aerodynamica Sinica, Vol. 20, 2002, pp. 77–81 (In Chinese with English abstract).

[4] Wang, Z.—Che, Y.: Parallel Implementation of a Low Mach Flow Simulator and Its Performance Analysis. Acta Aerodynamica Sinica, Vol. 20, 2002, pp. 82–87 (In Chinese with English abstract).

[5] Gropp, W. D.—Kaushik, D. K.—Keyes, D. E.—Smith, B. F.: Performance Modeling and Tuning of an Unstructured Mesh CFD Application. ACM/IEEE 2000 Conference on Supercomputing, pp. 34, DOI: 10.1109/SC.2000.10059.

[6] Andres, E.—Widhalm, M.—Caloto, A.: Achieving High Speed CFD Simulations: Optimization, Parallelization, and FPGA Acceleration for the Unstructured DLR TAU Code. 47th AIAA Aerospace Sciences Meeting, January 5–8, 2009, Orlando, Florida.

[7] Gorobets, A. V.—Borrell, R.—Trias, F. X.—Kozubskaya, T. K.—Oliva, A.: Efficiency of Large-Scale CFD Simulations on Modern Supercomputers Using Thousands of CPUs and Hybrid MPI + OPENMP Parallelization. Proceedings of the Fifth European Conference on Computational Fluid Dynamics, Lisbon, Portugal, June 14–17, 2010, 12 pp.

[8] Gourdain, N.—Gicquel, L.—Montagnac, M.—Vermorel, O.—Gazaix, M.—Staffelbach, G.—Garcia, M.—Boussuge, J.-F.—Poinsot, T.: High Performance Parallel Computing of Flows in Complex Geometries – Part 1: Methods. Computational Science and Discovery, TR_CFD_09_117.

[9] Duffy, A. C.—Hammond, D. P.—Nielsen, E. J.: Production Level CFD Code Acceleration for Hybrid Many-Core Architectures. NASA/TM-2012-217770, Langley Research Center, October 2012.

[10] Xu, C.—Deng, X.—Zhang, L.—Jiang, Y.—Cao, W.—Fang, J.—Che, Y.—Wang, Y.—Liu, W.: Parallelizing a High-Order CFD Software for 3D, Multi-Block, Structural Grids on the TianHe-1A Supercomputer. 28th International Supercomputing Conference (ISC 2013), LNCS, Vol. 7905, 2013, pp. 26–39.

[11] Lin, P.-H.—Jayaraj, J.—Woodward, P. R.—Yew, P.-C.: A Code Transformation Framework for Scientific Applications on Structured Grids. Technical Report 11-021, UMN Computer Science and Engineering Technical Report, 2011.

[12] Lang, S.—Carns, P.—Latham, R.—Ross, R.—Harms, K.—Allcock, W.: I/O Performance Challenges at Leadership Scale. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis – Supercomputing 2009 (SC '09), November 14–20, 2009, Portland, Oregon, USA, Article No. 40, DOI: 10.1145/1654059.1654100.

[13] Schwamborn, D.—Gerhold, Th.—Heinrich, R.: The DLT TAU-Code: Recent Applications In Research And Industry. European Conference on Computational Fluid Dynamics (ECCOMAS CFD 2006), 25 pp.

[14] Yang, X.-J.—Liao, X.-K.—Lu K.—Hu, Q.-F.—Song, J.-Q.—Su, J.-S.: The TianHe-1A Supercomputer: Its Hardware and Software. Journal of Computer Science and Technology, Vol. 26, 2011, No. 3, pp. 344–351.

[15] `http://www.top500.org/system/10587/`.

[16] Gropp, W. D.—Kaushik, D. K.—Keyes, D. E.—Smith, B. F.: Towards Realistic Performance Bounds for Implicit CFD Codes. Proceedings of Parallel CFD '99, pp. 233–240.

[17] `http://www.nscc-tj.gov.cn/en/resources/resources_1.asp#TH-1A`.

[18] `http://www.intel.com/software/products/vtune/`.

[19] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes. September 2013.

Yonggang Che received his Ph. D. in computer science from National University of Defense Technology (NUDT), Changsha, P. R. China in 2004. He has been an Associate Professor at School of Computer, NUDT since 2006. His research interests include program performance optimization and performance evaluation of computer systems.

Lilun Zhang received his Ph. D. in computer science from National University of Defense Technology, Changsha, P. R. China in 2002. He has been an Associate Professor at School of Computer, NUDT since 2004. His research interests include parallel algorithms and optimization.

Chuanfu Xu received his Ph. D. in computer science from National University of Defense Technology, Changsha, P. R. China in 2011. He has been an Assistant Professor at School of Computer, NUDT since 2006. His research interests include program performance optimization and performance evaluation of computer systems.

Yongxian Wang received his Ph. D. in computer science from National University of Defense Technology, Changsha, P. R. China in 2004. He has been an Associate Professor at School of Computer, NUDT since 2008. His research interests include parallel algorithms and optimization.

Wei Liu received his Ph. D. in aerodynamics from National University of Defense Technology, Changsha, P. R. China in 2010. He has been an Associate Professor at School of Computer, NUDT since 2010. His research interests include parallel CFD computing.

Zhenghua Wang received his Ph. D. in aerodynamics from National University of Defense Technology, Changsha, P. R. China in 1992. He has been a Professor at School of Computer, NUDT since 1999. His research interests include parallel algorithms and optimization.