Benchmark of Parallelization Methods for Unstructured Shock Capturing Code

Tsutomu Saito, Atsushi Abe and Kazuyoshi Takayama
Shock Wave Research Center, Institute of Fluid Science,
Tohoku University, 2-1-1 Katahira, Aoba-ku, Sendai, 980-8577 Japan
saito@bellanca.ifs.tohoku.ac.jp

Abstract

This paper presents benchmark results of three different parallel-programming paradigms on an unstructured shock capturing numerical code for transient problems. The three parallel programming methods include: (1) a shared-memory programming of OpenMP using cache coherent non-uniform memory access (CC-NUMA) of SGI Origin2000, (2) an MPI (Message Passing Interface) implementation and (3) a SHMEM implementation using the parallel library called "Shared Memory Access Library". The methods (2) and (3) are both based on distributed memory architecture. SGI Origin2000 is used throughout the current study. It is found that the scalability of the programming (1) is so poor that its usage for the unstructured CFD code is impractical. The scalabilities of programming (2) and (3) are much better than programming (1) and the computational speed of giga-flops range can be achieved with 16 CPUs. The parallel programming with SHMEM libraries is approximately twice as fast as the one with MPI.

1. Introduction

Numerical simulations using unstructured numerical grids became more common in the computational fluid dynamics (CFD) [5] [13]. This numerical method has advantages when it is combined with local solution-adaptive technique over the conventional methods developed on structured numerical grids. Although the unstructured code needs complex algorithm for automatic mesh refinement and coarsening, it provides much higher resolutions compared with structured codes for the same number of grid points. In other words, it requires orders of magnitude less grid points for the same space resolution compared with structured codes. A two-dimensional (2D) numerical code for both the viscous and inviscid flows using quadrilateral unstructured grids is developed. Comparisons of the numerical results with experimental data of flow visualizations provide information not only on the code accuracy but also on detailed wave interactions which take place during the process. In performing CFD, the code performance on a specific computer on which the code runs is also important as well as the best choice of the numerical scheme. Three-dimensional (3D) calculations or two-dimensional calculations with high degree of mesh refinement need large amount of computer resources. RISC-based parallel computers are now widely used but each processor of them is relatively slow. It is, therefore, essential to have a reasonable scalability in the parallel executions of a code. It is known, however, that the scalability depends very much on the structure of a numerical code.

There are several methods of performing parallel computations. In this paper we investigate the performance of three different parallel programming methods which are available to us on SGI Origin2000 at the Institute of Fluid Science, Tohoku University. The benchmark is carried out on the two-dimensional unstructured code with adaptive mesh refinement. Benchmark results on parallel computations of unstructured numerical code is reported by Oliker and Biswas [6]. They compared several critical factors in parallel computations such as runtime, scalability, programmability, memory overhead etc. They also compared different hardwares including SGI Origin2000, Cray T3E and Cray MTA. Since we have an access only to Origin2000, our investigation is focused on the scalability of parallelized codes with three different parallel programmings. The three parallel programming methods are: (1) a shared-memory programming of OpenMP using cache coherent non-uniform memory access (CC-NUMA) of the Origin2000, (2) a distributed-memory programming of MPI (Message Passing Interface) and (3) a SHMEM ("Shared Memory Access Library") implementation that is also based on distributed memory.

In what follows, some details of the numerical code and numerical results for a sample problem are described first (Section 2). The benchmark procedure and the code performance on SGI Origin2000 is compared and discussed (Section 3). Conclusions, then, follows at the end.

2. Numerical code

The numerical simulation code using unstructured meshes with automatic mesh adaptation is one of the most powerful tools for CFD. It is reported, however, that this type of code has poor performance on cache based large-scale multiprocessor architectures such as Origin2000, T3E etc. Oleiker and Biswas investigated the performance of the grid-handling part of the parallel codes for the unstructured adaptive scheme applied to transient problems [6]. They compared three different parallel paradigms on three different leading hardwares. For unsteady problems, grid refinement and coarsening are necessary much more frequently compared with steady cases. For instance, in our case, the mesh adaptation is done at every time step and about 40% of the total CPU time is spent in the process of grid-handling. Therefore the degree of significance of the grid-handling is high and its performance is important. It is, however, still more important to have efficient flow solver on parallel computers since it takes the major part of computational time. This paper, in contrast to Oleiker's work [6], compares the performance of the flow-solver of our unstructured code with different parallel programming methods. In this section the solver is described in some details.

2.1. Governing equations

Although the numerical code used in the current study is capable of simulating the 2D Navier-Stokes Equations, the effects of viscosity and heat-conduction are intentionally left out for clarifying the performance of the essential part of the flow solver. Namely the 2D governing equations are reduced to Euler equations as:

$$U_t + F_x + G_y = 0, \tag{1}$$

where t, x and y are the time and space coordinates [2]. The equations represent the conservation laws of mass, momentum and energy, and the conserved quantities expressed in the vector form, U, is of the following form:

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix}, \tag{2}$$

and the convection terms, F and G, are expressed as:

$$F = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix}, \quad G = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{pmatrix}, \tag{3}$$

where ρ, p, u, v are the density, pressure, flow velocity elements in x and y directions, respectively. The total energy per unit volume, E, is expressed as $E = \rho\epsilon + \rho(u^2+v^2)/2$ and an ideal-gas equation of state, $\epsilon = p/(\gamma+1)\rho$, is used to close the whole system of equations. Here γ is the ratio of specific heats of the gas. Here equations are written in Cartesian coordinates for simplicity but the actual code is developed in general curvilinear coordinates.

2.2. Numerical scheme

The numerical code for solving the basic conservation equations employs the finite volume method. Although triangular grid cells are more common, the code employs quadrilateral cells due to reasons: (1) it is better for representing boundary layers near solid walls in case of viscous flow analysis; (2) it is easier to vectorize and/or parallelize if cells are quadrilateral or the edge-number of the cell is even [7]. The property (2) is an advantage when it is run on vector and/or parallel computers.

The gasdynamic and thermodynamic flow quantities are updated for new time levels by summing up numerical fluxes corresponding to physical fluxes at cell interfaces. The numerical flux at cell interface is obtained by the Weighted Average Flux method (WAF) of Toro with TVD stability conditions [10] [11] [12]. The scheme is one of the most commonly used shock capturing schemes today. It is an extension of Godunov method to higher-order accuracy and has the second-order accuracy in both space and time when applied to smooth flows on Cartesian grids.

It used to be a difficult task to generate unstructured numerical grids but now many useful commercial mesh generators are available. The commercial softwares GAMBIT (FLUENT Inc.) or ICEM-CFD is used in the current study. The numerical meshes are generated interactively with a convenient graphical user interface.

2.3. Flow visualization and numerical simulation

The benchmark problem that is used in the current study is the transient flow in a 2D nozzle. The starting process of nozzles is quite challenging for numerical simulations since it involves highly nonstationary wave interactions among shock waves, rarefaction waves, contact surface, vortices and boundary layers etc. as shown in Fig.1 [1] [3] [8] [9]. The figure shows the flow inside a 2D nozzle taken at $155\mu s$ after the moment when the incident shock wave has passed through the nozzle throat. The incident shock wave Mach number is 2.5 and the test gas is air. The transmitted shocke wave, upstream-running secondary generated shock wave, separation bubbles and a pair of vortices are clearly seen in the figure. One of the unstructured numerical grids produced by GAMBIT is shown in Fig.2. For investigations of nozzle starting process, the numerical grids are automatically refined or coarsened based on the implemented adap-

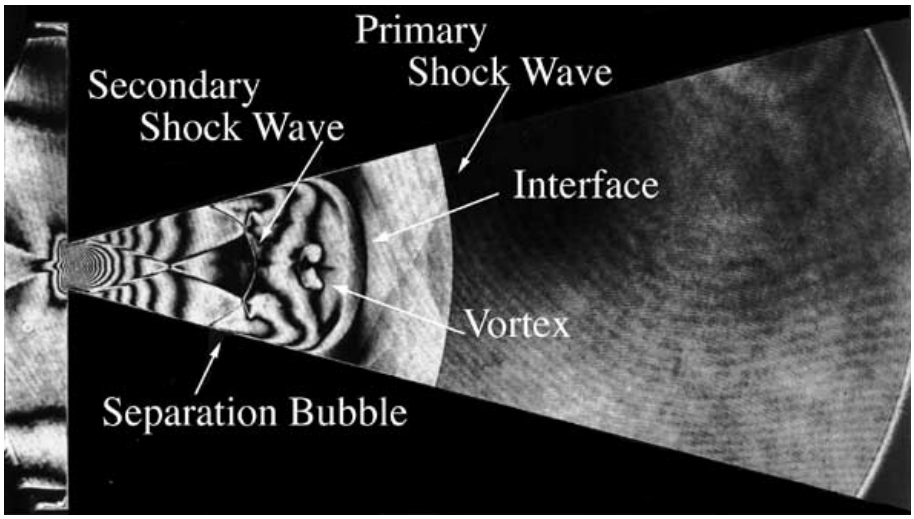

Figure 1. Experimental flow visualization of 2D nozzle flow

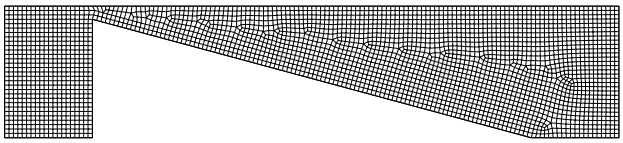

Figure 2. Numerical grid generated by Gambit

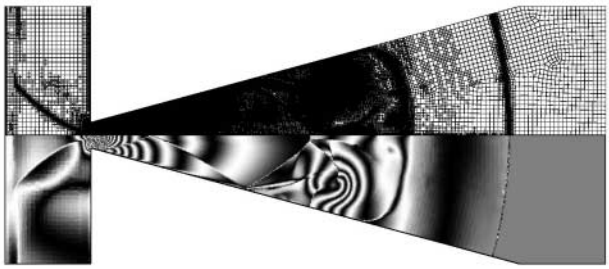

Figure 3. Numerical result and automatically refined numerical grid

tation algolism in order to obtain high spatial resolutions. The numerical result for the incident shock Mach number of 2.5 in the air together with the automatically refined numerical grids are shown in Fig.3. It shows all important flow features that compare with experimental data. Detailed descriptions about flow features are found in references [8] and [9].

3. Code performance on parallel computer

As already mentioned before, this paper is focused on the performance of the flow-solver of our unsteady unstructured CFD code. Therefore, the automatic mesh adaptation is turned off during the benchmark. We investigate three different parallel programmings as mentioned before, and each method is described in this section.

3.1. Performance on a single CPU

Origin2000 has theoretical maximum CPU speed of 600 Mflops per processor. However it is known that, for most CFD applications, the typical sustained CPU speed is 10 to 15 % of the maximum speed. The values of measured code speed on a single CPU are listed in Table 1.

In the table, the original code is the one which is developed on a vector-parallel computer (Cray C90). The CPU speed of the original code was about 65 Mflops. After optimizing the code by promoting the efficiency of the cache usage, the speed is increased to about 105 Mflops. The main modification was to change the structure of the variable arrays so that as much data as possible are used from data caches without rewriting them.

Virtual memory computers such as Origin2000, need the conversion between the real memory address and the virtual memory address. The conversion is done by referencing the conversion table called the translation look-aside buffer (TLB) on a fast but small register of the CPU board. Since TLB can keep only a part of whole address conversion table, if the part which is necessary for the code execution is not on it, the appropriate part of the table must be fetched from the memory. The time wasted by this fetch process is also

Table 1. Code performance on single CPU

		Original code Total CPU: 38.1sec	Modified code Total CPU: 154.sec
CPU Speed	$(Mflops)$	64.5	105.
TLB Miss	(sec)	15.7	0.408
L1 Cache Hit Rate	$(\%)$	85.7	93.5
L2 Cache Hit Rate	$(\%)$	98.6	96.6

listed in the table as TLB miss. It is noted that as much as 16 seconds is wasted out of 38 second total CPU time in the original code. While with the modified code, TLB miss is only 0.4 seconds for the total execution time of 154 seconds.

These values are measured by the software tool Perfex provided by SGI. Although Perfex gathers timing data by sampling and gives only an approximate results, we did not take any specific precautions or averaging to improve the accuracy of the results. The values in the table are copied directly from the output of the tool. Considering the complex code structure with conditional jumps and the usage of list vectors, further improvement of the code performance was difficult.

3.2. Shared memory parallel programming

The performance of the code is first measured for the shared-memory parallel programming of OpenMP. In this method, the parallel execution is mostly at the loop level and the conversion from the original code to the multi-cpu program is mainly done by inserting appropriate directives at the beginning of the Do loops. The conversion is straightforward and similar to the vectorization process.

As already mentioned before, since the speed of each processor, especially the sustained speed is relatively slow, it is essential to have high scalability with respect to the number of processors used. The speedups for different numbers of CPUs are listed in Table 2. It is found that the speedup of the original code is less than unity, meaning that the code runs slower if multiple CPUs are used. The speedup of the optimized code is increased to more than unity as listed in Table 2 but it is still much lower than the corresponding number of CPUs. Due to decrease in each processor's speed with the increase in the number of processors, the total speed of the code has a maximum value at a relatively low Mflops rate ($\sim$ 300 Mflops).

This extreme degradation in speedups in parallel computation is known as a result of the false sharing of cache lines on multiple CPUs. This phenomena happens when the same cache lines reside on several different CPUs and some values on them are updated. Under the condition, unnecessary cache updates take place wasting a large amount of CPU time resulting in very poor parallel performance. The current code is found to have this unfortunate characteristics. In principle, it may be avoided but for practical application codes, it is not easy to remove the cache false-sharing without extensive code rewritings. One of the ways to do so, perhaps, is to use the domain decomposition with the message passings as it is widely used in steady state calculations.

3.3. Distributed memory parallel programing

In the previous section, it was shown that the numerical code has poor parallel performance with shared memory programming on the cache based computer, Origin2000. This is mainly due to the cache false-sharing that is inherent to the cache based memory architecture and is quite difficult to eliminate.

In order to achieve a reasonable scalability, the message passing is a natural choice since cache false sharing does not occur in these programming methods. In this study, we tried MPI and SHMEM libraries as message passing programmings.

3.4. Domain decomposition

In order to investigate the parallel performance of the flow solver, separated from the rest of the code, we prepared grid files for each CPU in advance. The original numerical grids generated with GAMBIT, such as Fig.2, is divided into a specific number of subdomains corresponding to the number of CPUs. The graph partitioning library, METIS, is used to divide the global numerical grids into subgrids. METIS is developed by Karypis et.al. [4] and is widely used for domain decomposition. The subdomains generated by METIS are shown in Fig.4. In the figure, the original grids are divided into 32 subdomains since 32 CPUs are planned to be used in this case. The data corresponding to different subdomains, such as the grid coordinates and all necessary information regarding the data exchange

Table 2. Parallel performance with OpenMP for different CPU numbers

Number of CPU	Wall clock time (sec)	Speedup	Mflops/CPU	Total Mflops*
1	146.	1.00	112.	112.
2	97.2	1.50	84.3	169.
4	63.5	2.29	64.3	257.
8	57.5	2.53	36.2	290.
16	52.6	2.77	19.6	314.
32	61.9	2.35	8.26	264.

*: Mflops/CPU × Number of CPU used

between subdomains next to each other are written into separated data files. These files serve as the input files of the parallel execution.

3.5. Parallel program with MPI

In the present numerical code, there are two places where data exchange is required. One is in the subroutine where the time step is determined. After determining the time steps from the CFL stability condition in all subdomains, the minimum of them is selected and broadcasted by a single call to MPI reduction subroutine, MPI_Allreduce().

Another place for data exchange is the inner boundary, the boundary between adjacent subdomains in the flow region. Data to be exchanged are the four flow parameters, i.e. the pressure, density and two components of flow velocity, and their jumps at the cell interfaces that are one cell layer inside the subdomains. The latter four are necessary to keep numerical stability in WAF scheme. Therefore, at a single cell boundary of subdomains, say domain A and domain B, eight data are transferred from domain A to domain B and also from domain B to domain A. Two MPI calls are required for transferring a group of data, one for sending and one for receiving. As a result, four MPI calls are necessary at each inner cell boundary. In the present numerical code, however, flow parameters and the values of their jumps across the cell boundary are transferred separately. Therefore, the number of MPI call is doubled at each inner boundary cell interface. It is possible to modify the code in such a way that the flow parameters and their jumps occupy contiguous memory area, so that we can send or receive all eight data as one group. This modification, however, is not tried in this study. As far as the coding is concerned, the number of additional statements for the data communication is about sixty.

It may give an impression that the additional statements, sixty lines, for data exchange is insignificant. However a great deal of detailed considerations, such as rearrangements of arrays due to introductions of local addressing, took much time. In a sense, it is almost a complete rewrite of the code.

3.6. Parallel program with SHMEM libraries

Parallel programming with SHMEM libraries is similar to MPI. The main difference between the two lies in the process how the data is transferred to the target memory. The libraries directly write data into the target memory of other CPUs without going through the communication buffer as in MPI. Therefore, in order to transfer data, only one subroutine call to the library is needed. This makes the coding simpler compared with MPI programming. The libraries, however, write data in the target memory directly when it is called and the programmer is responsible for the synchronization. Code synchronization is explicitly done calling barrier library routines whenever it is needed. For determining the time increments, we used two library calls, one for the barrier routine, shmem_barrier(), and one for the reduction routine, shmem_min_to_all().

The number of additional statements for the data communication is only 17 with the libraries including calls to barrier routines. This is less than a half of what is needed for MPI programming. In this study, the code conversion was carried out from MPI code that is described in the previous section by replacing the MPI call to a corresponding SHMEM library call. So it was much easier than before, i.e. from the original code to MPI code.

3.7. Benchmark results and discussions

Benchmark results of MPI parallel calculations are listed in Table 3. The total number of cell is 85959 and the number of time step is 1000. It is noted that the value for the speedup is still increasing at 64 CPUs in contrast to the case of OpenMP, Table 2, where the speedup is saturated at a relatively low level of 2.8, limiting the total speed of only about 300 Mflops. It should be mentioned here that the program is simply rewritten using MPI and no specific

Figure 4. Numerical grid subdomains generated by METIS

optimizations for either single CPU performance or parallel performance have been done. Once program size and the number of CPUs are fixed, specific code optimizations based on the program size on each CPU can be done. It is expected, from our experience, that the performance can be doubled or at least 50% up with a reasonable amount of effort.

Benchmark results for MPI parallel calculations with a much less cell number are listed in Table 4. The total number of cell is 21450 and the number of time step is 3000. The performance either of a single CPU or of the linearity for the parallel executions depends on the amount of work since the ratio of scalable to non-scalable part of a job including data communications will change. Again it is noted that the value of the speedup still keeps increasing at 64-CPU case and the total performance exceeds one gigaflops with 16 CPUs. Comparing Tables 3 and 4, we notice: (1) the linearity is better for larger calculations; (2) the performance (speed) per processor is better for smaller calculations. The first characteristic is explained by the fact that the time spent for data communication has relatively larger influence for smaller calculations. The relative time spent for data communication compared with the other parallel execution of flow calculation is larger for smaller cell numbers. The second characteristic can be explained by the efficiency of cache usage. For unstructured code, it is expected that the cache hit rate be higher for smaller arrays.

Benchmark results with SHMEM libraries are compared with those of OpenMP and MPI in Table 5. It is seen that the parallel programming with SHMEM libraries shows even better scalability than MPI code. It shows speedups more than the number of CPUs used (super-linear scalability) up to 16-CPU case in this specific case. As a result, the speed degradation par CPU is also smaller and the total speed of more than four gigaflops is acheved with 64 CPUs. It is expected that the difference between MPI and SHMEM codes becomes smaller for cases with larger cell numbers since the data communications become less significant for larger numerical cell numbers. The cell numbers used in the present investigations are typical to our 2D simulations. Therefore, unless it is necessary to consider the portability, it is found that the parallel programming with SHMEM libraries is the best choice for the current unstructured CFD code on Origin2000.

4. Conclusions

A locally adaptive numerical code for solving unsteady compressible flows is developed and applied to a 2D nozzle starting problem. The code performance of the parallel programmings with three different methods are investigated. The code has reasonable performance on a single processor. However, the parallel programming with OpenMP based on the shared memory architecture has very poor scalability due to cache-line false sharing. As a result, the performance of the code is limited to a maximum of relatively low value ($\sim$ 300 Mflops). The parallel programmings based on distributed-memory concept using MPI or SHMEM libraries have much better parallel performances. It is expected that the performance will be further improved provided that the problem size and the number of CPU are fixed. The code with SHMEM libraries has good scalability and the total computational speed.

Table 3. Parallel performance with MPI (85959 cells)

Number of CPU	Wall clock time (sec)	Speedup	Mflops/CPU	Total Mflops*
1	1393.	1.00	44.3	44.3
2	664.	2.10	46.9	93.8
4	448.	3.11	34.6	138.
8	203.	6.86	39.2	314.
16	109.	12.8	37.1	594.
32	68.0	20.5	33.0	1060.
64	54.0	25.8	22.7	1450.

∗: Mflops/CPU × Number of CPU used

Table 4. Parallel performance with MPI (21450 cells)

Number of CPU	Wall clock time (sec)	Speedup	Mflops/CPU	Total Mflops*
1	1097.	1.00	89.8	89.8
2	719.	1.53	68.9	138.
4	300.	3.66	82.7	331.
8	164.	6.69	77.0	616.
16	93.0	11.8	69.2	1110.
32	63.0	17.4	52.7	1690.
64	59.0	18.6	32.2	2060.

∗: Mflops/CPU × Number of CPU used

Table 5. Comparison of different parallel programming methods

CPU Number	OpenMP		MPI		SHMEM		
	time	speedup	time	speedup	time	speedup	Total Mflops
1	1210.	1.00	1097.	1.00	1200.	1.00	77.4
2	810.	1.51	719.	1.53	598.	2.00	165.
4	516.	2.35	300.	3.66	257.	4.67	400.
8	435.	2.79	164.	6.69	140.	8.58	711.
16	361.	3.36	93.0	11.8	74.0	16.2	1340.
32	364.	3.33	63.0	17.4	40.0	30.0	2560.
64	–	–	59.0	18.6	29.0	41.4	4120.

References

[1] Amann HO (1968) Experimental study of the starting process in a reflection nozzle. The Physics of Fluids Supplement I:I-155–I-153.

[2] Anderson DA, Tannehill JC, Pletcher RH (1984) Computational fluid mechanics and heat transfer. Hemisphere publishing corporation, New York, NY USA.

[3] Igra O, Wang L, Falcovitz J, Amann O (1998) Simulation of the starting flow in a wedge-like nozzle. Shock Waves 8:253-242

[4] Karypis G, Kumar V (1998) MeTiS 4.0: Unstruc-

tured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota.

[5] Löhner R (1987) An adaptive finite element scheme for transient problems in CFD, *Comput. Meths. Appl. Meh. Engrg.* V61, pp.323-338.

[6] Oliker L, Biswas R (1999) Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms. Supercomputing'99, Seattle, Washington; also: IEEE Transactions on Parallel and distributed System, to appear.

[7] Sun M (1998) Numerical and experimental studies of shock wave interaction with bodies, Ph.D. Thesis, Tohoku University, Japan.

[8] Saito T, Takayama, K (1999) Numerical simulations of nozzle starting process, Shock Waves, 9:73–79.

[9] Saito T, Timofeev EV, Sun M, Takayama K (1999) Numerical and Experimental study of 2-D nozzle starting process, Proceedings of the 22nd International Symposium on Shock Waves, Vol.2, 1071–1076.

[10] Toro EF (1999) Riemann solvers and numerical methods for fluid dynamics, 2nd edition, Springer.

[11] Toro EF (1992) Riemann Problems and the WAF Method for Solving Two-Dimenshional Shallow Water Equations. *Phil. Trans. Roy. Soc. London*, A338:43–68.

[12] Toro EF (1992) The Weighted Average Flux Method Applied to the Time-Dependent Euler Equations. *Phil. Trans. Roy. Soc. London*, A341:499–530.

[13] Voinovich P, Timofeev E, Takayama K, Saito T, Galyukov A (1998) 3-D unstructured adaptive supercomputing for transient problems of volcanic blast waves.
AIAA paper 98-0540.