Poor Scalability of Parallel Shared Memory Model: Myth or Reality?

Mark Kremenetsky, Arthur Raefsky, and Steve Reinhardt

Supercomputer Applications, Silicon Graphics Inc.,
Mountain View, California 94043
{mdk,raefsky,spr}@sgi.com

Abstract. Large CFD models require memory sizes larger than can be supported by today's single 'node' computers. Using the memory of more than one node can greatly complicate the creation of a well-performing program. We believe that preserving globally addressable memory beyond the boundary of a single node enables diversity of programming methods and provides flexibility essential for optimum algorithm development. We isolate the effects of algorithm and implementation by porting the same parallel CFD algorithm in three styles: fine-grain shared memory, coarse-grain shared memory, and coarse-grain distributed memory.
Despite some fundamental differences in programming implementation, both distributed memory and coarse-grain shared memory code provide very close parallel performance due to algorithmic similarities. At the same time fine-grain shared memory code, despite use of the same programming paradigm as coarse-grain shared memory program, falls far behind due to unavoidable parallel performance penalties caused by Amdahl's law and some other limitations.

1 Introduction

It has long been recognized that the two necessary elements for achieving scalability in application performance are scalable hardware and software. Both these elements have existed for some time. Scalable hardware in the context of physically distributed memories connected through a scalable interconnection network has been commercially available since the 1980's. Such systems provide only an interconnection network and the burden of scalability then falls on software. As a result, scalable software for such systems will exist only in a message passing model.
There is another class of parallel architecture with scalable hardware support for cache coherence. These are generally referred to as Shared Memory Multiprocessor or SMP architectures. For SMP systems, the native programming model is shared memory and message passing is built on a top of shared memory model. This software environment provides multiple parallel programming methods and parallel paradigms.

This paper will demonstrate that the so-called "poor scalability" of shared memory programming paradigm is "myth" rather than "reality" and that the impediments to

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 657–666, 2003.

software scalability are not the programming model but rather parallel application methods.

Within this context we implement a single key CFD algorithm in multiple styles (methods) to illustrate the benefits of a low-latency, high-bandwidth memory system available via multiple programming methods. The first style is classic OpenMP loop-level parallelism, the second is OpenMP directives used in a coarse-grain fashion with ghost cells, and the third is a distributed memory version implemented with MPI. We conclude that coarse-grain parallelism is required for strong scalability, and its implementation via a shared memory approach or distributed memory paradigm (MPI) provides similar parallel performance.

2 Parallel Programming Models (Paradigms)

This paper concerns itself with two different programming models, message passing and shared memory. The message passing model assumes the underlying parallel architecture is simply a collection of computers, or nodes, linked together by network. Each individual node knows nothing about the memory of other nodes in the system. The programmer must explicitly pass information between nodes by sending messages. In addition, a message cannot be received by a node unless that node explicitly posts a "receive". This last restriction is relaxed for so-called "one-sided" message passing as exemplified by Cray's shmem model or MPI-2 standard. One–sided message passing requires hardware support for good performance. In common two-sided message passing, both sending and receiving node must coordinate to successfully share data. Although each processor can execute a different program, most often the "Single Program, Multiple Data" (SPMD) programming style is used; all processors execute the same program, acting on different parts of data set. This requires an appropriate partitioning of the data and operations. The partitioning of the data must be done in such a manner that the workload is well balanced between processors, and communications and synchronizations are minimized.

In contrast, in the shared memory model every processor has direct access to the memory of every other processor in the system. This means a processor can directly load or store any shared address. The programmer can also declare certain pieces of memory to be private to the processors. Parallelism in a shared memory model exists in a form of multiple threads of execution. These threads typically fork from a master thread, and throughout the course of execution may join and refork a number of times. The fork/join execution model makes it easy to get loop level parallelism out of a sequential program. Unlike the message passing model, where the program must be completely decomposed for parallel execution, in a shared memory model it is possible to parallelize just at the loop level without explicitly decomposing the data structures.

Unfortunately loop level parallelism is rarely scalable because it typically leaves some constant fraction of sequential work in the program that, by Amdahl's law, can quickly decrease the gains from parallel execution. Loop level parallelism can easily lead to race conditions and frequent synchronization. False sharing when two or more processors want to write to the same cache line can be another negative factor. All these deficiencies created a "myth" about poor scalability of the shared memory programming model.

It is important, however, to distinguish between the style of parallelism (e.g. loop level vs. coarse grained) and the programming model. Programming in the distributed memory model can be done only with coarse grain parallelism. It is often much more difficult than programming with loop level parallelism in the shared memory model, but it provides a much higher level of scalability. The programmer must be aware of the location of the data in the local memories and has to move or distribute these data explicitly when needed. The partitioning of the data and all necessary communication has to be included explicitly in the program. The sequential program often needs significant changes in order to parallelize it.

But there is nothing in the shared memory parallel model that prevents a programmer from achieving coarse grained parallelism. In the remainder of this paper we will show that the level of parallelism exposed in a program is dependent on the algorithm and data structures employed and not on the programming model.
Therefore, given a parallel algorithm and SMP architecture, a shared memory implementation should scale as well as a message passing implementation.

3 CFD Solver Algorithm

The most widely accepted methods for studying the transient behavior of the compressible Euler and Navier-Stokes equations are implicit. Unfortunately when solving realistic multi-dimensional problems, the system of equations that need to be solved are often too large to apply direct methods to. Most codes resort to solving a modified system of equations and employ techniques such as the implicit approximate factorization [1,2] and the diagonal approximation [3,4], which reduce a block 3D system to a serial sequence of one-dimensional blocked or scalar systems. These techniques work adequately on structured orthogonal grids, but also have uncontrolled factorization errors. Besides, these techniques do not apply for problems defined on unstructured meshes.

Here we employ the Bi-CGSTAB [5] algorithm and use it to solve a linear system of original discretized equations without resorting to approximate factorization techniques.

3.1 Notation

Consider the dimensionless Navier – Stokes equations in the strong conservative form:

$$U_{,t} + F_{i,i} = \mathrm{Re}^{-1} F^{v}_{i,i} \tag{1}$$

where:

$$\mathrm{U} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{bmatrix} = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho e \end{bmatrix} \tag{2}$$

The Euler flux F_i is given by

$$F_i = u_i U + p \begin{bmatrix} 0 \\ \delta_{i1} \\ \delta_{i2} \\ \delta_{i3} \\ u_i \end{bmatrix} \tag{3}$$

and the viscous flux F_i^v is given by

$$F_i^v = \begin{bmatrix} 0 \\ \tau_{i1} \\ \tau_{i2} \\ \tau_{i3} \\ \tau_{ij} u_i - \dfrac{\mu}{P_r(\gamma-1)}(a^2)_{,i} \end{bmatrix} \tag{4}$$

where the viscous tensor τ_{ij} is given by

$$\tau_{ij} = \lambda\, u_{k,k} \delta_{ij} + \mu\, (u_{i,j} + u_{j,i}) \tag{5}$$

where λ and μ are the viscosity coefficients.

3.2 Discretization

To define the discrete form of the full Navier-Stokes equations we use a general three-point implicit time-stepping method [3]

$$\Delta U^n = \frac{\theta \Delta t}{1+\phi}\frac{\partial}{\partial t}\Delta U^n + \frac{\Delta t}{1+\phi}\frac{\partial}{\partial t}U^n + \frac{\phi}{1+\phi}\Delta U^{n-1} + O[(\theta - \frac{1}{2} - \phi)\Delta t^2 + \Delta t^3] \tag{6}$$

where $\Delta U^n = U^{n+1} - U^n$ and $U^n = U\ (n\Delta t)$. The parameters θ and ϕ can be chosen to produce different schemes of either first or second order accuracy in time. We restrict ourselves to the first order in time scheme with $\theta = 1$ and $\phi = 0$ (although what follows can easily be extended to second order accuracy). Applying eq.(6) to eq.(1) with time step h = Δt results in:

$$U^{n+1} - U^n + h\, F_{i,i}^{n+1} = h\, R_e^{-1} (F_{i,i}^v)^{n+1}, \tag{7}$$

In order to evaluate (7) we must know the advective and diffusive flux vectors F_i and F_i^v at time t = (n+1) Δt. We see that Eq.(7) is nonlinear in U^{n+}. The nonlinear terms are linearized in Taylor series expansion about $\Delta U^n = 0$,

$$F_i^{n+1} = F_i^n + A_i^n \Delta U^n + O(h^2), \tag{8}$$

and

$$F_i^{v\,n+1} = F_i^{v\,n} + M_i^n \Delta U^n + O(h^2), \tag{9}$$

where the Euler Jacobian is $A_i = \partial F_i / \partial U$ and the diffusive Jacobian is $M_i = \partial F_i^v / \partial U$. The analytical form of A_i and M_i can be found in [2].
Using (8) and (9) we rewrite (7) as:

$$[I + h A_{i,i}^n - h M_{i,i}^n] \Delta U^n = h R_e^{-1} F_{i,i}^{v\,n} - h F_{i,i}^n, \tag{10}$$

For the space discretization of flux and Jacobian derivatives we use central differences. Finally, to handle strong shock waves we added a second and fourth-order explicit artificial dissipation terms to the right hand side of Eq.(10) and a second-order implicit diffusive term to the left hand side.

Such discretization leads to a large, sparse nonsymmetric system of linear equations that must be solved at every time step. Thus the unfactored set of linear equations that we solve using a parallel version of Bi-CGSTAB algorithm. With $A = [I + hA_{i,i}^n - hM_{i,i}^n]$ and $x = \Delta U^n$, and $b = hR_e^{-1} F_{i,i}^{v\,n} - F_{i,i}^n$ we rewrite (9) in the familiar linear equation form $Ax = b$.

3.3 Iterative Algorithm

Several iterative algorithms [6] have been proposed for the solution of nonsymmetric systems of linear equations. Here we employ an algorithm due to Van der Vorst [5] which is fast and smoothly converging. The Bi-CGSTAB algorithm is described below.

Bi-CGSTAB:

x_0 is an initial guess; $r_0 = b - Ax_0$;
y_0 is an arbitrary vector, such that
$(y_0, r_0) \neq 0$, e.g. $y_0 = r_0$;
$\rho_0 = \alpha = \omega_0 = 1$;

$v_0 = \rho_0 = 0$;
for i = 1,2,3…
$\rho_i = (y_0, r_{i-1})$; $\beta = (\rho_i / \rho_{i-1})(\alpha / \omega_{i-1})$;
$p_i = r_{i-1} + \beta (p_{i-1} - \omega_{i-1} v_{i-1})$;
$v_i = A p_i$;
$\alpha = \rho_i / (y_0, v_i)$;
$s = r_{i-1} - \alpha v_i$;
$t = A s$;
$\omega_i = (t, s) / (t, t)$;
$x_i = x_{i-1} + \alpha p_i + \omega_i s$;

if x_i is accurate enough then quit;
$r_i = s - \omega_i t$;
end

We note that the iteration loop of Bi-CGSTAB involves two matrix vector multiplications (Ap_i and As). There are also four inner products as well as vector updates which lead to an additional 12 N flops.

4 Implementation

We have implemented the algorithm described in the previous section entirely in Fortran90. Using orthogonal coordinates in three spatial dimensions, the structure of the matrix A is block-sparse. The global spatial structure of A is similar to any second order operator (e.g., Laplacian operator) but additionally A has a local dense 5 by 5 block structure. After computing the Jacobians A_i and M_i which involve mostly local computation with a bit of differencing, the heart of the algorithm involves fetching 5 element vectors from neighboring grid points followed by a local 5 by 5 matrix multiplying a 5 element vector at every grid point.

As is the case in most iterative algorithms, the computational time is dominated by the matrix vector multiplication. Our linear solver has no knowledge of the structure of the matrix and one need only modify the matrix_times_vector( ) subroutine to support other spatial structures (e.g. high order and upwind schemes).

4.1 Parallel Methods

The algorithm described in the previous sections has been parallelized as follows:

Fine-Grain Method. This mode implements loop level parallelism within a shared memory concept. Domain decomposition was applied through multithreading the

outer loops over the last spatial dimension using OpenMP parallel directives. All data structures are shared. Such an approach can be described as one-dimensional domain decomposition with edge dissection (non-overlapping domains, Fig.1a)

Coarse-Grain Method. This mode implements a shared memory paradigm with the use of OpenMP directives. In contrast to the Fine-Grain Mode, this approach is based on the Schwarz additive domain decomposition approach with overlapping local regions (Fig.1b). Major data structures are declared "private" which means that they are not explicitly shared between computational threads. There are only two parallel directives in this mode: the first opens the parallel region in the very beginning of the code and the second one closes the parallel region at the end of this code. In order to use a shared memory paradigm in combination with local data structures, we created a set of shared pointers to those local structures that allows any parallel thread to address private data from other domains. This unusual approach allowed us to implement the coarsest possible level of decomposition within a shared memory paradigm. As a result of this implementation we were able to avoid typical hazards of fine-grain (loop-level) parallelism such as frequent synchronization, false sharing of cache lines and almost completely eliminate a sequential component of parallel code.

Distributed Mode. This mode is also based on Schwarz additive domain decomposition but implements it within a distributed memory paradigm with the use of the MPI (Message Passing Interface) protocol. Algorithmically, Coarse-Grain and Distributed Methods are identical, but the implementation is very different.

The code was compiled with the Intel Linux compiler for the IPF/Linux platform. The OpenMP library was supplied by Intel and the MPI library is the SGI implementation of the MPI-1 protocol.

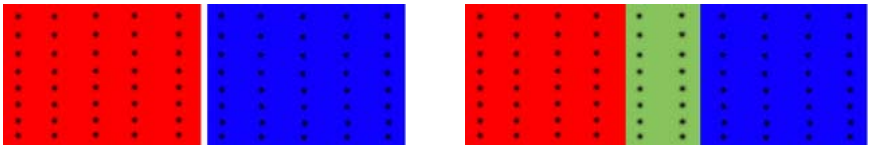

Fig. 1 (a). Partition of a domain in two subdomains

Edge dissection – non-overlapping regions
Vortex dissection – overlapping regions with one "ghost" cells layer

4.2 Flow Problem

We computed the three-dimensional transonic transient laminar flow, which includes such interesting gas dynamic phenomena as shock and rare fraction waves. We believe that such complex problems can be considered a reasonable test.

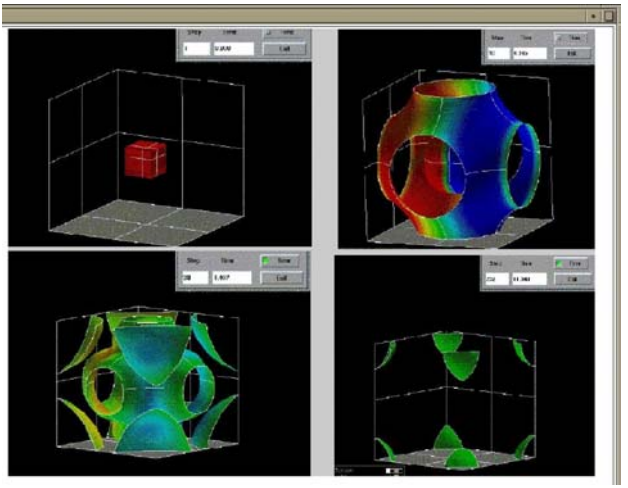

Fig. 2. Blast Waves transient flow (pressure iso - surface colored by velocity field)

The initial configuration consists of a high pressure and density region at the center of the cubic cell with the low pressure gas at rest. Periodic boundary conditions are applied to the array of cubic cells forming an infinite network. At the initial moment the high pressure volume begins to expand in the radial direction as the shock waves flows. At the same time the rare waves move to fill the void at the center of the cubic cell. When the expanding flow reaches the boundaries it collides with its periodic images creating a complicated structure of interfering shock waves (Fig.2). In the case of viscous gas, these processes create a nonlinear damped periodic system with energy being dissipated in time. Finally, the system will come to an equilibrated steady state.

5 Performance Results

The layout of our benchmark suite looked like this:

- We tested 4 sets of input data with variable size in order to evaluate the parallel scaling properties as a function of problem size. The first two dimensions of the computational grid were kept constant but the last dimension along which the parallel decomposition was performed was increased in powers of 2. All physical and computational parameters other than grid size, were fixed for all test cases.
- At first we tested our models on a well known multiprocessor ccNUMA system, the SGI Origin3000 with 128 MIPS R14000 processors running at 600 Mhz clock rate
- The next set of tests was performed on a new SGI system – Altix3300 with 64 Itanium2 processors running at 900Mhz clock rate. This system belongs to the same class of ccNUMA computers as the Origin3000, allowing global addressing from every CPU. The internal network of this system is based on NumaFlex link fabrics from SGI and provides extremely low latency and high bandwidth

communication layer that allows parallel applications to scale to the high count of processors.

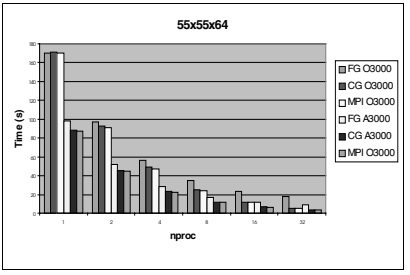

Fig. 3.

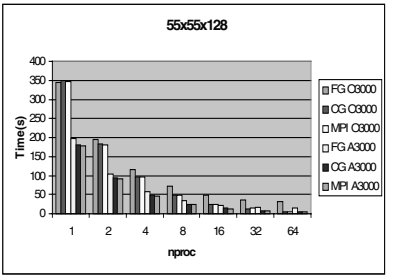

Fig. 4.

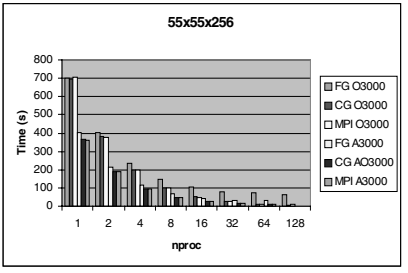

Fig. 5.

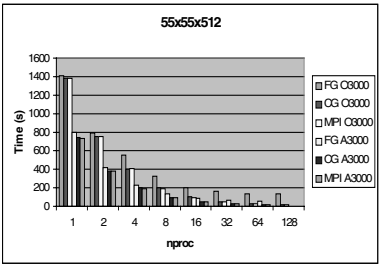

Fig. 6.

We observe the following about this algorithm:

- The easiest way to parallelize any technical application is through loop-level (fine-grain) parallelism within the shared memory paradigm. OpenMP directives provide all the necessary tools for such an approach. Unfortunately the parallel performance of such codes will always be limited by Amdahl's Law constraints.
- The coarse-grain parallel approach eliminates, or significantly reduces, the sequential part of a code, which in turn provides much better parallel scaling. Coarse-grain parallelism can be realized within either distributed or shared memory paradigms. Either memory model requires a complete redesign of the sequential code to expose the parallelism properly. This contrasts with the incremental approach possible with loop-level parallelism. Domain decomposition is one technique for exposing the parallelism.
- The coarse-grain shared memory and distributed memory code show approximately the same level of parallel scaling. Both codes have essentially identical algorithms, which proves that parallel performance of computer codes is not defined by the nature of parallel paradigm in use but rather is a function of a specific implementation algorithm.
- Large shared memory systems present a very capable platform for implementation and production use of both shared memory and distributed memory parallel programs.

6 Conclusions

We demonstrated that scalable programs can be easily written in a shared memory model under constraints of adequate parallel algorithm and data distribution. In order to prove this statement we implemented and tested various parallel modes of a complex CFD code and demonstrated good parallel scaling for a coarse grain parallel implementation in both shared and distributed memory paradigms.

References

1. Beam, R. and Warming, R.F., An Implicit Finite-Difference Algorithm for Hyperbolic Systems in Conservative Law Form, J. Comp. Phys. 22(1976), 87–110
2. Pulliam, T.H., Efficient Solution Methods for the Navier-Stokes Equations. Lecture Notes for the Von Karman Institute for Fluid Dynamics Lecture Series: Numerical Techniques for Viscous Flow Computation in Turbomachinery Blades. Jan 20–24, 1986, Brussels, Belgium, pp.1–102
3. Warming, R.F. and Beam, R.M., On the Construction and Application of Implicit factored Schemes for Conservation Laws in Computational Fluid Dynamics, SIAM-AMS Proceedings Vol.11 ed. Herbert Keller, American Mathematical Society, Providence, Rhode island, 1978
4. Jespersen, D. and Levitt, C., Numerical Simulation of Flow Past a Tapered Cylinder. NASA Ames Technical Report RNR 90-021, October 1990,1–10
5. Van der Vorst, H.A., Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM J., Sci. Stat. Comput., Vol.13(2), PP.631–644, 1992