Performance of PDE Solvers on a Self-Optimizing NUMA Architecture

Sverker Holmgren, Markus Nordén, Jarmo Rantakokko and Dan Wallin

Uppsala University, Information Technology
P. O. Box 120
SE-751 04 UPPSALA, Sweden
{Sverker.Holmgren, Markus.Norden, Jarmo.Rantakokko, Dan.Wallin}@it.uu.se

Abstract. The performance of shared-memory (OpenMP) implementations of three different PDE solver kernels representing finite difference methods, finite volume methods, and spectral methods has been investigated. The experiments have been performed on a self-optimizing NUMA system, the Sun Orange prototype, using different data placement and thread scheduling strategies. The results show that correct data placement is very important for the performance for all solvers. However, the Orange system has a unique capababilty of automatically changing the data distribution at run time through both migration and replication of data. For reasonable large PDE problems, we find that the time to do this is negligible compared to the total solve time. Also, the performance after the migration and replication process has reached steady-state is the same as what is achieved if data is optimally placed at the beginning of the execution using hand tuning. This shows that, for the application studied, the self-optimizing features are successful, and shared memory code without explicit data distribution directives yields good performance.

1 Introduction

Many important phenomena in nature are modeled by partial differential equations (PDEs). For example, such equations describe flow of fluids and gases, and propagation of electromagnetic and sound waves. Also, PDEs arise in many other application areas ranging from chemistry to economics. The equations are often impossible to solve analytically and numerical solution methods must be employed. The discretizations used can mainly be categorized into three groups; *finite difference methods, finite element/volume methods,* and *spectral methods.* Finite difference methods normally employ structured discretizations of the domain while finite element/volume methods are often used for unstructured grids. Spectral methods can only be applied for specific combinations of geometries and boundary conditions, and use global operations such as Fourier transforms to approximate the spatial derivatives.

In realistic settings, the numerical solution of PDEs requires large-scale computation. The computations are often carried out on parallel computers. Recently, cc-NUMA (cache coherent non-uniform memory access) parallel computers with OpenMP as a programming model has gained in popularity. In this paper, we will evaluate how different PDE solvers perform on a self-optimizing cc-NUMA architecture built from SMP nodes. A similar investigation has earlier been performed for a simple finite difference solver kernel [9]. We compare three more complex PDE solver kernels, representing realistic problems, to investigate both the performance and the effects of automatic optimization strategies such as migration and replication.

2 Generic PDE solvers

We have chosen three different numerical kernels representing the three types of numerical methods used for solving PDEs; a finite difference method for solving the non-linear Euler equations (gas flow), a finite volume method for solving Maxwell's equations (electromagnetic wave propagation), and a pseudospectral method for solving the time-dependent Schrödinger equation (ab-initio modeling of chemical reactions).

2.1 Finite difference solver

The finite difference kernel solves the time-dependent Euler equations in a three dimensional orthogonal curvilinear coordinate system. The geometry is discretized with a structured grid and spatial derivatives are approximated by central difference operators, including realistic artificial viscosity. Further details on the discretization are given in [2].

Since we are using a curvilinear grid, the coefficients in the finite difference stencil will vary over the grid. Furthermore, the Euler equations are non-linear and the stencil coefficients depend on the solution. Thus, the operator coefficients have to be recomputed at every time step. However, the structure of the stencil is the same all over the grid. For the time-stepping, a standard Runge-Kutta scheme is used.

2.2 Finite volume solver

The finite volume kernel for the Maxwell equations is based on the integral formulations of Faraday's and Ampère's laws. The three dimensional geometry is discretized with an unstructured staggered grid using tetrahedrons in the primary grid, generated by a Delaunay grid algorithm. The time derivatives are approximated with an explicit time-stepping scheme, the third-order Adams-Bashforth method. The update of the field variables is performed as matrix-vector and vector-vector operations, where the matrices are stored in compress sparse row format. For further details on the solver see [3]. The application relates to an aircraft where the surface currents are computed after an electro-magnetic pulse hits the nose, see Figure 1.

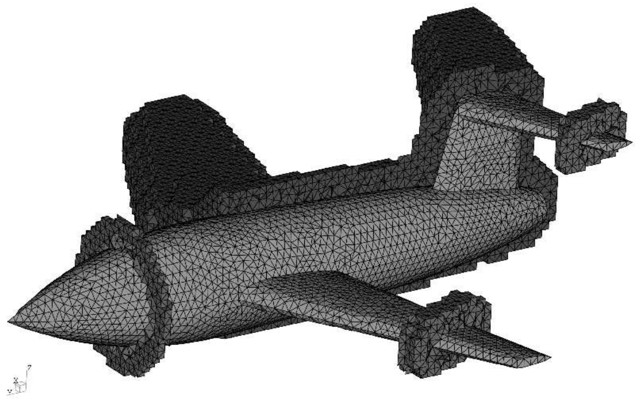


Fig. 1. Cross-section of the unstructured grid around a model aircraft used in the finite volume solver.

2.3 Pseudospectral solver

In the kernel of the pseudospectral method for the time-dependent, two-dimensional Schrödinger equation, the computation of the spatial derivatives is performed within a standard second order accurate split-operator time-marching procedure. Further details on the scheme are given in, for example, [6]. The computation consists of a convolution, i.e. a transform to frequency space, a local multiplication, and an inverse transform [10]. The grid is uniform, and the FFT is used for the transforms, resulting in $\mathcal{O}(n^2 \log_2 n)$ arithmetic complexity for a grid with n^2 grid points.

The standard implementation of a 2D FFT is first to perform 1D FFTs for all the columns in the data matrix, and then do the same for all the rows. In a convolution computation, this results in a five-stage scheme illustrated in Figure 2.

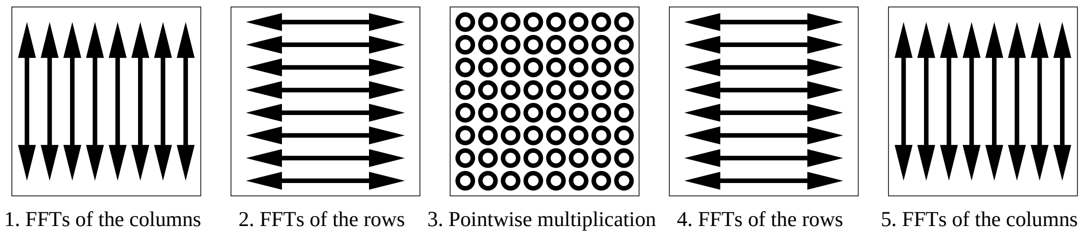


Fig. 2. A single convolution computation for a 2D problem.

In general, it is sufficient to study 2D problems to gain insight into the performances for multi-dimensional pseudospectral solvers, since the FFTs for the extra dimensions will be performed locally.

In Figure 2, each arrow represents a 1D FFT. For a vector of length n, this is a $\log_2 n$-stage computation involving a rather complex but structured communication pattern. A number of different FFT algorithms are available, for a review see, for example, [7]. In the experiments presented here, we use an in-place radix-2 Gentleman-Sande version of the FFT, and an in-place radix-2 Cooley-Tukey version for the inverse transforms. This allows for a convolution algorithm where no bit-reversal permutations are required. This is important, since the bit-reversal permutation introduces a lot of communication, which affects the performance considerably. Also, the FFTs should be performed in situ. If workspace is used, the maximal number of grid points is reduced leading to a less accurate solution of the Schrödinger equation.

3 A self-optimizing cc-NUMA architecture

The Sun Orange system [5] is a prototype architecture developed to evaluate a scalable alternative to symmetric multiprocessors (SMPs). The system can be viewed as a cc-NUMA computer with self-optimizing features, built from unusually large SMP nodes. Up to four nodes, each with up to 28 CPUs, can be directly connected by a point-to-point network between the Orange Interfaces (OI) in each node.

The experiments presented in this paper have been performed on the two-node Orange system at the Department of Scientific Computing, Uppsala University. Each node has 16 processors (250 MHz UltraSPARC II with 4 Mbyte L2 cache) and 4 Gbyte memory. Logically, there is no difference between accessing local and remote memory, even though the access time varies: 310 ns for local and 1700 ns for remote memory. Coherence between all the 32 caches is maintained in hardware, which creates an illusion of a system with 8 Gbyte shared memory. The processors are by now one generation old, but the modern self-optimization features built into the system are not found in any current commercially available system. We regard the Orange system as a prototype for a kind of parallel computer architecture of the future. Furthermore, it is reasonable to assume that the relations between processor, memory and interconnect speeds in such future systems will be similar to those of the Orange system, implying that the results presented below will remain relevant.

On a cc-NUMA system, an application could be optimized by explicitly placing data in the node where it is most likely to be accessed. In order to ease the burden on the programmer, different forms of optimization may be supported by the system. For example, on SGI Origin systems, pages may be migrated to the node where they are used by a page migration daemon. On the Orange system, a similar software daemon also detects pages which have been placed in the wrong node and migrates them to the correct node. However, the Orange daemon uses other algorithms for detecting candidate pages for migration. Furthermore, the

Orange system also detects which pages are used by threads in both nodes and replicates them, hereby avoiding ping-pong effects. The cache coherence protocol maintains the coherence between replicated memory pages with a cache line granularity. This is called *Coherent Memory Replication* (CMR), but the technique is also sometimes referred to as Simple COMA (S-COMA) [4]. The maximum number of replicated pages as well as other parameters in the page migration and CMR algorithms may be altered by modifying system parameters.

4 Orange configurations

On the Orange system, allocation of data uses a first-touch policy. The allocate statement reserves virtual address space, and the physical memory is allocated on the node where the thread first touching the data resides.

The default thread scheduling policy is, if possible, to confine the threads to a single node. Only if the number of threads is larger than the number of processors in the first node are threads spawned on the other node. The threads normally stay on the processor they are spawned on. Threads can be explicitly bound to a specific SMP node using a system call. By employing the first-touch policy and thread binding, it is possible to examine the performance effects of where threads are spawned and where the data is initially placed. If both page migration and CMR are disabled, the code will run in pure cc-NUMA mode.

The configurations in table 1 have been tested. Here, memory can be either thread matched (TM) or single node (SN) allocated. In the thread matched allocation, data is allocated on the node where each thread resides. Single node allocation locates all data on one of the nodes. The threads can either be bound to a specific node or be unbound, whereas the Orange system uses its default scheduling policy. In the case of bound threads, the threads are evenly distributed between the two nodes. Different optimization strategies have been used; enabling migration and replication (m/r), enabling migration (mig), enabling replication (rep) and disabling both migration and replication.

Table 1. Orange configurations

Configuration	1	2	3	4	5	6	7	8	9
Allocation of memory	SN	SN	TM	SN	TM	SN	TM	SN	SN
Bound threads	-	no	no	yes	yes	yes	yes	yes	yes
Optimization	off	m/r	m/r	m/r	m/r	off	off	rep	mig

Configuration 1 corresponds to a single SMP server, where the number of threads is limited to 16 in our experiments. Configurations 2 and 3 represent the default system configuration differing only in the memory allocation; either data is initialized serially (Configuration 2) or data is initialized in parallel (Configuration 3). Configurations 4-9 all have bound threads where the load is balanced between the two nodes, but use different memory allocation and optimization strategies. Configurations 6 and 7 represent pure cc-NUMA systems, one with an unfavorable (6) and one with optimal (7) memory allocation. Configurations 4, 6, 8 and 9 differ only in the self-optimization strategies that are enabled.

5 Parallelization of the PDE solvers

The parallel codes for the solvers are written in Fortran 90 using double precision data (complex data for the pseudospectral solver). The programs were compiled and parallelized using the Sun Forte 6.2 early access 2 compiler employing OpenMP-directives. The experiments were performed on a dedicated system, measuring wall-clock timings.

5.1 Parallelization of the finite difference solver

The kernel of the finite difference solver consists of three nested loops that traverse the grid. At each grid point the operator is constructed and applied. The loops are located within a parallel region and the outermost loop is parallelized with an OpenMP-directive using static scheduling. This implies that the domain is divided into stripes, with each thread always updating the solution in the same subdomain. Furthermore, since finite difference stencils are local operators, the need for communication between different subdomains is limited to the subdomain boundary points.

5.2 Parallelization of the finite volume solver

The kernel of the computations consists of matrix-vector products. The matrices are constructed once in a serial section of the code. Consequently, we can only use the single node allocation configurations. All of the matrix-vector products are parallelized at loop-level within a single parallel region, using static scheduling over the rows. The matrices are unstructured, and the workload varies between the rows. However, the load imbalance is very small for large problems. Use of dynamic scheduling only marginally improves the performance for the SMP configuration 1. If the self-optimizing features are enabled, dynamic scheduling makes it impossible for the software daemon to recognize pages for migration and replication since the data access pattern changes between iterations. The communication is irregular and depends on the numbering of the nodes. No effort is made to renumber the nodes in a more favorable way.

5.3 Parallelization of the pseudospectral solver

The FFTs in the pseudospectral solver kernel are first performed for the columns in the matrix and then for the rows. Cache utilization becomes extremely poor for large problems if the algorithm is applied directly to the rows. Instead, a more efficient implementation is employed, where the matrix is transposed in parallel

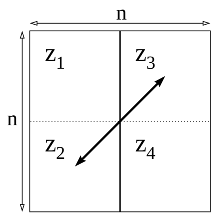


Fig. 3. The $n \times n$ matrix z consists of the blocks z_1, z_2, z_3 and z_4. If the data is evenly distributed between the two SMP nodes, the z_2 and z_3 blocks will travel across the Orange interface when the matrix transpose is applied.

before applying the FFTs. The algorithm is parallelized using OpenMP by dividing the columns of the matrix between the threads. The FFTs are computed internally in each thread.

On a two-node system with evenly distributed data, the transpose operation leads to an exchange of data located in the upper right matrix block and the lower left matrix block between the nodes, see Figure 3. Because of the matrix transposes the pseudospectral solver has the densest communication requirement of the three PDE solvers.

6 Impact of migration and replication

The impact of migration and replication on performance for the three solvers is summarized in Figures 4, 5 and 6. In order to show the behavior over time, iteration time is plotted versus iteration number. The codes were executed using 24 threads and run long enough to reach a steady-state iteration time. The problem sizes were chosen to yield steady state iteration times of the same magnitude for the three solvers.

The finite difference and pseudospectral solvers show similar iteration times for the thread matched memory allocation configurations 3, 5 and 7. These configurations achieve a steady-state iteration time after only the first iteration. The memory has optimal placement from the onset. Balanced distribution of the threads between the nodes does not appear to influence the iteration time significantly. The bus utilization on a single SMP can therefore be expected to be low.

All configurations with single node memory allocation and enabled Orange optimizations - Configurations 2, 4, 8 and 9 - show decreasing iteration time during the first iterations. During that time, memory is either migrated or replicated in order to reach a more beneficial placement on the nodes. The finite difference solver takes the longest time to reach a steady-state due to its large problem size. Configuration 6, representing a pure cc-NUMA system, introduces many more remote accesses due to a disadvantageous memory placement and exhibits significantly worse performance.

Using all available processors on a single node for computation leads to large variations in iteration times, probably because activities of other processes executed by the operating system stall the computation. This is most apparent in Configuration 2.

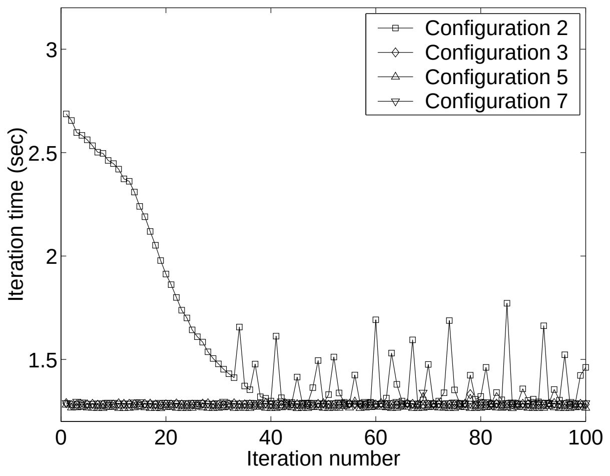
For the finite difference solver, Configuration 4 takes approximately 50 percent longer than Configuration 2 to reach steady-state. This is due to the fact that for Configuration 2, only 8 of the 24 threads run in the node where data is not initially placed, whereas for Configuration 4 there are 12 threads running in each node. Since each thread uses a fixed amount of data, there will be 50 percent more memory pages that have to migrate in configuration 4 than in Configuration 2. This accounts for the increase in time needed to reach a steady-state iteration time.

Subfigures 4(b), 5(b) and 6(b) shows the different optimization strategies for the PDE solvers. It is apparent that Configuration 8, which implements pure memory replication, is the quickest to reach a steady-state iteration time for the finite difference and finite volume solvers. The pseudospectral solver has a more complex communication scheme and the effect here is less pronounced. This phenomenon is described in [9] and is due to the implementation of the migration and replication techniques. When a page is to be migrated from one node to another, a *whole* page (8192 bytes) must be copied. If the page is instead replicated, the replication takes place on a cache-line (64 bytes) basis, thereby preventing unnecessary copying. Another advantage with replication compared to migration is that it lets the optimization daemon (see section 3) work more efficiently. Since the daemon is only allowed to run intermittently, it will be able to mark more pages for replication if it does not have to copy pages too. On the other hand, using replication is more memory consuming. Instead of migrating a page from one node to another thereby using the same amount of memory, replicating the page will require twice as much memory.

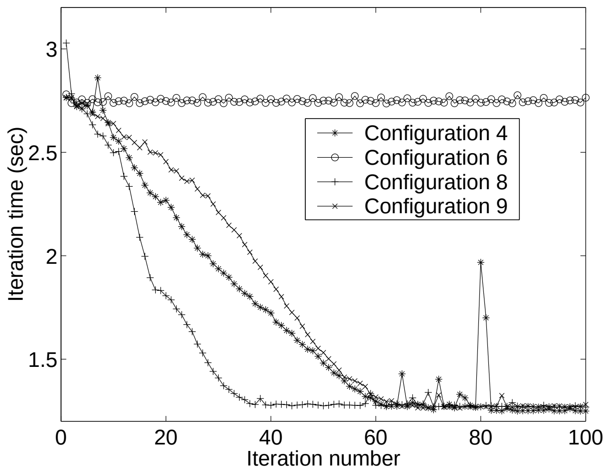
In Table 2 the number of migrated and replicated pages are presented for the different optimization strategies after the configurations have reached steady-state iteration times. The finite difference and finite volume solvers primarily migrate pages in Configuration 4 where both migration and replication are enabled. The pseudospectral solver invokes comparably more replication. For the pseudospectral solver the number of replicated pages continues to grow also after a steady-state iteration time has been reached. All data in the bottom right corner of the matrix in Figure 3 is well suited for migration, whereas the data in the top right corner and bottom left corner have a more complex behavior and the Orange system employs both migration and replication.

7 Speedup

Figure 7 presents measurements of the speedup for the three different solvers. We show the speedup of the time per iteration when steady-state has been reached, i.e., after a large number of iterations (c.f. Figures 4, 5 and 6).

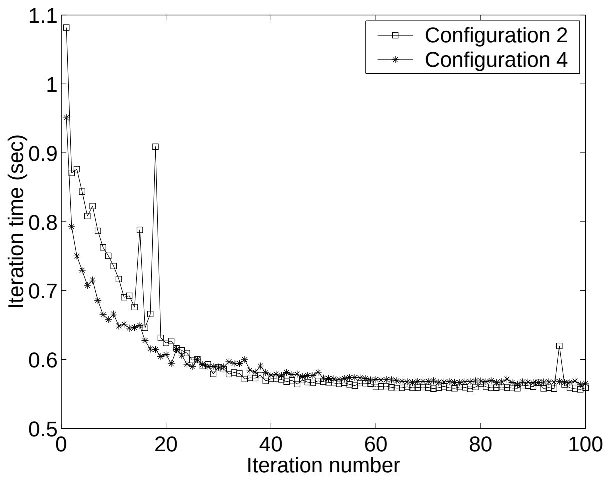


(a) Configurations 2,3,5 and 7

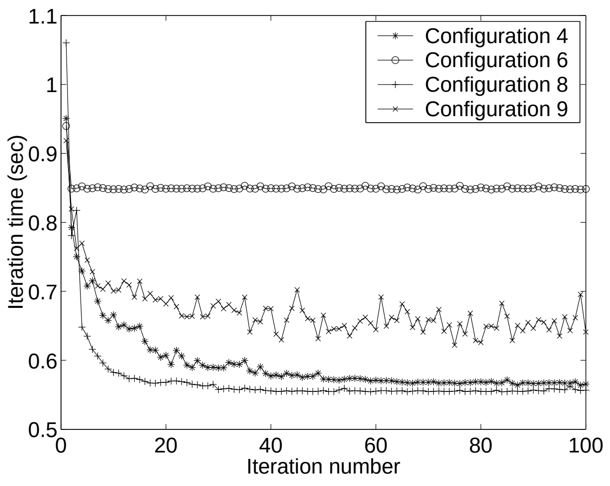


(b) Configurations 4,6,8 and 9

Fig. 4. Iteration times for the finite difference solver

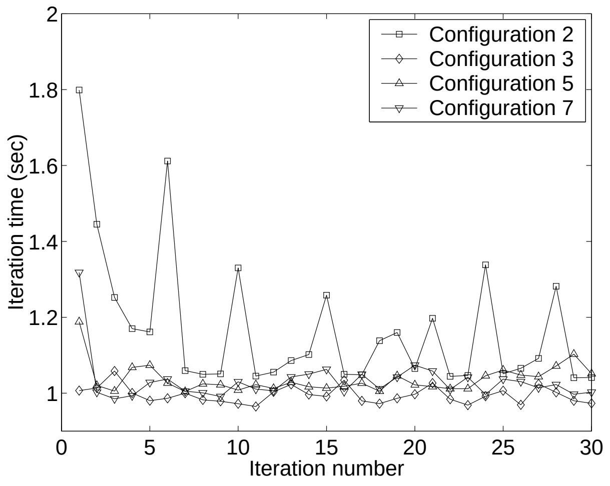


(a) Configurations 2 and 4

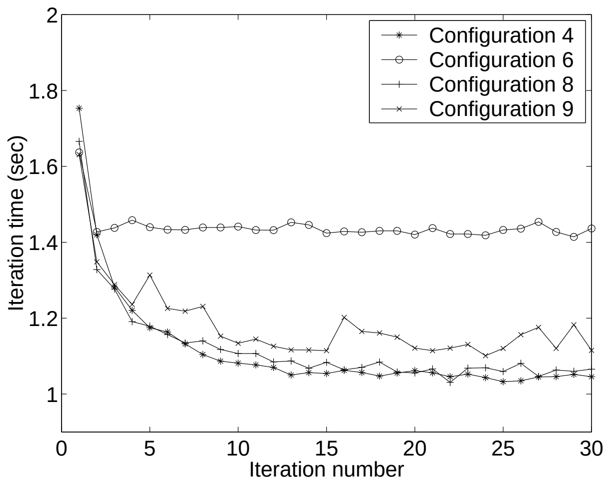


(b) Configurations 4,6,8 and 9

Fig. 5. Iteration times for the finite volume solver



(a) Configurations 2,3,5 and 7



(b) Configurations 4,6,8 and 9

Fig. 6. Iteration times for the pseudospectral solver

Table 2. Iteration time, number of migrated and replicated pages for the PDE solvers.

Configuration	Iter Time	# Migrs	# Repls
4	1.249	79179	149
8	1.267	N/A	79325
9	1.264	79327	N/A

(a) Finite difference solver

Configuration	Iter Time	# Migrs	# Repls
4	0.562	50087	401
8	0.552	N/A	10418
9	0.649	49860	N/A

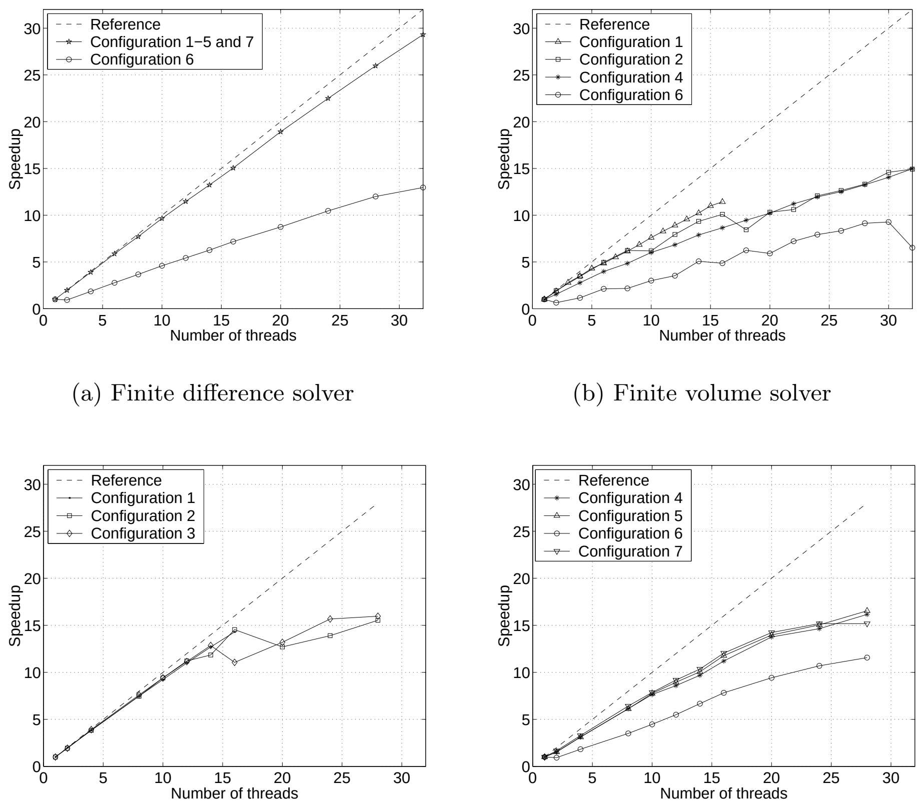
(b) Finite volume solver

Configuration	Iter Time	# Migrs	# Repls
4	1.06	6495	1794
8	1.04	N/A	8422
9	1.12	7455	N/A

(c) Pseudospectral solver

The finite difference solver shows an almost perfect speedup except for Configuration 6, where data is allocated on one node and replication/migration is disabled. Large amounts of data must then be communicated over the Orange interface in each iteration, since it resides on the remote node. With optimal data placement (thread matched allocation or steady-state of migration and replication) only boundary values between subdomains have to be communicated. This yields a very high ratio of computations per communication resulting in linear speedup for configurations 1-5 and 7.

In the pseudospectral solver we use global transposes of the data resulting in a lot of data movements. The single node SMP (configuration 1) and the Orange configurations using the standard scheduling policy (2,3) show very good scaling up to 14 threads. This is because all threads are scheduled to the same node and no explicit data movements are necessary. Above 16 threads we get scheduling on both nodes and communication over the Orange interconnect. This explains the drop in the performance in Subfigure 7(c). For the balanced configurations (4-7), there is a more even growth in speedup as the number of threads is increased. The amount of communication caused by remote accesses is constant, which should result in a smooth speedup. For configuration 6 we have the largest amount of remote accesses, resulting in poor speedup.



(a) Finite difference solver

(b) Finite volume solver

(c) Pseudospectral solver

(d) Pseudospectral solver

Fig. 7. Speedup curves for the PDE solvers

The unstructured finite volume solver has very irregular data dependencies making it difficult to optimize for loop-level parallelism. We get a lot of remote memory accesses when running on two nodes but also a high number of cache misses within the nodes. The speedup results are similar to the pseudospectral solver, but somewhat lower. We get the best performance running on a single SMP node. When scheduling threads on two nodes the performance drops due to the high number of remote memory accesses.

Problem size influences the speedup behavior depending on the amount of communication needed in each PDE solver. The communication/computation ratio decreases for larger problems, leading to improved speedup.

8 Conclusions

We have evaluated the performance of three different numerical kernels, representing three types of numerical methods for solving PDEs; finite difference methods, finite volume (element) methods, and spectral methods, on a self-optimizing cc-NUMA architecture. The codes were written in Fortran 90 and parallelized with OpenMP. We have experimented with different settings of data placement, thread binding, and page migration/replication.

For the finite difference method it is easiest to get good scaling because of the structured data and nearest neighbor dependencies. For large enough problems all methods will scale well. To get good scaling on a NUMA system, data placement is very important. This was discussed at SC2000 [1, 8]. On the Sun Orange system data distribution can be achieved with thread matched allocation, i.e. data is allocated on the node where the thread first touching the data is residing, or by letting a software daemon detect pages that are placed on the wrong node and migrate or replicate them. The experiments show that migration and replication of pages gives performance equal to the case where the pages were placed optimally from the beginning.

For the finite volume solver the time to migrate and replicate the pages is negligible compared to total time to solve the problem. The iteration time has levelled out, i.e. the migration and replication process has reached steady-state, within 100 iterations while the solver needs about 15000 iterations to compute the solution. Thread matched allocation is very difficult to achieve in the finite volume solver as the initialization and solver phases use completely different data access patterns. Without extensive restructuring of the code, data would be unfavorably distributed between the nodes. Also, data distribution directives would not help as it is impossible to know in advance the optimal page placements, unless the code is parallelized in MPI like fashion with different nodes responsible for different domains. Again, this would require extensive restructuring of the code.

A common feature of all three kernels is that they repeatedly access the same data set in an iterative numerical scheme. Many identical iterations allow the self-optimizing features to migrate and replicate pages in an optimal way. For PDE solvers where adaptive mesh refinement is employed, the sizes of the grids and the data partitioning changes during execution. For such solvers, different advanced load-balancing and re-partitioning software techniques have been developed, see for example [11]. A topic for future research is to examine the performance of self-optimizing systems also for problems with dynamically changing data structures. If good performance is achieved also here, this would greatly simplify the implementation of adaptive PDE solvers.

References

1. Bircsak J. et al., *Extending OpenMP for NUMA Machines*, Proceedings of Supercomputing 2000.

2. Brandén H., Holmgren S., *Convergence Acceleration for the Steady State Euler Equations*, Accepted for publication in Computers and Fluids.
3. Edelvik F., *Finite Volume Solvers for the Maxwell Equations in Time Domain*, Licentiate thesis 2000-005, 2000, Department of Information Technology, Uppsala University, Box 337, S-751 05 Uppsala, Sweden.
4. Hagersten E., Saulsbury A., Landin A, *Simple COMA Node Implementations*, Proceedings of Hawaii International Conference on System Science, 1994.
5. Hagersten E., Koster M., *WildFire: A Scalable Path for SMPs*, Proceedings of 5th International Symposium on High-Performance Architecture, 1999.
6. Leforestier C. et al., *A Comparison of Different Propagation Schemes for the Time Dependent Schrödinger Equation*, J. Comp. Phys., vol 94, 1991.
7. van Loan C., *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.
8. Nikolopoulo D. S. et al., *Is Data Distribution Necessary in OpenMP?*, Proceedings of Supercomputing 2000.
9. Noordergraaf L., van der Pas R., *Performance Experiences on Sun's WildFire Prototype*, Proceedings of Supercomputing 99, 1999.
10. Fornberg F., *A Practical Guide to Pseudospectral Methods*, Cambridge University Press, 1998.
11. Steensland J., *Efficient Partitioning of Dynamic Structured Grid Hierarchies*, PhD thesis, 2002, Department of Information Technology, Uppsala University, Box 337, S-751 05 Uppsala, Sweden.