Performance of a High-Accuracy PDE Solver on a Self-Optimizing NUMA Architecture

Dan Wallin
Uppsala University, Information Technology,
Department of Scientific Computing

February 23, 2001

Abstract

High-accuracy PDE solvers use multi-dimensional fast Fourier transforms. The FFTs exhibits a static and structured memory access pattern which results in a large amount of communication. Performance analysis of a non-trivial kernel representing a PDE solution algorithm has been carried out on a Sun WildFire computer. Here, different architecture, system and programming models can be studied. The WildFire system uses self-optimization techniques such as data migration and replication to change the placement of data at runtime. If the data placement is not optimal, the initial performance is degraded. However, after a few iterations the page migration daemon is able to modify the placement of data. The performance is improved, and equals what is achieved if the data is optimally placed at the start of the execution using hand tuning. The speedup for the PDE solution kernel is surprisingly good.

Contents

1 Introduction

The kernel in many important computational codes consists of multi-dimensional fast Fourier transforms, i.e. 2D, 3D, or higher-dimensional FFTs. One area where such computations arises is the numerical solution of partial differential equations (PDEs) using spectral or pseudospectral discretizations. Such methods are used in a wide spectrum of applications, e.g. computations of turbulent flows for optimization of aircraft performance, numerical weather prediction, and ab initio computations for predicting the outcome of chemical reactions.

When using discretization methods employing structured grids, the data is represented as large multi-dimensional arrays where the size is determined by the number of grid points. Using many grid points generally yields a more accurate solution. For multi-dimensional problems, the resolution is in practice often limited by the amount of main memory available. The major advantage of employing pseudospectral discretizations is that, for many problems, it gives the best possible accuracy for a given number of grid points.

The time-consuming part in a PDE solution algorithm consists of computing approximations of the derivatives. Standard schemes with low or medium orders of accuracy use a local finite difference or finite element approach, where the derivative in a grid point is computed using only a small number of function values in neighboring points. For all reasonable parallel computer architectures, schemes of this type are very efficient. The arrays are distributed block-wise over the nodes, and the processes/threads in a given node performs the computations for the grid points in the local block. Remote accesses to memory in other nodes is only required for points close to the block boundaries. Furthermore, because of the surface-to-volume effect, the performance impact of communication decreases as the number of grid points is increased.

In a pseudospectral scheme, approximations of derivatives are performed using multi-dimensional FFTs, which are *global* multi-stage grid operations. Each stage has a specific communication pattern involving a large amount of data, and every value in the solution array is updated using information originating from all other grid points. At a first glance, this is a very difficult situation for parallel computations. However, since the communication patterns are static and highly structured, efficient parallel implementations are possible. A number of quite efficient parallel implementations for multi- dimensional FFTs have been developed. For example, the FFTW package [9] includes both a multi-threaded (Pthreads) and a message passing (MPI) implementation.

A multi-threaded implementation of a kernel representing a PDE solver employing a pseudospectral discretization [8] is studied in this report. The aim is to examine the parallel performance of an important non-trivial algorithm with significant inherent communication on a cc-NUMA system with SMP nodes [11]. For this realistic PDE solver problem, performance effects of self-optimizations such as page migration and replication are studied. A similar investigation has earlier been performed for a finite difference solver kernel [17], which only involves local grid operations and very little communication. It is of great interest for a programmer to know how successful the optimization techniques are. The result determines the importance of performing careful hand tuning, considering data allocation and thread scheduling policies.

The PDE solver algorithm based on a pseudospectral solver is described in section 2. In section 3 the Sun WildFire computer system is introduced. The

theory behind the fast Fourier transform is given in section 4 followed by a number of techniques of optimizing a serial 1D FFT in section 5. Finally, the PDE solver is parallelized and several performance experiments are presented in sections 6 and 7.

2 A generic PDE solver using a pseudospectral method

The high-accuracy derivative approximation in a pseudospectral solver is performed by a convolution, i.e. a transform to frequency space, a local multiplication, and an inverse transform back again. For a uniform grid, the FFT and its inverse yield a very efficient tool for the transformations, resulting in $\mathcal{O}(n^2 \log_2 n)$ arithmetic complexity for computing the derivatives on a grid with n^2 grid points. Normally, the computation is performed within an iterative solver or a time-marching procedure. Hence, a representative kernel for a pseudospectral solver is an iteration where the loop body consists of convolution computations.

The standard implementation of a 2D FFT is to first 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 described in Figure 1.

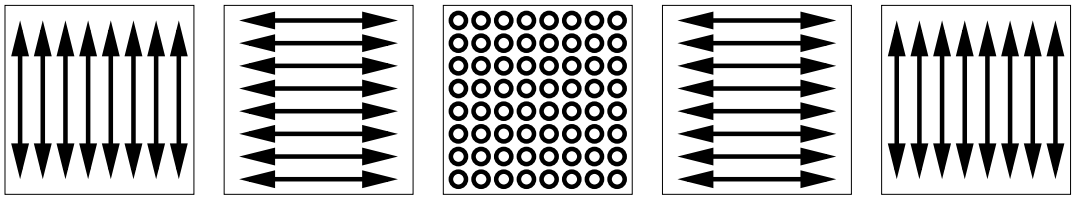


Figure 1: A single convolution computation for a 2D problem.

Each arrow In Figure 1 represents a 1D FFT. For a vector of length n, this is a $\log_2 n$-stage computation involving a rather complex but highly structured communication pattern, described in Figure 2.

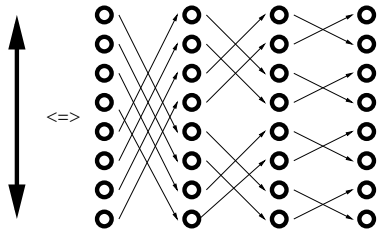


Figure 2: Communication scheme for a 1D FFT, often referred to as a FFT butterfly.

In general, it is sufficient to study 2D problems to get a picture of the performance also for multi-dimensional pseudospectral solvers, since the FFTs for the extra dimensions will be performed locally.

There are a number of different FFT algorithms available, for a review see, e.g. [15]. In the experiments presented here, an in-place radix-2 Gentleman-Sande version of the FFT, and a radix-2 in-place Cooley-Tukey version for the inverse transforms, further described in section 4, have been used. 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, and affects the performance significantly. Also, the FFTs should be performed in situ. If workspace is used, the maximal number of grid points is reduced, leading to a less resolved solution.

3 A self-optimizing cc-NUMA architecture

The Sun WildFire system [11] is a prototype architecture developed to evaluate a scalable alternative to symmetric multiprocessors (SMPs). WildFire can be viewed as a cache coherent non uniform memory architecture system (cc-NUMA) 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 WildFire Interfaces (WFI) in each node. For a description of various parallel computer architectures see [13].

The experiments presented in this paper have been performed on the two-node WildFire system Albireo at the Department of Scientific Computing, Uppsala University. A schematic sketch of Albireo can be found in Figure 3. Each SMP 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: 310ns for local and 1700ns 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.

A WildFire application can be optimized by explicitly placing data in the node where is most likely to be accessed. In order to ease the burden on the programmer, different forms of optimization are supported by the system. A software daemon detects pages which have been placed in the wrong node and migrates them to the other node. The daemon also detects pages used by threads in both nodes and replicates them. WildFire's cache coherence protocol keeps 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) [10]. The maximal number of replicated pages as well as other parameters in the page migration and CMR algorithms may be altered by modifying system parameters.

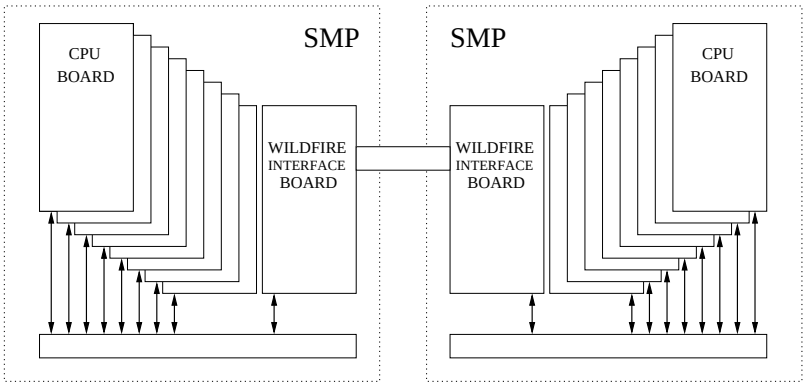


Figure 3: Schematic sketch of the Sun WildFire system Albireo.

4 Theory of fast Fourier transforms

The complexity of computing the discrete Fourier transform changed greatly in 1965, when Cooley and Tukey [4] showed that a discrete Fourier transform could be computed in $\mathcal{O}(n\ logn)$ operations. This was to become the fast Fourier transform. Computing the discrete Fourier transform using a straight-forward algorithm requires a much greater effort of $\mathcal{O}(n^2)$ computations.

4.1 Discrete Fourier transform

The general formula for the Fourier transform of a continuous function is given by

$$y(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt \ . \tag{1}$$

The discrete Fourier transform is the counterpart of (1) in a n-dimensional space,

$$y_k = \sum_{j=0}^{n-1} \omega_n^{kj} x_j \quad k = 0, ..., n-1 \ , \tag{2}$$

where

$$\omega_n^{kj} = e^{-\frac{2\pi i}{n}kj} \ . \tag{3}$$

Instead of expressing the formula as a sum, a matrix notation can be used,

$$y = F_n x \ , \tag{4}$$

y is a vector of size n and F_n is called the Fourier matrix, where the entries are given by

$$[F_n]_{kj} = \omega_n^{kj} = e^{-\frac{2\pi i}{n}kj} \quad k, j = 0, ..., n-1 \ . \tag{5}$$

4.2 Fast Fourier transform

The idea of the fast Fourier transform (FFT) is to find a relationship between the Fourier matrices F_n and $F_{n/p}$ so that the problem of size n can be solved by combining solutions of p smaller problems of size n/p. This is the same approach as in many other applications, e.g. sorting and searching problems, normally referred to as the "divide and conquer" technique. The benefit of this approach is that the problem can be reduced to $\mathcal{O}(n\ logn)$ complexity. The focus will be on so called radix-2 FFTs, that is FFTs with $p = 2$, dividing the problem into two problems of half the size in each step.

The discrete formulation of the Fourier transform (2) can be split into two smaller sums

$$y_k = \sum_{j=0}^{n-1} \omega_n^{jk} x_j = \sum_{j=0}^{(n/2)-1} \omega_n^{2jk} x_{2j} + \sum_{j=0}^{(n/2)-1} \omega_n^{(2j+1)k} x_{2j+1} = \tag{6}$$

$$= \sum_{j=0}^{(n/2)-1} \omega_{n/2}^{jk} x_{2j} + w_n^k \sum_{j=0}^{(n/2)-1} \omega_{n/2}^{jk} x_{2j+1} = y_k^{'} + \omega_n^k y_k^{''} \ . \tag{7}$$

For $k \geq n/2$ the relation $w_n^{n/2+k} = e^{\frac{2\pi i}{n}\frac{n}{2}} + e^{\frac{2\pi i}{n}k} = e^{i\pi} w_n^k = -w_n^k$ holds, which implies that

$$\begin{cases} y_k = y_k^{'} + \omega_n^k y_k^{''} \\ y_{n/2+k} = y_k^{'} - \omega_n^k y_k^{''} \end{cases} .$$

In a matrix notation, the same relation can be formulated as

$$y = F_n x = \begin{pmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{pmatrix} \begin{pmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{pmatrix} \begin{pmatrix} even\ x_i \\ odd\ x_i \end{pmatrix} =$$

$$= \begin{pmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{pmatrix} \begin{pmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{pmatrix} P_n x \ ,$$

where $I_{n/2}$ is the identity matrix of size $n/2$ and

$$\Omega_{n/2} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & \omega_n & 0 & \dots & 0 \\ 0 & 0 & \omega_n^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \omega_n^{n/2-1} \end{pmatrix} .$$

The matrix P_n is a permutation matrix that reorders the vector x so that all even vector elements are on top and all odd vector elements at the bottom of the new vector; a so called bit reversal. The FFT is in this formulation well suited for solving with a recursive algorithm.

4.3 The Cooley-Tukey Algorithm

The Cooley-Tukey algorithm is an iterative version of the FFT. It is based on a factorization of the Fourier matrix F_n described above. If $n = 2^t$, the matrix can be factorized as

$$F_n = A_t ... A_2 A_1 P_n^T \ . \tag{8}$$

By applying each matrix A_q to the input vector x in the order $q = 1, 2, .., t$ the discrete Fourier transform can be computed iteratively. The matrix A_q has the form

$$A_q = \begin{pmatrix} B_{2^q} & 0 & \dots & 0 \\ 0 & B_{2^q} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & B_{2^q} \end{pmatrix} ,$$

where the number of block matrices B_{2^q} is given by 2^{t-q}. The block matrix B_{2^q} is the same matrix as seen in earlier equations

$$B_{2^q} = \begin{pmatrix} I_{2^q/2} & \Omega_{2^q/2} \\ I_{2^q/2} & -\Omega_{2^q/2} \end{pmatrix} .$$

The FFT can be computed in situ according to

$x \leftarrow P_n^T x$
for q=1:t
  $x \leftarrow A_q x$
end

4.4 The Gentleman-Sande Algorithm

Another way of computing the fast Fourier transform is to transpose the factorization of the Fourier matrix F_n. This is the Gentleman-Sande algorithm, having the same complexity as the Cooley-Tukey framework. In the Gentleman-Sande algorithm, the Fourier matrix becomes

$$F_n = P_n A_1^T A_2^T ... A_t^T \ . \tag{9}$$

This transposition does not change the matrix F_n, since it is symmetric. Note that, when transposing the factorization, the bit reversal step of the FFT can be carried out as the last step.

Transposing A_n^T leads to a transpose of the block matrices, which now will look like

$$B_{2^q}^T = \begin{pmatrix} I_{2^q/2} & I_{2^q/2} \\ \Omega_{2^q/2} & -\Omega_{2^q/2} \end{pmatrix} \ .$$

The algorithm is very similar to the Cooley-Tukey algorithm, differing only in the inner-most loop of the FFT and in the order that the matrices are multiplied with the input vector. The Gentleman-Sande algorithm can be programmed as

for q=t:-1:1
  $x \leftarrow A_q^T x$
end
$x \leftarrow P_n x$

4.5 Bit reversal

The fast Fourier transform based on the Gentleman-Sande algorithm contains two major steps. In the first step the Fourier transform is computed using so called butterfly operations (see Figure 2). The second step of is to rearrange the elements to the initial positions, using a bit reversal algorithm. The bit reversal algorithm is non-trivial and requires the same amount of integer arithmetic operations as the butterfly requires floating point operations, $\mathcal{O}(n\ logn)$.

The name bit reversal comes from way the reordering can be computed. A 3-bit example can be found in Table 1. In the conversion from the input to the

input order (decimal)	input order (binary)	output order (binary)	output order (decimal)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1: Bit reversal example

output vector, the bits are just rearranged with the most significant bit taking the position of the least significant bit etc.

In an efficient implementation, the bit reversal is normally implemented very similar to the FFT-algorithm described above. The permutation matrix P_n can be factorized as

$$P_n = R_1 R_2 ... R_m \ , \tag{10}$$

where each R_q matrix looks like

$$R_q = \begin{pmatrix} \Pi_{2^q} & 0 & \dots & 0 \\ 0 & \Pi_{2^q} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \Pi_{2^q} \end{pmatrix} \ .$$

The matrices Π_{2^q} are block matrices on the diagonal of the matrix R_q called "odd-even sort" matrices. These matrices contain ones in the odd positions in the row vectors making up the top half of the matrix and in the even positions in the row vectors making up the bottom half of the matrix, e.g.

$$\Pi_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \ .$$

4.6 Inverse FFT

The inverse FFT is very similar to the forward FFT. The inverse of equation (4) yields, in the Cooley-Tukey case, a factorization

$$F_n^{-1} = (1/n)\bar{A}_t ... \bar{A}_2 \bar{A}_1 P_n \ . \tag{11}$$

The algorithm is unchanged except that the complex conjugate of the A matrices should be used. At the end, the vector should also be normalized. The Cooley-Tukey inverse FFT can be programmed in the following way

$x \leftarrow P_n x$
for q=1:t
  $x \leftarrow \bar{A}_q x$
end
$x \leftarrow x/n$

5 Optimizing the serial 1D FFT

The first step in designing efficient FFTs in a PDE solver is to obtain good performance on a single non-parallelized problem.

The original code of the fast Fourier transform was written in Fortran 90 based on the Gentleman-Sande algorithm without the bit reversal phase. In the one dimensional case the algorithm is applied once on a one dimensional array of size $n = 2^t$. The original code can be found in appendix A.1 and can in a simple notation be written as

for q=t:-1:2
  $x \leftarrow A_q^T x$
end
$x \leftarrow A_1^T x$

The input vector x is operated on as

$$x = A_2^T(...(A_{t-1}(A_t x))) \; . \tag{12}$$

The last step, multiplication with A_1, is treated in a separate step. The final result is stored in the same vector as the input vector, resulting in a non-bit reversed Fourier transform performed in situ. The reason for treating the last step, multiplication with A_1 separately is to avoid unnecessary operations. The first position in the Ω_q-vector has the value 1.0 and it is therefore unnecessary to multiply with this term. The block matrix B_{2^1}, in the diagonal of A_1, will in this case look like

$$B_{2^1} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \; .$$

Each multiplication requires 6 floating point operations in this complex case. By avoiding these multiplications, the number of floating point operations will be reduced.

In this version of the Gentleman-Sande algorithm, the Ω_q-vector is computed only once before the FFT-subroutine is called. This is also an efficient way of reducing the number of operations. Another possibility would be to compute the vector for each size inside the algorithm, but many of the computations would be repeated in that case.

The number of floating point operations in the original code is $(10t-6)\cdot 2^{t-1}$.

5.1 Technique 1: Reduction of floating point operations

A very easy way of further reducing the amount of computational work is to avoid all unnecessary multiplications with ω_0, having the value 1.0. This can be done by modifying the code of the matrix-vector multiplication carried out in the loop over all matrices A_q. In each lap of the loop the multiplication with the ω_0 values is carried out separately. The number of computations can in this way be reduced not only in the last matrix-vector multiplication but also in the earlier multiplications. The total number of floating point operations can now be computed as

$$2^t(5t-6)+6 \; . \tag{13}$$

5.2 Technique 2: Breaking the iterative loop

Running the program leads to a lot of overhead when performing the last matrix-vector multiplications. The reason is that the non-computational work will become demanding when the inner loops of the matrix-vector multiplication has to be carried out many times for a small problem. This can be avoided by breaking the loop at an earlier stage and multiply with a larger precomputed matrix.

As described earlier the FFT, in the Gentleman-Sande formulation, can be factorized as

$$y = F_n x = P_n A_1^T A_2^T \dots A_t^T x = R_1 R_2 \dots R_t A_1^T A_2^T \dots A_t^T x \; . \tag{14}$$

Given the size of the problem, the matrices A_q^T can be computed in advance and will contain a constant block matrix B_{2^q} on the diagonal. The last matrices to be multiplied with the input vector contain small block matrices and can be precomputed and stored as a constant matrix in the program. For example, the block matrices in A_1 and A_2 look like

$$B_{2^1} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} and \; B_{2^2} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & -i & 0 & i \end{pmatrix} \; .$$

The multiplication of B_{2^1} and B_{2^2} with pieces of the input vector, correspond to the two last laps in the loop. The smaller the size of the B_{2^q}-vector the larger number of times the multiplication has to be performed.

Knowing the size of the problem, it is also possible to multiply the A_1 and A_2 matrices to reduce the number of iterations. The resulting matrix $A_{1,2} = A_1 A_2$ has a block matrix $B_{2^1 2^2}$, that will look like

$$B_{2^1 2^2} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \\ 1 & i & -1 & -i \end{pmatrix}$$

A problem of size n can now be computed by applying this $B_{2^1 2^2}$ matrix $n/4$ times to pieces of the input vector. Unfortunately this matrix is not symmetric, which is the case with the original F_n matrix. The reason for this is that no bit reversal has been carried out on these matrices. Performing a bit reversal is not trivial in this algorithm.

The block matrices B_{2^1}, $B_{2^1 2^2}$, $B_{1^2 2^2 3^2}$ and $B_{1^2 2^2 3^2 4^2}$ have been precomputed, having the sizes 2×2, 4×4, 8×8 and 16×16 respectively. These constant matrices were stored in an initialization stage of the FFT solver.

The number of floating point operations will be the same for all tested ways of breaking the iterative loop, described in section 5.2.1, 5.2.2 and 5.2.3. The number of flops can for a problem of size $n = 2^t$ be calculated with the formula

$$2^t \left(10 \left(\frac{t-q}{2^q} \right) + 8 \cdot 2^q - 2 \right) \; , \tag{15}$$

where q represents the stage at which the loop is broken, e.g. for $q = 2$ the two last laps of the loop are avoided and are replaced by multiplying with $A_{1,2}$.

5.2.1 Matrix-vector multiplication

The first approach was to apply the block matrix of size p to n/p pieces of the input vector. Each matrix-vector multiplication was carried out by a call to the BLAS-2 subroutine, *ZGEMV* [5]. This routine multiplies a general matrix with a vector with complex variables of double precision. Unfortunately, the routine requires that the result is stored in a temporary variable of the same size as the number of rows in the matrix, which leads to some extra operations. Especially when the problem is large, the number of calls to the subroutine increases rapidly.

5.2.2 Matrix-matrix multiplication

Another possibility was to rearrange the input vector into a matrix with pieces of the vector stored row-wise in the matrix, according to Figure 4.

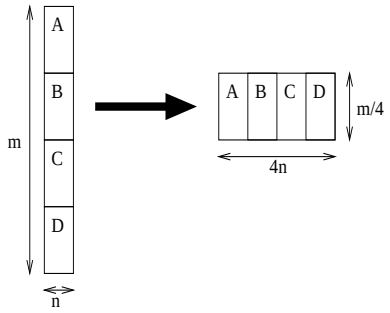


Figure 4: The input vector of size $m \times n$ is reordered into a matrix of size $n/4 \times 4m$.

This gives two general matrices that can be multiplied with each other. Using an efficient BLAS-3 subroutine, *ZGEMM* [6], good performance could be expected. After the multiplication the resulting matrix has to be reordered once again into a vector of size n, thus leading to some extra work.

5.2.3 Non-BLAS matrix-vector multiplication

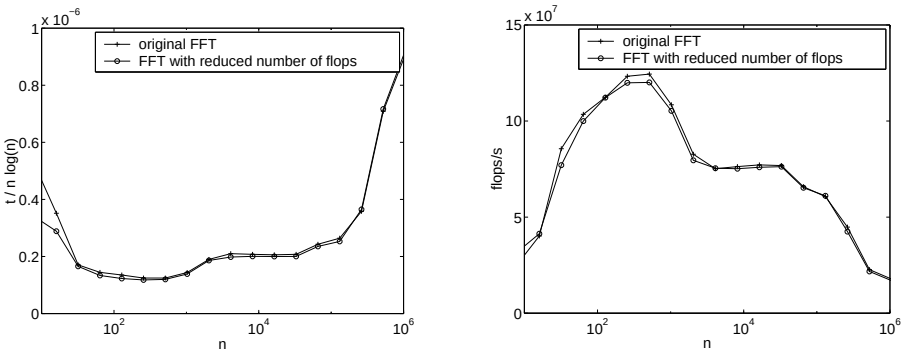
The last approach was to implement an own matrix-vector multiplication routine not based on a BLAS library. By doing this, work could be avoided by not calling any subroutines.

5.3 Performance of the serial 1D FFT

In the following section results are presented for the different implementations written for the serial 1D FFT. There are two kinds of graphs; graphs showing the wall-clock time for solving a problem for different sizes of problems and graphs showing the number of floating point operations per second (flops/s) that each implementation yields. The different versions are compared to the original code in appendix A.1.

The $\mathcal{O}(n\ log_2 n)$ complexity of the FFT makes it difficult to compare the timing results for problems of different sizes. The timing graphs show the time divided by $n\ log(n)$ on the y-axis to get a size independent value of the time required for a single operation. All graphs have the logarithm of the size on the x-axis.

The first test was to use technique 1 which reduces the number of floating point operations by avoiding multiplications with the value 1.0 contained in the vector Ω.



(a) Time

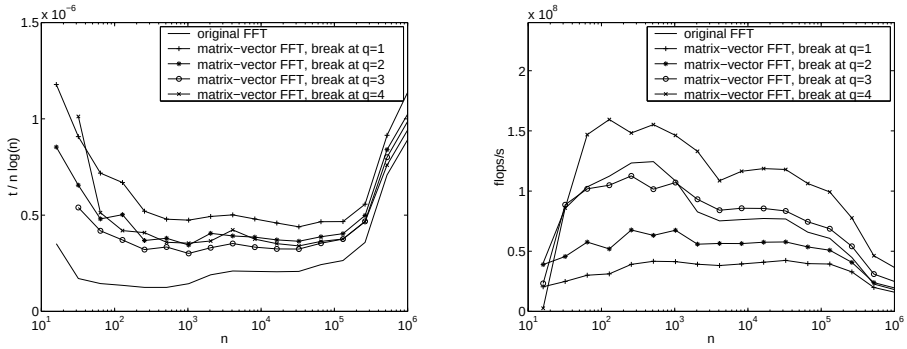
(b) Flops/s

Figure 5: FFT with reduced number of floating point operations compared to original FFT (technique 1).

As can be seen in the graphs in Figure 5 the new implementation was almost equivalent in speed to the original code. The new code seems to be somewhat faster for small problems of size smaller than 10^1. The difference in number of floating point operations is larger relative to the problem size for these problems. Unfortunately, these problem sizes are too small for being of interest in a real application. The performance gain is therefore limited using this technique.

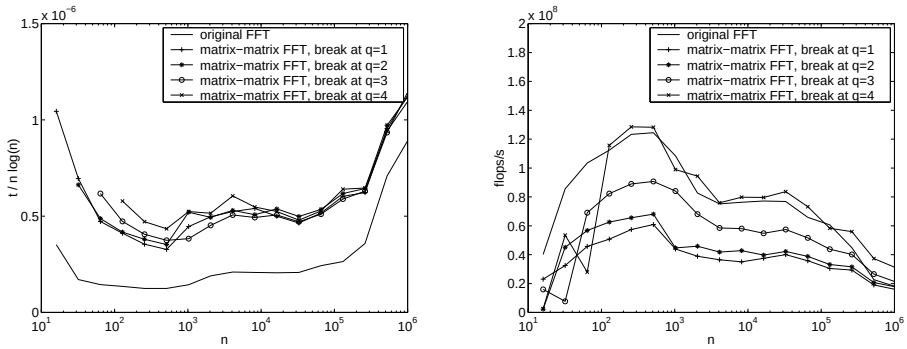
All the three algorithms that break the loop at an earlier stage and multiply with a precomputed matrix (technique 2) gave rather similar timing results. None of the algorithms were faster than the original code. According to the graphs in Figures 6 - 8, the number of flops/s becomes large when the loop is broken at an early stage, that is the block matrix is large. Unfortunately many of these extra operations are introduced in the matrix multiplications and are unnecessary in the original implementation.

The non-BLAS multiplication was the fastest implementation of technique 2, even though the differences were small. The reason for this was probably that this method caused the least amount of overhead when rearranging the input vector and calling subroutines. The worst implementation was the matrix-matrix multiplication. The rearranging phase described in Figure 4, takes a large amount of time.



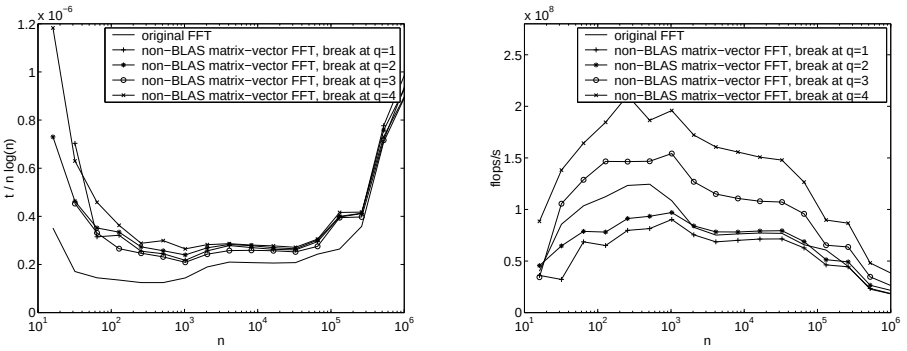
(a) Time (b) Flops/s

Figure 6: Technique 2, the loop is broken at stage q and a multiple matrix-vector multiplication is performed.



(a) Time (b) Flops/s

Figure 7: Technique 2, the loop is broken at stage q and a matrix-matrix multiplication is performed.



(a) Time

(b) Flops/s

Figure 8: Technique 2, the loop is broken at stage q and a non-BLAS multiple matrix-vector multiplication is performed.

The differences in time, compared to the original code, are smaller the larger the problem size. This is an expected result, since the reason for multiplying with a matrix was to reduce the large number of times the inner loop has to be performed, which grows rapidly with the size of the problem.

The Figures 6 - 8 also show that the most beneficial stage to break the loop is when $q = 3$ for most problem sizes and multiplication techniques. The balance between the number of matrix multiplications and avoided innerloop iterations seems to be optimal for $q = 3$.

5.4 Serial optimization conclusion

The original code shows a good performance compared to the new codes. A somewhat better performance can be achieved for very small problems by avoiding unnecessary multiplications with 1.0 contained in the Ω-vector. The cost of using this technique is a more advanced code. Because of the limited interest of such small problem sizes, the original code is best suited for further studies.

6 Parallelization

To obtain considerable gain in performance compared to the implementations in earlier sections, the problem was parallelized on the WildFire computer described in section 3. In this section several approaches of obtaining efficient implementations have been tested.

The codes were written in Fortran 90 using double precision complex (16 byte) data. The program was compiled and parallelized using the Sun Forte 6.1 compiler. All experiments were performed on a lightly loaded system using the original FFT algorithms, presented in appendix A.

There exists many efficient implementations of FFTs. Sun Performance library [19] contains highly optimized routines for both 1D and 2D FFTs. These library routines outperform most of the algorithms tested here, but do not satisfy the requirement of being performed in situ without bit reversal as mentioned in section 2.

The first experiments were carried out on a simple 1D FFT. At a later stage the more realistic PDE solve problem, involving 2D FFTs, were tested on the computer.

6.1 Parallelization techniques

Two major paradigms of writing parallel code were used, OpenMP and MPI. Both OpenMP directives and MPI bindings are supported by the Sun Forte 6.1 compiler. The Sun implementations supports version 1.1 of OpenMP [18] and version 1.1 of MPI [14].

The main difference between the MPI and the OpenMP codes is the use of processes and threads. In a MPI program a number of independent processes, each having a private address space, is started. The processes can interact with each other using MPI bindings that explicitly exchange data. OpenMP invokes a single process that branches into several threads. The threads share the same address space and therefore no interprocess communication has to occur.

The first efficient implementations of 1D FFTs were written using MPI running two processes. Each process was bound to a separate node on the WildFire

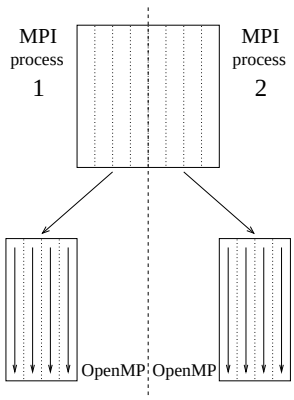


Figure 9: Two MPI processes are bound to separate nodes. On each node the FFTs are parallelized with OpenMP.

system and branched into a number of threads using OpenMP directives, see Figure 9. The reason for this nested parallelism was that with separate MPI processes, it was easy to explicitly allocate data on the two separate nodes of the WildFire system. There are no OpenMP directives for explicit memory allocation on a cc-NUMA system.

Unfortunately, these MPI implementations are not working well on a 2D problem. The reason for this is that intense communication has to occur between the two processes in the matrix transpose phase of the 2D FFT. This communication is not parallelized and therefore yields limited performance.

Further studies showed that explicit memory placement could be done in the OpenMP implementations. Inserting OpenMP directives is usually rather simple in an already existing serial code compared to writing an efficient parallel MPI program. The shared memory model of OpenMP also makes it easy to solve 2D problems where the communication is performed in parallel. Another advantage of using OpenMP in this problem is that MPI parallelization is later to be used on higher levels of the solution of a pseudospectral PDE problem. All results presented below is therefore generated with OpenMP programs.

6.2 Multiprocessor architectures, system and programming models

The WildFire system can be used to test a number of different architecture models, thread placement strategies and memory allocation models.

Data allocation

The allocation of data normally 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.

Binding of threads to a node

The threads normally stay on the processor they are spawned at. The default scheduling policy is to, if possible, confine the threads to a single node. Only if the number of threads is larger than the number of processors in the first node, threads are spawned also on the other node.

Threads can be explicitly bound to a specific SMP node using the system call *pset_bind*. A short *C*-function *omp_bind* has been written and can be called by the Fortran program.

```
#include <sys/pset.h>
#include <stdio.h>
#include <stdlib.h>
void omp_bind_(int *cabinet){
  if(*cabinet == 1 || *cabinet == 2)
    pset_bind(*cabinet,P_LWPID,P_MYID,NULL);
  else if (*cabinet == 0)
    pset_bind(PS_NONE,P_LWPID,P_MYID,NULL);
  else
    fprintf(stderr,"Argument in omp_bind must be 0, 1 or 2");
}
```

The program takes an integer as an input. In case of 1 or 2, the thread is bound to a specific node. If the input argument is 0, the thread is released and can move freely between the nodes.

Migration and replication

The page migration and CMR algorithms briefly described in section 3 detects data that is frequently accessed and located on a remote node and moves it to a local node. These optimization strategies can be turned off on the WildFire system with the command *amctl*.

Tested multiprocessor configurations

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 listed below has been used. Here, thread matched allocation means that the data is allocated such that the vertical FFTs in phase 1 of the PDE solver can be computed without introducing any remote accesses:

1. **Single node SMP** - Data is allocated on one node. The threads are bound to the same node. Migration and replication are turned off.
2. **Single node allocation WildFire** - Data is allocated on one node. The threads are not bound, and the WildFire default scheduling algorithm is used. Migration and replication are turned on.
3. **Thread-matched allocation WildFire** - Data is allocated using thread matching. The threads are not bound, and the WildFire default scheduling algorithm is used. Migration and replication are turned on.
4. **Single node allocation balanced WildFire** - Data is allocated on one node. The threads are evenly distributed between the two nodes and bound. Migration and replication are turned on.
5. **Thread-matched allocation balanced WildFire** - Data is allocated using thread matching. The threads are evenly distributed between the two nodes and bound. Migration and replication are turned on.
6. **Single node allocation balanced cc-NUMA** - Data is allocated on one node. The threads are evenly distributed between the nodes and bound. Migration and replication are turned off.
7. **Thread-matched allocation balanced cc-NUMA** - Data is allocated using thread matching. The threads are evenly distributed between the nodes and bound. Migration and replication are turned off.

6.3 Parallelization of multiple 1D FFTs

The first problem to study was to iteratively computing multiple 1D FFTs of the columns in a matrix, i.e. phase 1 in Figure 1 only. The computation is embarrassingly parallel, and each 1D FFT is local to a thread. A number of tests to study possible performance gains were made, e.g. the constant Ω-vector was copied to the two nodes and different OpenMP parallelization directives were used. These tests did not show any significant gain in performance.

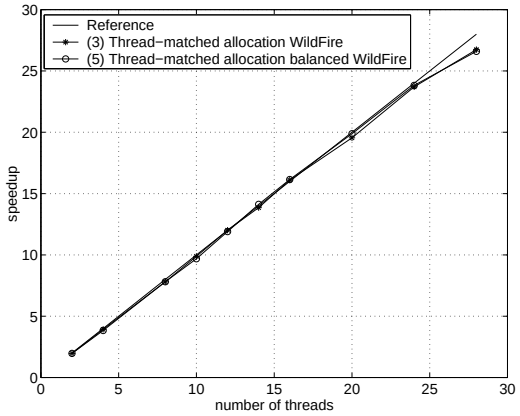


Figure 10: The speedup for configurations (3) and (5), compared to the execution time of a single thread. The grid size is 2048×2048.

For memory bound algorithms, spawning many threads on a single SMP node leads to a high load on the SMP bus. Distributing the threads in a balanced way on more than one SMP introduces a few remote accesses, but the performance may still improve because of the larger bus bandwidth available. In Figure 10, the speedup for computing a large number of iterations using the WildFire configurations (3) and (5) is shown. For both cases, the speedup is optimal. Also, the multiple 1D FFT computation reuses data in caches to some degree, and there is no apparent gain in balancing the thread distribution.

6.4 Parallelization of the pseudospectral solver kernel

As mentioned in section 2, the 1D FFTs in the convolution algorithm are first carried out for the columns of the data matrix, and then for the rows. For large number of grid points, experiments show that applying the FFTs directly to the matrix rows is not efficient. Using this type of implementation leads to extremely poor cache utilization, and the performance and the speedup for large problems is not acceptable. If the threads reside in both nodes, optimizations like page migration and CMR are not able to detect and adapt to the changing access pattern fast enough, and in practice almost no migration/replication occurs. Hence, a large amount of remote accesses further degrades the performance.

To improve cache utilization and to allow for more efficient communication between the nodes, experiments show that a better scheme is to explicitly transpose the data matrix, and again apply the FFTs to matrix columns. However,

recall that after applying the 1D FFTs in one direction, FFTs in the other direction should be computed. If the threads reside on more than one SMP node, some data will always be located on a remote node when the transpose is performed. On a two-node system with evenly distributed data, the lower left and the upper right matrix blocks will have to be exchanged between the nodes in the transpose operation, see Figure 11.

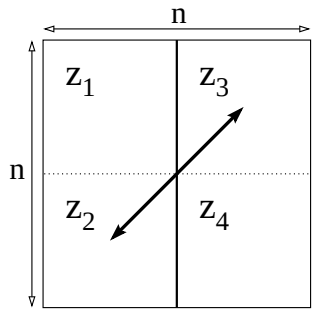


Figure 11: The $n \times n$ matrix z consists of the blocks z_1, z_2, z_3 and z_4. If the data is evenly distributed between the two SMP nodes, the z_2 and z_3 block will travel across the WFI when the matrix transpose is applied.

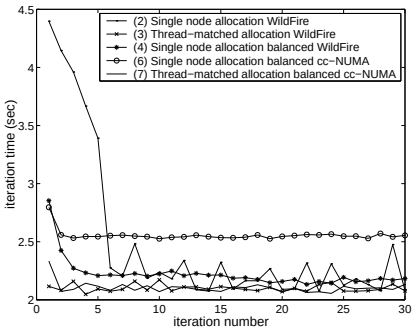
Half of the data matrix will bounce back and forth between the two nodes, still causing a large amount of communication over the WFI. The parallel transpose operation is in this implementation performed using the *ZTRANS* routine in the Sun Performance Library [19].

6.4.1 Impact of migration and replication

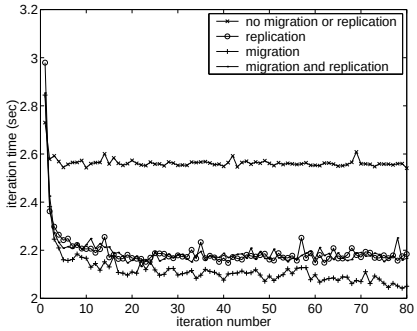
The time per iteration for the PDE solver kernel for several configurations using 24 threads is shown in Figure 12(a).

The performance results for the finite difference algorithm presented in [17] are derived using the single node allocation WildFire configuration (2). Using the same setting, the results for the pseudospectral solver kernel are similar. There is a significant decrease of the time per iteration during the first 6-7 iterations, and then the curve levels out. The WildFire optimizations move/replicate data from the remote to the local node, and remote accesses become more and more rare. Hence, the time per iteration decreases to a steady-state. Further investigation shows that the amount of replicated pages is small. The number of pages migrated is large at the beginning but decreases over time. The same phenomena is also present for the single node allocation WildFire with balanced thread scheduling (4), but here it is less pronounced. The reason could be that, when all processors on a single node are computing as in configuration (2), the bus is heavily loaded in this node, and the bandwidth available for page migration will be small. The page migration daemon will suffer from this, and the migrating pages will be unaccessible for a longer time.

Using all the threads in a single node for computations leads to large variation in iteration times, probably because activities of other users stall the computations. This is most apparent in the WildFire configurations with the default scheduling policy (2,3).



(a) Iteration times for the first iterations on different computer configurations.



(b) Iteration times for the single node allocation balanced WildFire (4) configuration using different optimization techniques.

Figure 12: Results for a 2048×2048 grid using 24 threads.

For the other configurations shown in Figure 12(a), the behavior is different. The first iteration takes longer time, but after this, a steady-state is immediately reached. Here, the relatively slow first iteration can be explained by cache effects. For the cc-NUMA configurations (6) and (7) the result is natural, since the adaptive optimizations are shut off. For the single node allocation balanced cc-NUMA configuration (6), one of the nodes perform exclusively remote accesses, leading to unbalanced execution times and a significantly larger time per iteration in steady-state.

The performance is almost the same for the thread-matched allocation WildFire (3) and cc-NUMA configurations (7). The memory is initially optimally placed for the first FFT, and in the matrix transpose a minimal amount of communication takes place. The WildFire optimizations are not activated, but it is also clear that they do not introduce any performance degradation.

Figure 12(b) shows an interesting, but not yet fully understood, result. Here, the single node allocation WildFire (4) configuration has been tested with different optimization strategies. With no migration and replication, the configuration is equivalent to the cc-NUMA case (6). The default setting is to enable both optimizations. Interestingly, the best results are achieved when only migration is enabled. Similar results have been observed also for a number of different problem sizes.

6.4.2 Speedup

Speedup results for a 2048×2048 grid are shown for a number of different configurations in Figure 13. The graphs show the average time per iteration when steady-state has been reached, c.f. Section 6.4.1. The results are normalized by the execution time of a single thread.

In general, the results are remarkably good. As mentioned before, the algorithm uses global operations, and involves heavy communication. The single node SMP (1) and the WildFire configurations using the standard scheduling

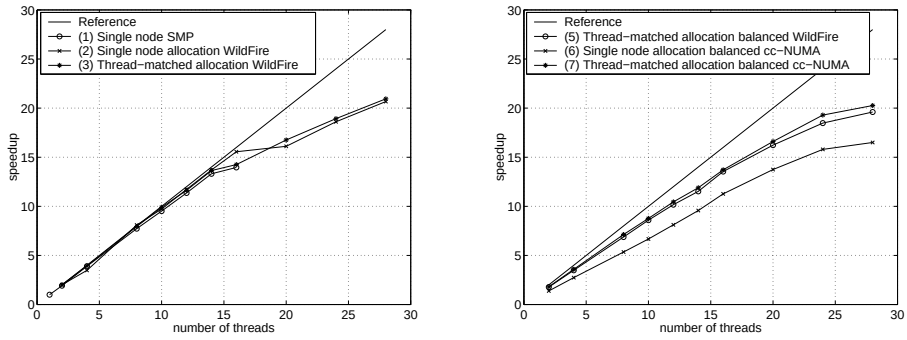


Figure 13: The speedup for different configurations, compared to the execution time of a single thread. The grid size is 2048×2048

policy (2,3) show very similar behavior up to about 14 threads. This is natural, since for these cases only one SMP node is involved in the computations. For 16 threads, the first SMP node is filled, and for the WildFire configurations (2,3), it is possible that one of the threads have been moved by the OS scheduler to the other (almost idle) SMP node. This is not possible for the SMP configuration (1), where the threads are bound to a single node. Again, the problem of computing on a filled SMP node results in a degradation of performance.

There is a short plateau in the speedup curve around 16 threads for the WildFire configurations (2,3). Here, the threads begin to be spawned on the other SMP node. For the balanced configurations (5,7), there is a more even growth in speedup as the number of threads is increased. The amount of communication causing remote accesses is constant, which should result in a smooth speedup curve. Note that, for less than 16 threads, the communication now results in that it is favorable to use the default thread scheduling, compared to spawning the threads in a balanced way on the two nodes, c.f. the results for multiple 1D FFTs in Section 6.3.

The speedup is considerably smaller for the single node allocation cc-NUMA configuration (6) than for the other configurations. The reason is again that a large amount of remote accesses are being performed by threads in one of the nodes.

6.4.3 Impact of problem size

For the single node allocation WildFire configurations, the number of iterations performed before steady-state is reached grows as the number of grid points is increased. This result is consistent with the results in [17]. The number of grid points used influences the speedup significantly. As the number of grid points grows, the computation/communication ratio increases, and the speedup grows.

As seen in Figure 14, there is no performance gain in using more than one SMP node for a small problem. However, as the problem size grows, the slope of the speedup curve once again approaches the ideal speedup ratio when the

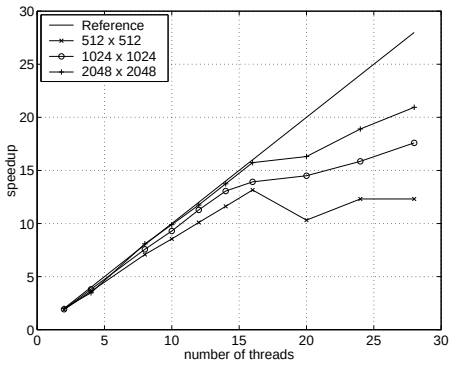


Figure 14: Thread-matched WildFire configuration (3) speedups.

"WildFire-plateau" mentioned in Section 6.4.2 has been passed. For a very large grid, possibly more than two dimensions, the performance gain from using more than one SMP node will be large. Note that, for such problems, the memory of the additional SMP nodes will probably also be needed.

7 Conclusions

The original code based on the Gentleman-Sande algorithm shows good performance for a serial 1D FFT problem. Replacing a number of iterations with a multiplication with a precomputed matrix, leads to longer run times due to added unnecessary computations.

The speedup on the WildFire system is very close to ideal for a perfectly parallelizable problem, like the multiple 1D FFTs in Section 6.3. Also for the pseudospectral solver kernel, which implements a non-trivial algorithm with heavy communication, the results are surprisingly good. Using 28 OpenMP threads distributed over two SMP nodes, the speedup is approximately 21. For problems of interest in application, the number of grid points will be even larger than used in the experiments, and the scalability will probably be further improved.

The WildFire system will perform page migration if the initial distribution of data over the SMP nodes is not optimal. After some iterations, a steady-state is reached where no further migration occurs. For all configurations where the data is optimally distributed in steady-state, the difference in performance is very small. The WildFire migration optimization makes up for programming errors and/or deficiencies in the programming model, without introducing a performance loss when the data is optimally placed from the beginning. Note that, if the data is allocated on only one of the nodes and the optimizations are disabled, i.e. the code is executed in pure cc-NUMA mode (configuration 6), the performance is significantly reduced.

The WildFire system using the default configuration exhibits a typical speedup behavior for a problem involving communication, e.g. the pseudospectral solver kernel: Until the number of threads is almost equal to the number of processors in an SMP node, the performance is identical to that of the SMP. When the number of threads is further increased, there is a short plateau in the speedup curve before it starts to grow again. If the problem is large enough, the slope of the speedup curve will again be close to optimal.

The speedup curve becomes smoother using a balanced thread scheduling policy. For the pseudospectral solver this implies a small performance loss when the number of threads is small, because of the large amount of communication over the WFI. However, distributing the threads in a balanced way over the SMP nodes might yields improved performance for a memory bound algorithm with a small amount of communication.

Note that the initial distribution of data has a large effect on the execution time if the convolution in the pseudospectral solver is only performed a small number of times. The goal of algorithm improvements, e.g. preconditioning, is to reduce the number of iterations in the computational scheme. It is important to make sure that the data is optimally distributed from the beginning if only a few iterations are required. There is currently discussion whether directives for data distribution should be included in OpenMP [16, 1]. Without such directives great care has to be taken when writing or porting the code in these situations.

References

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

[2] Cobby M., University of Strathclyde, http://www.spd.eee.strath.ac.uk/~interact/fourier/dft.html

[3] Cobby M., University of Strathclyde, http://www.spd.eee.strath.ac.uk/~interact/fourier/fft.html

[4] Cooley J.W., Tukey J.W., *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Comp. 19, 1965.

[5] Dongarra J. et. al., *An extended Set of Fortran Basic Linear Algebra Subprograms*, Argonne National Laboratory, 1986.

[6] Dongarra J. et. al., *A Set of Level 3 Basic Linear Algebra Subprograms*, Argonne National Laboratory, 1988.

[7] Falsafi M., Wood D. A., *Reactive NUMA: A Design for Unifying S-COMA with CC-NUMA*, Proceedings of ACM/IEEE International Symposium on Computer Architecture 1997.

[8] Fornberg F., *A Practical Guide to Pseudospectral Methods*, Cambridge University Press, 1998.

[9] Frigo M., Johnson S. G., *FFTW: An Adaptive Software Architecture for the FFT*, 1998 ICASSP proceedings (vol. 3, p. 1381).

[10] Hagersten E., Saulsbury A., Landin A, *Simple COMA Node Implementations*, Proceedings of Hawaii International Conference on System Science, 1994.

[11] Hagersten E., Koster M., *WildFire: A Scalable Path for SMPs*, Proceedings of 5th International Symposium on High-Performance Architecture, 1999.

[12] Itzkowitz M., *The Forte Developer 6 update 1 Performance Tools*, Sun Microsystems.

[13] Lenoski D. E., *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, San Francisco, 1995.

[14] *A Message-Passing Interface Standard*, Message Passing Interface Forum, University of Tennessee, June 1995.

[15] van Loan C., *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[16] Nikolopoulo D. S. et al., *Is Data Distribution Necessary in OpenMP?*, Proceedings of Supercomputing 2000.

[17] Noordergraaf L., van der Pas R., *Performance Experiences on Sun's WildFire Prototype*, Proceedings of Supercomputing 99, 1999.

[18] *OpenMP Fortran Application Program Interface*, Version 1.1, Nov 2000.

[19] *Sun Performance Library Reference*, Revision A, SunSoft Inc., Dec 1996.

A Code Excerpts

A.1 Original Gentleman-Sande Algorithm

```
subroutine gentleman_sande_fft(fft,z)
  type(fast_fourier_transform), intent(in) :: fft
  complex(kind=cfp), dimension(0:,1:) :: z
  integer :: m,n,lm,nbl,mbl,nbu,ibe,ibo,ie,io,l,bl,bu,i
  complex(kind=cfp) :: e,o

  m = fft%m
  n = size(z,2)
  if (m>1) then
     lm = fft%lm
     do i=1,n
        nbl = 1
        mbl = m
        do l=0,lm-2
           nbu = mbl/2
           ibe = 0
           do bl=0,nbl-1
              ibo = ibe + nbu
              do bu=0,nbu-1
                 ie = ibe + bu
                 io = ibo + bu
                 e = z(ie,i) + z(io,i)
                 o = fft%omega(nbl*bu)*(z(ie,i) - z(io,i))
                 z(ie,i) = e
                 z(io,i) = o
              end do
              ibe = ibe + mbl
           end do
           nbl = 2*nbl
           mbl = mbl/2
        end do
        ibe = 0
        do bl=0,nbl-1
           ibo = ibe + 1
           e = z(ibe,i) + z(ibo,i)
           o = z(ibe,i) - z(ibo,i)
           z(ibe,i) = e
           z(ibo,i) = o
           ibe = ibe + 2
        end do
     end do
  end if
end subroutine gentleman_sande_fft
```

A.2 Original Inverse Cooley-Tukey Algorithm

```
subroutine cooley_tukey_ifft(fft,z)
  type(fast_fourier_transform), intent(in) :: fft
  complex(kind=cfp), dimension(0:,1:) :: z
  integer :: m,n,lm,nbl,mbl,nbu,ibt,ibb,it,ib,l,bl,bu,i
  complex(kind=cfp) :: t,b

  m = fft%m
  n = size(z,2)
  if (m>1) then
     lm = fft%lm
     do i=1,n
        nbl = m/2
        mbl = 2
        ibt = 0
        do bl=0,nbl-1
           ibb = ibt + 1
           t = z(ibt,i)
           b = z(ibb,i)
           z(ibt,i) = t + b
           z(ibb,i) = t - b
           ibt = ibt + 2
        end do
        nbl = nbl/2
        mbl = 2*mbl
        do l=1,lm-1
           nbu = mbl/2
           ibt = 0
           do bl=0,nbl-1
              ibb = ibt + nbu
              do bu=0,nbu-1
                 it = ibt + bu
                 ib = ibb + bu
                 t = z(it,i)
                 b = conjg(fft%omega(nbl*bu))*z(ib,i)
                 z(it,i) = t + b
                 z(ib,i) = t - b
              end do
              ibt = ibt + mbl
           end do
           nbl = nbl/2
           mbl = 2*mbl
        end do
     end do
  end if
end subroutine cooley_tukey_ifft
```