HPCx CAPABILITY COMPUTING

3D FFTs on HPCx
(IBM vs FFTW)

Adrian Jackson and Gavin J. Pringle

June 30, 2003

Abstract

Fast Fourier Transforms (FFTs) are an essential part of many scientific codes: from Molecular Dynamics to Climate Modelling. It is, therefore, evident that HPCx requires efficient methods for performing FFTs and related calculations. This study compares the performance of the two main FFT libraries on HPCx: IBM's ESSL/PESSL and FFTW. Both serial and parallel (distributed-memory only) 3D complex-to-complex FFT routines are investigated, and the performance of the two different libraries is investigated.

In general, the ESSL and FFTW serial 3D FFT routines are comparable. For parallel FFTs, the PESSL library is, in general, slightly faster, however, FFTW has better parallel efficiency. FFTW measured plans are extremely expensive to compute and only give a modest improvement in performance over estimated plans.

Some further comments are made about the overall performance of HPCx, and its impact of the use of FFT library routines.

Contents

1 Introduction **1**
1.1 Compilation Details . . . 2

2 FFT Algorithms **2**

3 Serial Behaviour **3**
3.1 FFTW Plans . . . 4
3.2 ESSL's Approach . . . 4
3.3 Performance . . . 6
3.4 Other Matrices . . . 6

4 Parallel Performance **6**
4.1 PESSL . . . 7
4.2 FFTW . . . 7
4.3 Results returned in transformed arrays . . . 8
4.4 Scaling the output of the FFT libraries . . . 8
4.5 Results . . . 8
4.6 Scaling the problem size with the number of processors . . . 11
4.7 FFTW Plans . . . 12
4.8 Other matrices . . . 13

5 Summary **19**
5.1 Timings . . . 19
5.2 Memory usage . . . 20
5.3 Environment variable OMP_NUM_THREADS . . . 20
5.4 Future Work . . . 20

A Code fragments **21**
A.1 ESSL code fragments . . . 21
A.2 Serial 3D FFTW code fragments . . . 22
A.3 PESSL code fragments . . . 23
A.4 Parallel 3D FFTW code fragments . . . 23

B Matix sizes for benchmarks **24**

List of Figures

1 Halo construction for results matrix . . . 3
2 Decomposition of data for parallel FFTs . . . 4
3 Times for FFTW plan construction . . . 5
4 FFT calculation times using varied FFTW plans . . . 5
5 Serial FFT times (non-power-of-2 matrices) . . . 7
6 Speed-up using a global matrix of size 128^3 . . . 9
7 Times for FFT calculations using a global matrix of size 128^3 . . . 9
8 Speed-up using a global matrix of size 256^3 . . . 10
9 Times for FFT calculations using a global matrix of size 256^3 . . . 11
10 Speed-up using a global matrix of size 512^3 . . . 12
11 Times for FFT calculations using a global matrix of size 512^3 . . . 13
12 Times of FFTs using a fixed local size matrix . . . 14
13 Time to compute FFTW plans with a global matrix of size $128 \times 128 \times 128$ . . . 14
14 Speed-up of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 128^3 . . . 15
15 Speedup of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 256^3 . . . 15
16 Speedup of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 512^3 . . . 16
17 Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 128^3 . . . 16
18 Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 256^3 . . . 17
19 Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 512^3 . . . 17
20 Times for FFT calculations using non-power-of-2 matrices and 8 processors . . . 18
21 Times for FFT calculations using non-power-of-2 matrices and 16 processors . . . 18
22 Times for FFT calculations using non-power-of-2 matrices and 64 processors . . . 19

1 Introduction

Fast Fourier Transforms (FFTs) are an important part of many scientific applications. A typical program may perform many FFTs on large datasets for a single production run. Therefore, a high performance service such as HPCx needs to provide efficient methods for performing FFT and associated calculations. IBM provide a number of libraries with HPCx, of which the Engineering and Scientific Subroutine Library (ESSL) for AIX [1] contains FFT routines. However, there are also a number of public domain numerical libraries that include FFT calculations. One such library is the *Fastest Fourier Transform in the West*, or FFTW, [3], which provides routines to compute discrete Fourier transforms.

ESSL is a collection of optimised serial subroutines covering a wide range of mathematic functions, including a subset of BLAS [5] and LAPACK [6], which aims to provide the tuned numerical algorithms typically employed by engineering and scientific applications. ESSL can be called in 32- and 64-bit addressing modes from C, C++, and FORTRAN, although, for the purposes of this report, the C functionality alone was tested. ESSL actually contains two separate libraries:

- The ESSL Symmetric Multi-processing (SMP) Library
- The ESSL Serial Library

ESSL SMP contains a number of multi-threaded routines for distributing work in a shared-memory system. HPCx is currently composed of a large number of 8-processor shared-memory Logical PARtitions (LPARs), connected via an interconnect and an associated switch hierarchy. This means that, whilst ESSL SMP has potential benefits for small (i.e. jobs that run within a shared-memory partition) or hybrid[1] programs, for standard distributed-memory programs such as MPI, ESSL SMP is not as relevant. Therefore this report does not cover the performance characteristics of ESSL SMP.

As the ESSL serial library provides only provides serial routines, and we are not considering using ESSL SMP, a method for performing parallel FFTs is necessary. The IBM library used for parallel FFTs is PESSL [2] (Parallel ESSL). Again, this has two versions, PESSL and PESSL SMP, and both are available for C, C++, and FORTRAN. However, the PESSL version of the library only contains 32-bit routines, so 64-bit applications must use the PESSL SMP library. Readers must note that the name PESSL SMP is misleading in that it does not equate to a distributed ESSL SMP library. The PESSL SMP library is a purely distributed-memory library, and achieves parallelism through the IBM BLACS[4] (Basic Linear Algebra Communications Subprograms) library.

Aside: this distinction is important for *mixed-mode* programming, where both shared-memory and distributed-memory parallel techniques are used. To use ESSL/PESSL in a mixed-mode program, both PESSL SMP and ESSL SMP must be used together, with ESSL SMP performing the shared-memory parallelisation and PESSL SMP the distributed-memory parallelism.

Like ESSL and PESSL, FFTW has libraries for both serial and parallel calculations and is a self-tuning library[2], meaning that, although it hasn't been designed specifically for HPCx, it should compare favourably to the targeted IBM libraries. On HPCx, FFTW can be called in 32- and 64-bit addressing modes from C, C++, and FORTRAN, although, for the purposes of this re port, the C functionality alone was tested.

FFTW has methods for both shared-memory and distributed-memory parallelisations, with the latter using MPI. For the reasons mentioned previously, the shared-memory routines are not benchmarked in this report.

As well as being self-tuning, thus allowing it to be highly portable without degrading performance, it has been optimised for a large set of FFT sizes: sizes that can be factored into the primes 2, 3, 5, and 7, and multiples of 11 and 13. Otherwise it uses a slower, general-purpose, FFT routine. This compares favourable to ESSL, which can only perform transforms on matrices with sizes corresponding

[1] A hybrid program refers to a program that exploits both distributed- and shared-memory parallelisation (e.g. a code that employs both MPI and OpenMP)

[2] In this situation, self-tuning refers to an adaptive software approach where the library tunes the way it performs FFTs to the specific hardware architecture of the machine it is compile on (see [7] for more details)

to the following formula: $n = (2^h)(3^i)(5^j)(7^k)(11^m)$ for $n \leq 37748736$, where $h = 1, 2, ..., 25$, $i = 0, 1, 2$, and $j, k, m = 0, 1$. In the case when ESSL cannot compute the FFT for the specified size, the matrix is generalised to the next largest size that ESSL can compute the FFT for.

The FFT routines benchmarked in this report were 3D complex-to-complex. Note that both libraries limit the parallel data decomposition to one-dimensional, or *slab*, decompositions only. For a benchmark written in C this translates to only the first dimension being decomposed, meaning a 128^3 matrix distributed over 4 processors would give individual local matrices of $32 \times 128 \times 128$.

1.1 Compilation Details

Versions 3.3 of the ESSL libraries, and 2.3 of the PESSL libraries were used for this report. The FFTW library employed was version 2.1.5.

NB: The ESSL/PESSL libraries cannot be statically linked (see [1]/[2])

All codes were compiled using makefiles. The ESSL code was effectively compiled using:

```
xlc_r serial3dessl.c -q64 -O3 -lessl -lm
```

The serial FFTW code was effectively compiled using:

```
xlc_r serial3dfftw.c -q64 -O3
-I/opt/freeware/include -L/opt/freeware/lib -ldfftw64 -lm
```

The PESSL code was effectively compiled using:

```
mpcc_r parallel3dessl.c -q64 -O3
-qmaxmem=-1 -bmaxdata:0x70000000 -lpesslsmp -lblacssmp -lessl -lm
```

The parallel FFTW code was effectively compiled using:

```
mpcc_r parallel3dfftw.c -q64 -O3
-qmaxmem=-1 -bmaxdata:0x70000000 -I/opt/freeware/include -L/opt/freeware/lib
-ldfftw_mpi64 -ldfftw64 -lm
```

2 FFT Algorithms

For this study the FFT routines of the respective libraries were tested using the following algorithm:

- Randomly initialise a complex array, b, say
- Precompute trigonometric functions given in eqn. (2) below, and store in array f, say
- Perform a forward FFT on b
- Multiply the result by array f
- Perform a reverse FFT and store in the results array r, say
- Verify the answer

The verification step in the algorithm is based on eqn. (1), which states that if the calculation has been performed correctly the value each element in the initial matrix (b) should be equal to the specified combination of elements of the results matrix, r. The verification equation is given as

$$b_{i,j,k} = r_{i+1,j,k} + r_{i-1,j,k} + r_{i,j+1,k} + r_{i,j-1,k} + r_{i,j,k+1} + r_{i,j,k-1} - 6r_{i,j,k}, \tag{1}$$

where i, j, and k denote the position of the element in the matrix.

However, this relationship only holds as long as the correct sequence of trigonometric functions is used for f, where the value of each element in f is calculated using

$$\sin\left(\frac{i \times \pi}{nx}\right)^2 + \sin\left(\frac{j \times \pi}{ny}\right)^2 + \sin\left(\frac{k \times \pi}{nz}\right)^2. \tag{2}$$

As the verification step must be performed for each element in the original matrix, a *halo* (or *picture frame*) of elements must be constructed around the results matrix to allow the 6-point stencil to be

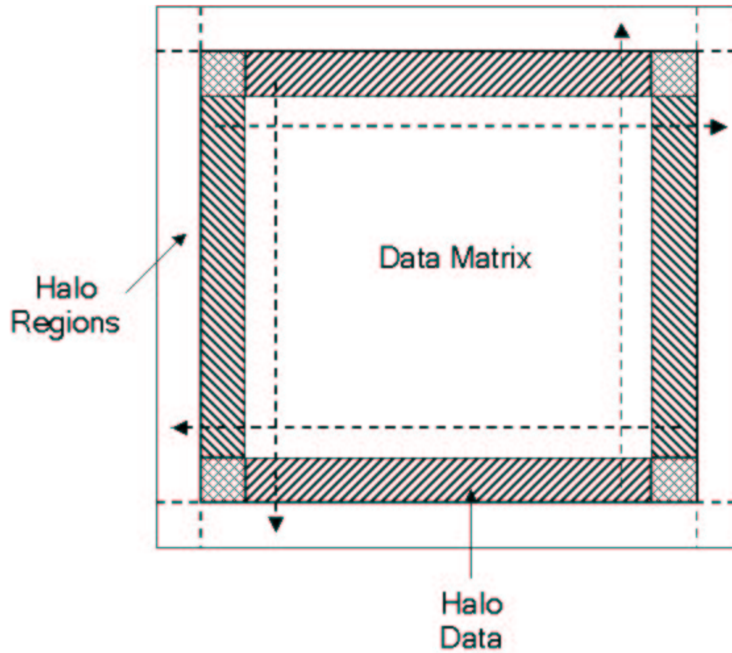

Figure 1: Halo construction for results matrix

applied to elements at the edge of the matrix (i.e. extra elements are needed to cope with using element $r_{i-1,j,k}$ when $i = 0$ etc...). For this situation a single element halo is used, and the results matrix is given *periodic* boundary conditions. This means that the data in the results matrix is wrapped around the matrix boundaries, with the halo for the bottom boundary of a matrix dimension containing the data from the top boundary of that dimension in the matrix (see figure 1). Halos are constructed after the results matrix has been calculated, but before the verification step takes place.

Each FFT is performed once, where the time to perform the FFT routines is measured, along with the time to calculate the trigonometric constants, determine the best routines to use, etc, where applicable.

For the parallel benchmarks, the algorithm is adapted so that the b and f arrays are smaller than the global array - which never actually exists (see figure 2). Halos are also computed and swapped between processors for the final verification step. The global variables used to track the programs progress also need to be calculated and communicated between processors. All these communications are performed using MPI.

3 Serial Behaviour

The starting point for the benchmarking of FFTs was the serial case. The jobs where run on a dedicated LPAR to ensure that they did not share CPUs with other jobs. The libraries were tested using 3D arrays of the following sizes: $(8 \times 8 \times 8)$, $(16 \times 16 \times 16)$, $(32 \times 32 \times 32)$, $(64 \times 64 \times 64)$ and $(128 \times 128 \times 128)$.

The physical memory limit for each processor ($\sim$970 MB)[3] meant that a matrix of 256^3 could not be tested. The performance of the forward and backward Fourier transformations was measured, in both 32- and 64-bit modes for both ESSL and FFTW, and it was found that there is little performance difference between the two modes. Since the 64-bit libraries are needed for large data sets it is advisable to always use them. Each code was run 5 times for every benchmark, to allow an average time to be calculated.

[3]Whilst this is an issue at the time of this report, the memory limit has been recently increased

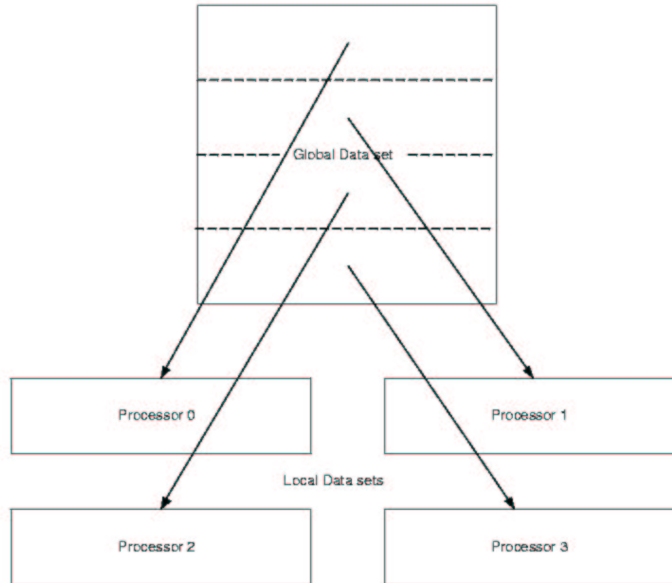

Figure 2: Decomposition of data for parallel FFTs

3.1 FFTW Plans

FFTW routines can be optimised in a number of ways, the easiest being to alter the *plan* creation method. Plans are the data structures FFTW uses to compute the FFT and include the trigonometric constants needed for the computation of a given size of matrix, and a list of the functions (or *codelets*) that FFTW should use to compute the FFT. The nature of FFT computation means that these trigonometric constants and codelets used for a FFT calculation are dependent purely on the size of the matrix (not the data within it), and so can be reused for FFT calculations on any matrix of the same size. For the benchmarks in this report, *estimated* plans were used (the default). An estimated plan means that the FFTW library guesses what codelets would be most efficient for the specified matrix:- this plan construction takes very little time. However, there is another method of plan construction: *measured* plans. This involves computing a number of FFTs (using different codelets) and actually measuring their execution time to find the optimal routine for that problem, meaning that the plan construction may well take longer, but could potentially speed-up the FFT calculations.

The performance of an FFTW routine, using both estimated and measured plans, was tested using a range of matrix sizes. The results are shown in figures 3 and 4. From these graphs it can be seen that measured plans take a relatively long time to construct yet they give no discernible performance benefit when computing the actual FFT. Therefore, estimated plans will be used to perform the remaining serial benchmarks.

3.2 ESSL's Approach

All FFT libraries have methods similar to FFTW's plans for reusing the precomputed data that is associated with performing FFTs, and ESSL is no exception to this. The 1- and 2-D ESSL FFT computation routines must be called twice when first used, once to initialise the working storage used for the FFT computation, and then to compute the FFT itself. As with FFTW, once this initialisation has occurred, the working storage can be reused as many times as is needs (assuming the same size matrix is being used). However, for 3-D transforms, this initialisation is not necessary. The ESSL documentation [1]

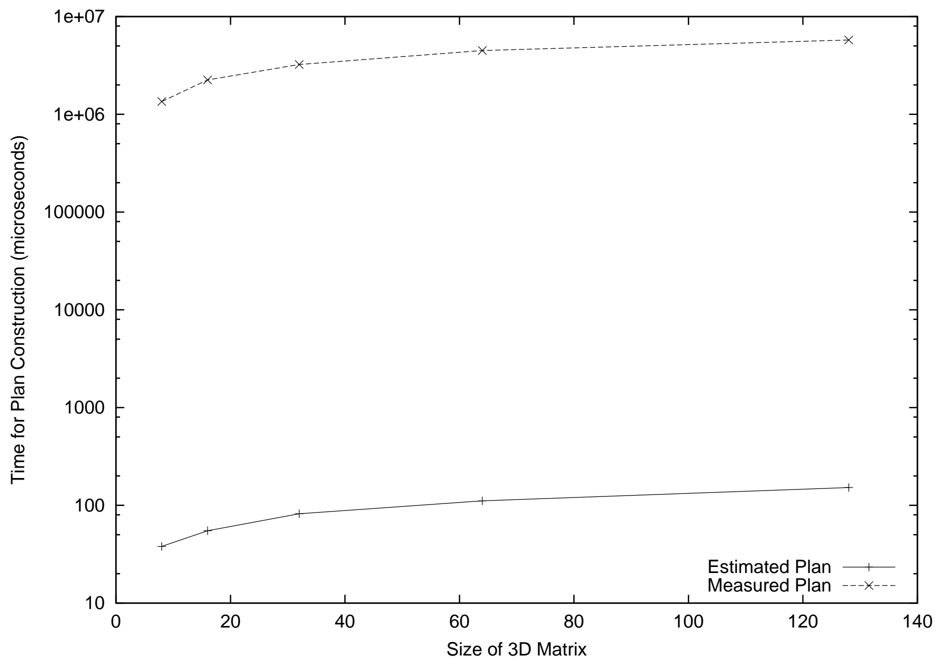

Figure 3: Times for FFTW plan construction

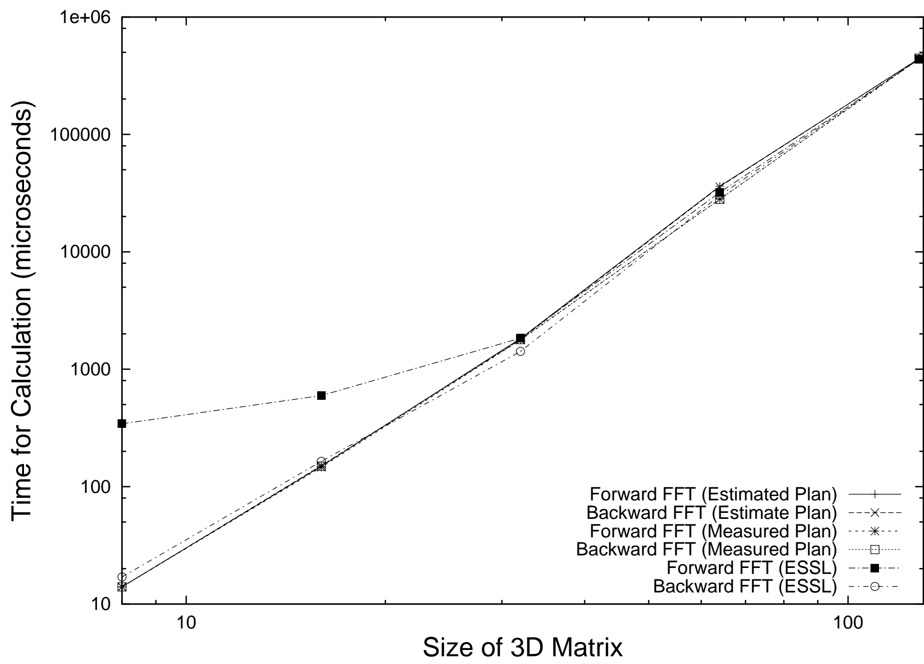

Figure 4: FFT calculation times using varied FFTW plans

states that the *initialisation phase* [...] *is a relatively small part of the total computation, so it is performed on each invocation*. Therefore, for these benchmarks, when the ESSL FFT computation routine is called it is performing both the initialisation and calculation, whereas when the FFTW routine only performs the calculation.

3.3 Performance

As can be seen from figure 4, for large arrays the IBM ESSL routines produces a slightly better performance than the FFTW routines. However, for small matrices the reverse is true: FFTW routines outperform ESSL. It is also interesting to note that the performance of the forward and backward transformations is almost identical for FFTW, whereas the ESSL routines have difference performance characteristics with ESSL forward FFTs performing very poorly for small data size.

3.4 Other Matrices

The benchmarks performed so far have used square matrices, where the length of the FFTs employed have been a power-of-two. However, real scientific applications may use non-square and/or non-power-of-two matrices. FFTW and ESSL were tested upon a large set of matrices (see Appendix B), and the results from these benchmarks are shown in figure 5. This graph contains the ESSL and FFTW forward and backward FFT times, along with the time for FFTW's forward FFT combined with it plan's calculation time. This is included because the ESSL FFT routines reinitialise their internal storage every time the FFT routine is called (i.e. the equivalent for recalculating plans for each FFT). Therefore, showing the FFTW FFT calculation time combined with the plan calculation time allows for fairer comparisons between the two libraries performance.

For large matrices, there appears to be only two data sets displayed in figure 5. This is because all data points associated with the FFTW are similar and, likewise, all the data points of ESSL are similar.

However, it can be seen from the graph that even though ESSL performs initialisation every time it is called, it still outperforms FFTW for matrices with over ~ 20000 elements (i.e. $24 \times 28 \times 28$). For instance, in the case where $N = 110^3$, ESSL is 1.75 and 1.68 times faster than the forward and backward FFTs of FFTW, respectively. In fact, the backward FFT for ESSL always gives performance comparable with FFTW. Only the forward ESSL FFT shows comparatively poor performance for small matrices.

4 Parallel Performance

The main area of importance for HPCx, and the main focus of this work, is parallel performance. Some HPCx users will construct their own parallel multi-dimensional FFTs using the serial FFT routines of FFTW and ESSL, however, other users may use the parallel FFT libraries of FFTW and PESSL.

PESSL uses IBM's BLACS[4] (Basic Linear Algebra Communications Subprograms) to perform data transforms, whereas FFTW uses MPI. Both libraries can only decompose data in one dimension. As the benchmarking code is written in C, this means it is the 1st dimension of the arrays that are decomposed.

To perform the parallel benchmarks, the serial program was extended to include the parallel FFT directives, data distribution and initialisation functions. Data arrays were extended to include halo regions, and code added to perform *halo swaps* (see Section 2).

As noted in the serial benchmarking, the largest square matrix that an individual processor can operate upon (with this particular benchmarking program) is 128^3. This limit applies to parallel programs as well as serial programs (i.e. the largest global data set that 8 processor can operate upon would be $1024 \times 128 \times 128$). Table 1 contains the processor numbers and matrices sizes used for this report.

As well as testing the performance of the parallel FFT routines for fixed size matrices (as the number of processors used is increased), we also tested the performance of the routines when the matrix size

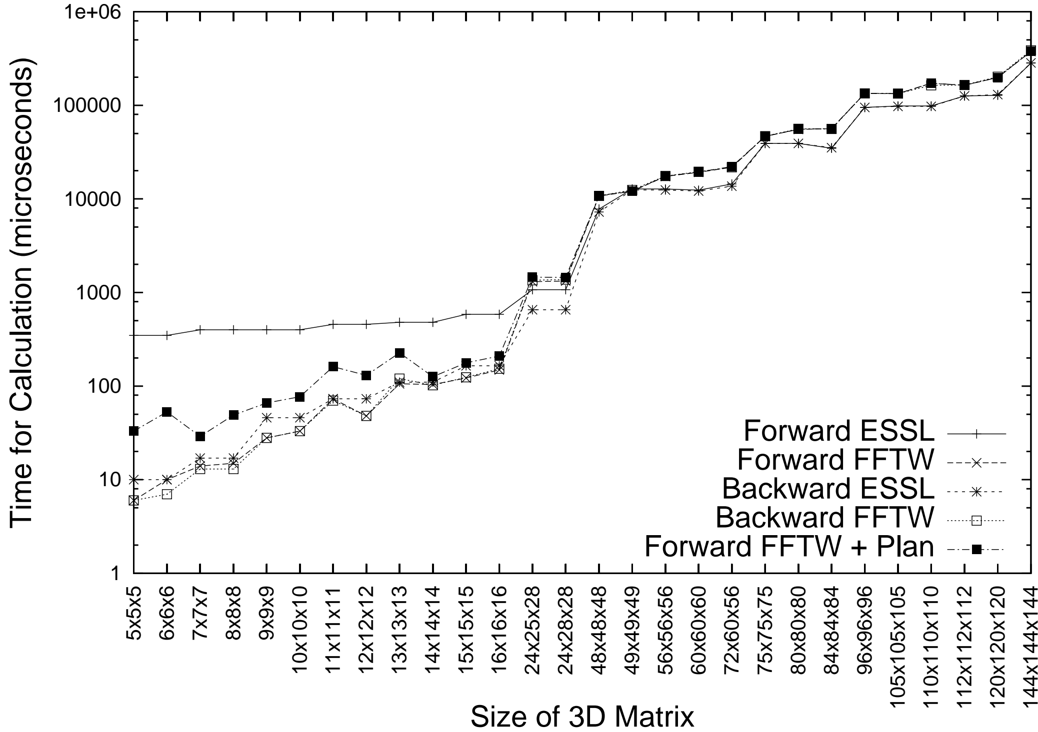

Figure 5: Serial FFT times (non-power-of-2 matrices)

Matrix Size	Processor Range
Data Set A: $128 \times 128 \times 128$	2, 4, 8, 16, 32, 64, 128
Data Set B: $256 \times 256 \times 256$	4, 8, 16, 32, 64, 128, 256
Data Set C: $512 \times 512 \times 512$	32, 64, 128, 256, 512

Table 1: Processor numbers for each benchmark matrix

is increased. This scaling exercise involved increasing the size of the global matrix proportional to the number of processors used, to ensure that each processors' local matrix was the same size (see Table 2).

4.1 PESSL

As PESSL uses the BLACS library to perform communications, the first routines initialise the BLACS environment. This is done by calling the `blacs_get` routine to obtain the BLACS *default system context*, followed by `blacs_gridinit` to construct a basic processor grid. Because of the one dimensional decomposition restriction, we initialised the processor grid to be $1 \times q$ (where q is the number of processors used).

Once these initialisation steps have been performed the FFT routines can be called as in the serial program. The test program also contained some MPI communication to facilitate error checking and calculation of global values. The final call for the program should be to the `blacs_exit` routine, which releases the BLACS *context* and deallocates all associated memory.

4.2 FFTW

FFTW performs communication through MPI, so the first step of the program is to initialise MPI. The data decomposition and processor grid for the FFT calculation is constructed by the FFTW library,

Local Matrix Size: $128 \times 128 \times 128$	
Processor Number	**Global Matrix Size**
2	$256 \times 128 \times 128$
4	$512 \times 128 \times 128$
8	$1024 \times 128 \times 128$
16	$2048 \times 128 \times 128$
32	$4096 \times 128 \times 128$
64	$8192 \times 128 \times 128$
128	$16384 \times 128 \times 128$

Table 2: Global matrix sizes for the *same size* benchmarks

facilitating the distribution of input data using MPI. Once MPI has been initialised, the FFTW *plans* can be constructed. These are identical to the serial FFTW plans, with the exception of the addition of the *MPI communicator* to the argument list. After the plans have been created for the forward and backward FFTs, the memory for input matrix can be allocated and the FFT calculations performed.

4.3 Results returned in transformed arrays

Both FFTW and PESSL allows the results of the FFT calculation to be returned in *transpose* order to improve the calculation performance.

For FFTW, this is a partial transformation, i.e. the first two dimensions of the output matrix are the reverse of the input matrix (i.e. (x,y,z) becomes (y,x,z), say). For PESSL, on the other hand, this means a full transformation (i.e. (x,y,z) becomes (z,y,x)).

Returning data in this transformed form reduces the amount of communications required for both the forward and backward parallel FFT calculations. However, since FFTW and PESSL do not return the same transformed arrays, both libraries employed to return the array in normal order. Explicitly, FFTW was run in `NORMAL` mode whilst PESSL has `IP(0)=0`, `IP(1)=1`, and `IP(19-22)=0`.

4.4 Scaling the output of the FFT libraries

PESSL scales the transformed data, whereas FFTW, like most FFT libraries, leaves the scaling of the data as an exercise for the programmer.

4.5 Results

The speed-up of the parallel routines is shown in figures 6, 8, and 10. For the smallest data set (data set A) PESSL and FFTW have comparable performance, with PESSL showing super linear speed-up for 4 processors, however, both libraries exhibit poor efficiency for more than 8 processors. This is because the dataset is not large enough to effectively use large number of processors.

The results also highlight the difference in performance encountered when using processors in more than one LPAR. Whilst the speed-up is good up to 8 processors (within one LPAR), once the program is tested across LPARs then there is a significant drop in performance. This highlights the fact that the performance of this code is dominated by communication, with good performance when using the fast *shared-memory* communication within a single LPAR but poor performance when communication is performed by sending messages across the network, between LPARs.

The time to solution performance of PESSL and FFTW is also worth noting, with PESSL generally computing FFTs quicker than FFTW (shown in figure 7) although both libraries have comparable times. As with the speedup results, FFTW performs considerably quicker for large numbers of processors.

Data set B (figures 8, and 9) shows a more pronounced difference compared with the previous scaling findings, with FFTW showing considerably better scaling than PESSL, although both libraries

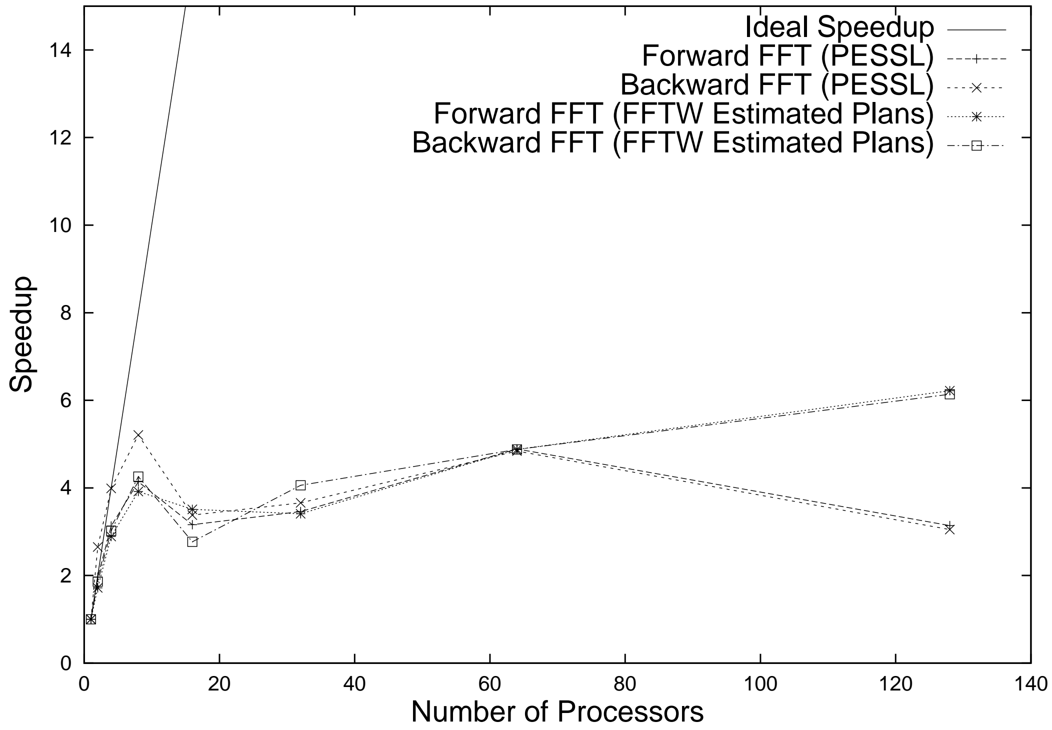

Figure 6: Speed-up using a global matrix of size 128^3

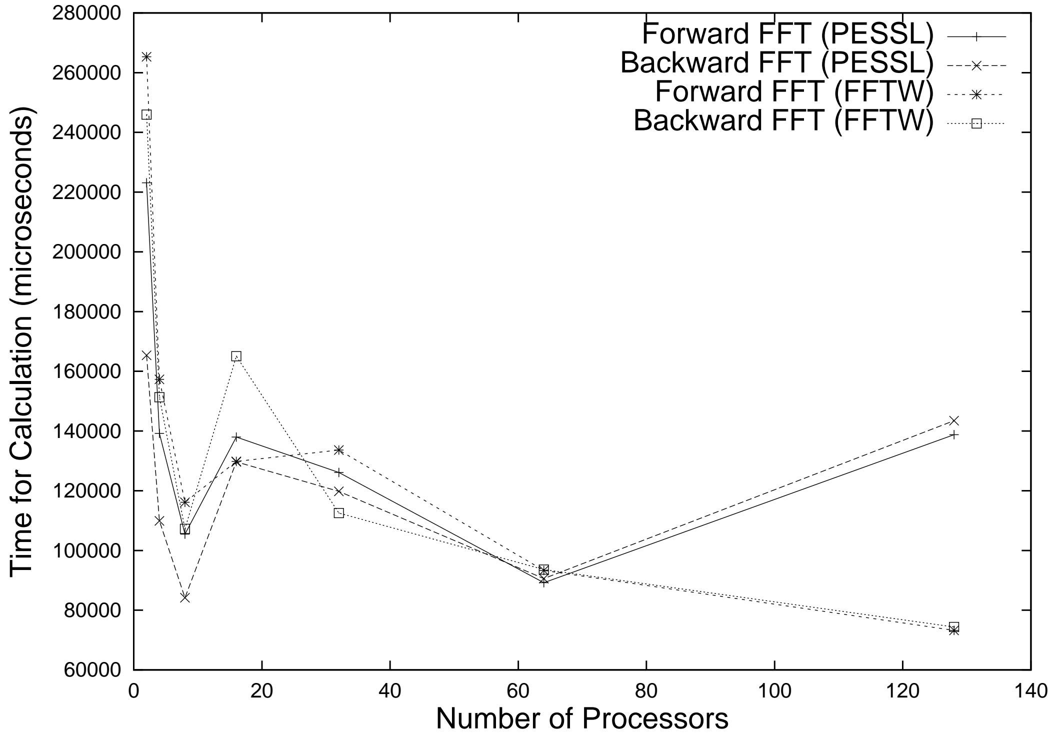

Figure 7: Times for FFT calculations using a global matrix of size 128^3

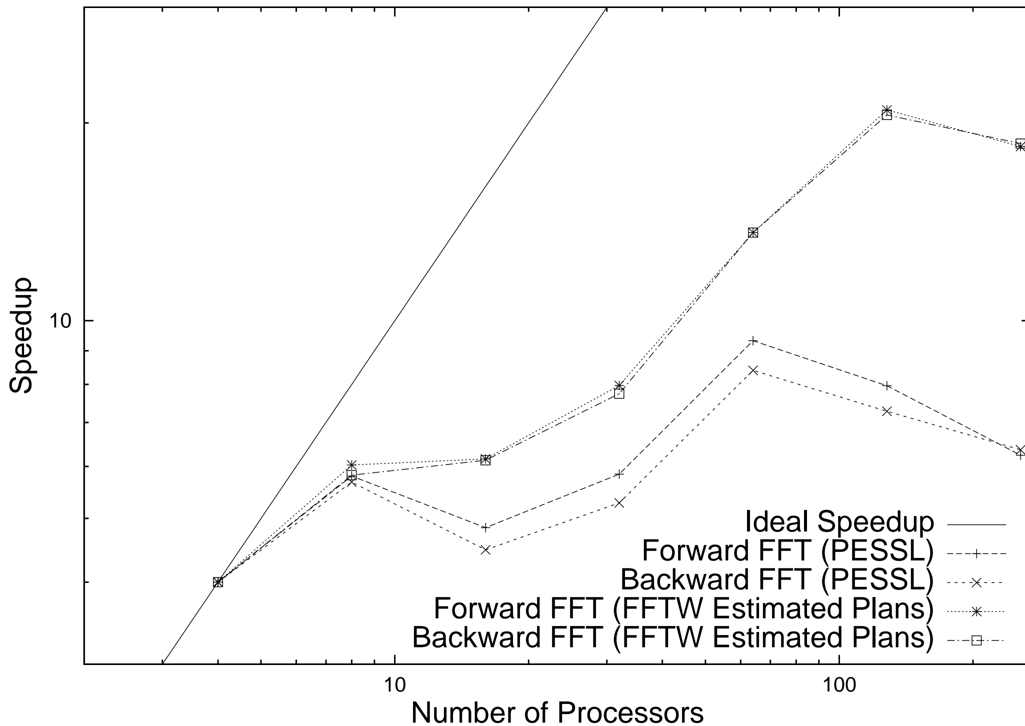

Figure 8: Speed-up using a global matrix of size 256^3

exhibit poor scaling. Given the memory restrictions, the smallest number of processors this data set could be calculated upon was 4, therefore, all speed-up values have be calculated using the 4 processor results. As with the previous data set, PESSL performs quicker FFT calculations than FFTW for small numbers of processors, with the reverse being true when larger numbers of processors are used.

For data set B, there is a less pronounced performance difference between jobs that are within one LPAR and those that span LPARs (shown in figure 8). This is a characteristic of the larger data set, which can make more efficient use of larger numbers of processors. However, even the larger data set does not mask the performance difference between communications within an LPAR and those across the switch.

For the largest data set tested (data set C: 512^3), FFTW exhibits good scaling throughout the range of processors (see figures 10, and 11), showing near perfect speedup for 256 processors, and superlinear speedup for 64 processors. It is a large data set and is, therefore, able to efficiently exploit large numbers of processors. However, the performance still degrades when moving from 256 to 512 processors. This is probably because even this data set isn't large enough to be able to efficiently use 512 processors. The size of the data set means that it cannot be tested on less than 32 processors (due to memory restrictions), so as with the previous benchmarks, the results have been scaled with reference to the results from the smallest number of processors.

The forward and backward PESSL routines have similar speed-up curves. The timing data display in figure 11, from the forward and backward FFT routines, for both FFTW and PESSL, are also very similar, with the backward FFT performing the calculation slightly quicker in both cases. For this data set, PESSL is consistently quicker than FFTW (see figure 11).

Whilst a large number of tests with a range of data sizes and processor numbers were benchmarked, there is scope for more testing using large numbers of processors (> 512). One of the main aims for HPCx is to *terascale* codes, that is, ensuring a significant fraction of user jobs use hundreds of processors on HPCx. Therefore, it would be useful to know whether FFTW and PESSL can cope with

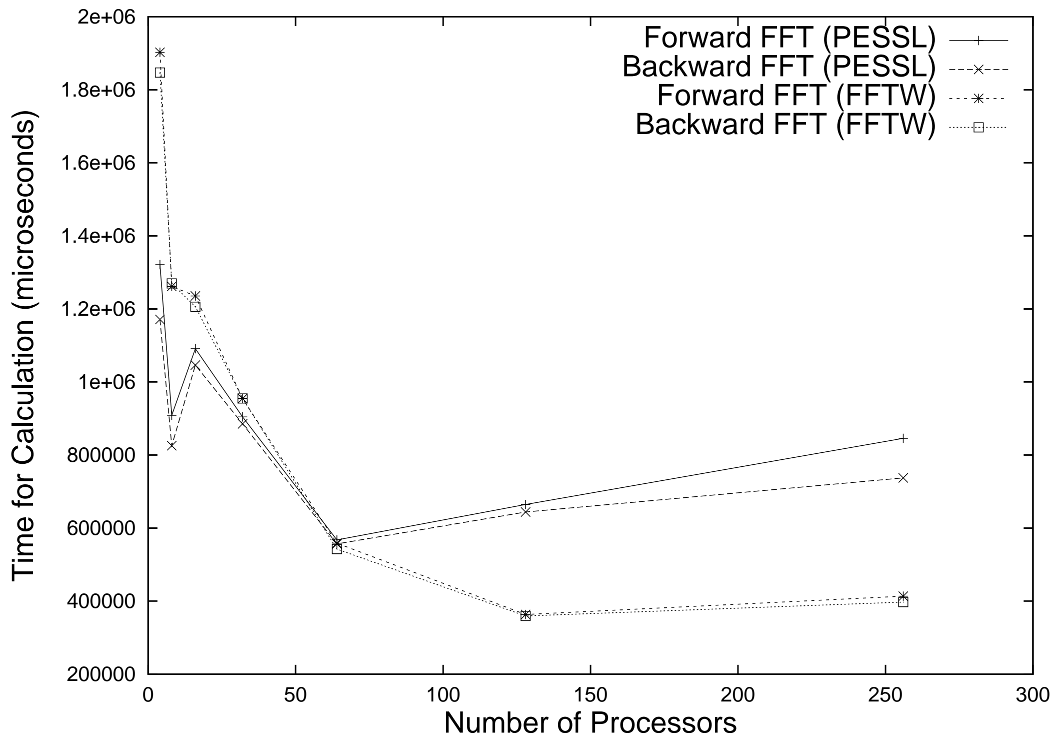

Figure 9: Times for FFT calculations using a global matrix of size 256^3

512 and 1024 processor jobs using extremely large data sets. The performance of 512 processors in the benchmarks performed for this report was poor and further work is needed to ascertain whether this was caused by the data set not being large enough, or whether the libraries cannot scale to these numbers of processors.

4.6 Scaling the problem size with the number of processors

The other type of benchmark performed involved increasing the size of the global matrices as the number of processors used was increased. This ensures that each processor local data set size is always the same. Both PESSL and FFTW were tested, with the results shown in figure 12. Unlike the previous graphs in this section (which contained speed-up results), this graph shows the *time to compute* the FFT vs *Number of Processors*. For these tests, PESSL significantly outperforms FFTW, especially for large numbers of processors (and therefore larger global matrix sizes).

FFTW performs in a more conventional manner, with a steady increase in the computation time as the number of processors increases. It also exhibits a large performance drop when it moves from using only one LPAR, to using two LPARs. Since FFTW uses MPI to perform communications, the performance of FFTW could be improved by using optimised versions of the MPI library, i.e. one which performs different communication methods/patterns for within an LPAR to between LPARs.

The difference in performance between the two libraries could be due to the nature of the data set used for the benchmarking. It is unusual to use global matrices that are much bigger in one dimension than the others, and if FFTW is optimised for square matrices it may perform badly for this kind of data set. However, it could be that PESSL FFTs involve less communications or handle the communications more efficiently. FFTW and PESSL use different libraries (MPI and BLACS, respectively) for communications, and it could be that the BLACS library has be optimised for this kind of applications. More extensive testing, such as testing a range of matrices or benchmarking the performance of MPI

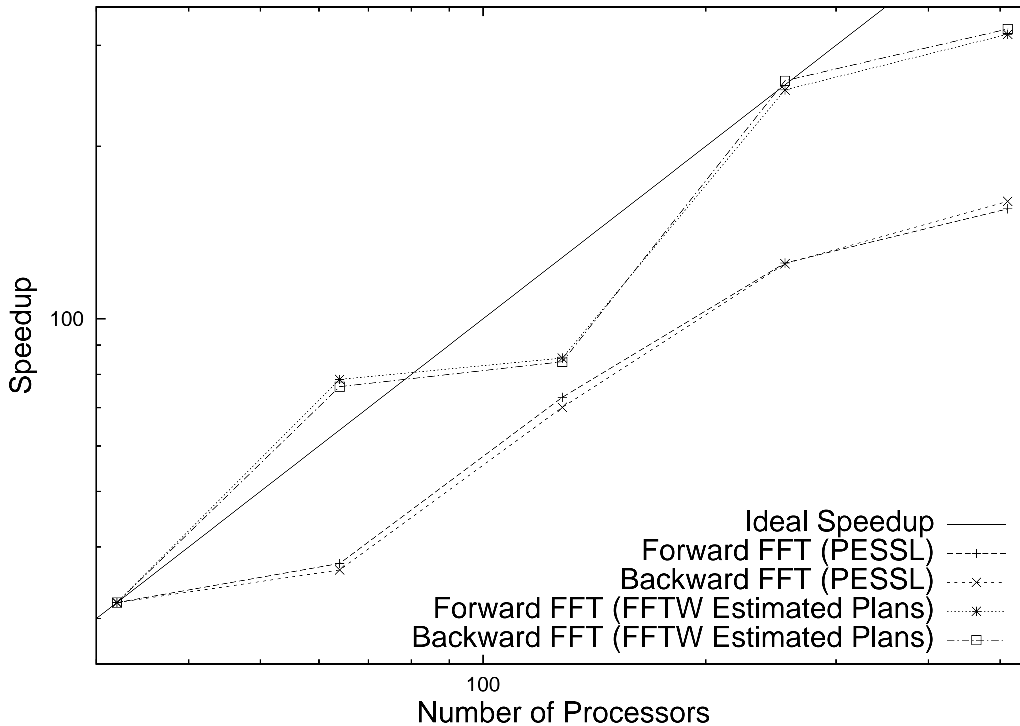

Figure 10: Speed-up using a global matrix of size 512^3

and BLACS, would be necessary to clarify these performance issues.

4.7 FFTW Plans

The plan creation times were recorded for the 128^3 and *same size* benchmarks, and are shown in figure 13. The most obvious feature of the plan creation times is that measured plans take much longer than estimated plans to construct. Indeed, it takes around one thousand times longer to calculate a measured plan (this figure is much larger for very small matrices). The time to calculate the measured plans also increase as the global data set increases.

From the graph it can be seen that estimated plan calculation times also depends on the number of processors used. Also, the estimated case is not significantly affected by the increase in the global data set size.

As well as measuring the plan creation times, the FFT calculations were timed using measured plans to see whether measured plans actually produced more efficient FFT calculation (for this system and problem size). The speed-up results are shown in figures 14, 15, and 16. For data set A, employing measured plans gives no performance benefit, and in fact dramatically reduces the speed-up for large numbers of processors. However, when the global matrix size is increased the measured plans begin to give performance benefits. In fact, the performance characteristics are the reverse of the smallest data set, with it giving massive speed-up increases for large processor numbers (as well as superlinear speed-up for small numbers of processors). This trend is further illustrated when data set C (the largest data set) is tested. Here the measured plans give superlinear speed-up for 128 and 256 processors.

This performance benefit for large numbers of processors means that for large programs, with long production runs, using measured plans could be beneficial to the scaling of the overall program. However, as figures 17, 18, and 19 show, whilst there can be a scaling benefit to using measured plans for larger processor sizes, that benefit does not translate into an improved time to solution.

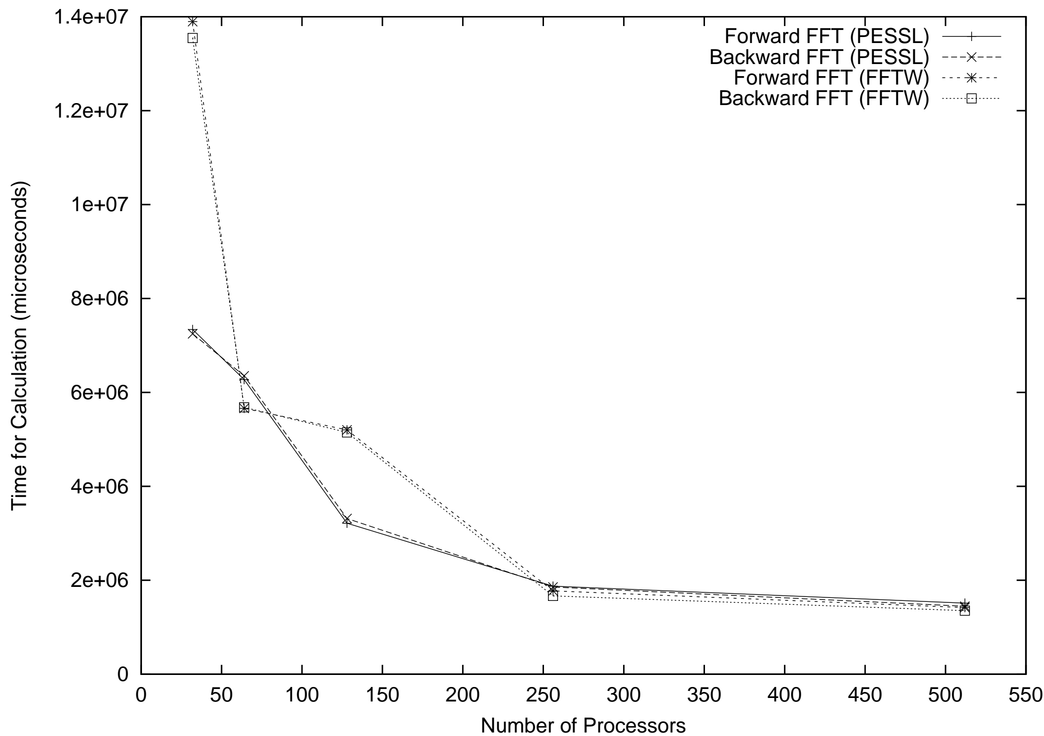

Figure 11: Times for FFT calculations using a global matrix of size 512^3

The calculation performed using measured plans are generally no faster than the estimated plans, and often can be considerably slower. Because the measured plans did not give shorter times to solution for FFT calculations, estimated plans were used for the parallel benchmarks.

4.8 Other matrices

The parallel FFT libraries where also tested on non-power-of-2 matrices, using a subset of the test cases used for the serial benchmarks (the matrices with 1st dimension sizes lower than the number of processors used could not be tested). The benchmarks where performed using 8, 16, and 64 processors, and the matrix sizes used are shown in Appendix B.

The results are shown in figures 20, 21, and 22. From these graphs it can be seen that for 8 processors, PESSL and FFTW have comparable performance regardless of the matrix size. It is also interesting to note that backward FFT calculations are significantly faster than forward calculations for both libraries.

When more processors are used, the performance characteristics of both libraries change. For the 16 processor case, PESSL's performance is noticeably better than FFTW, for both the forward and backward FFT calculations. However, when using 64 processors, the performances change again. FFTW is significantly quicker than PESSL for the smaller matrices when using 64 processors, with the reverse being true for large matrices. This reinforces the previous conclusions that FFTW performs communications more efficiently than PESSL. (The larger the number of processors, the larger the amount of communication need to perform the parallel FFT.) Performance of the FFTW library would be achieve if an optimised version of the MPI library were employed.

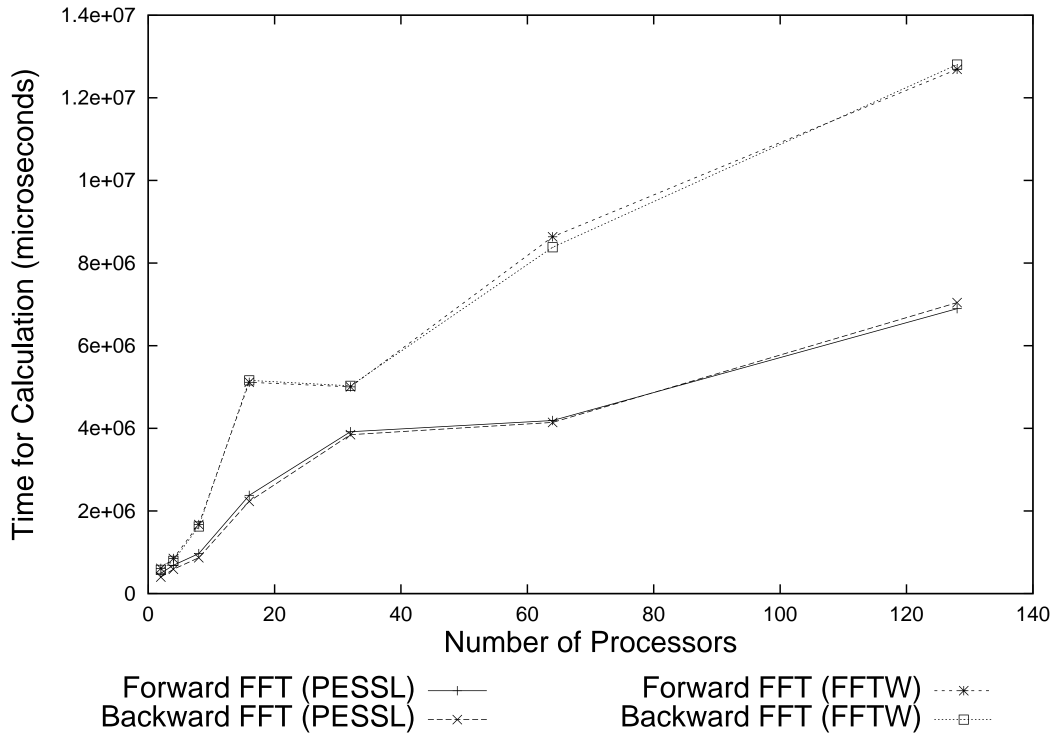

Figure 12: Times of FFTs using a fixed local size matrix

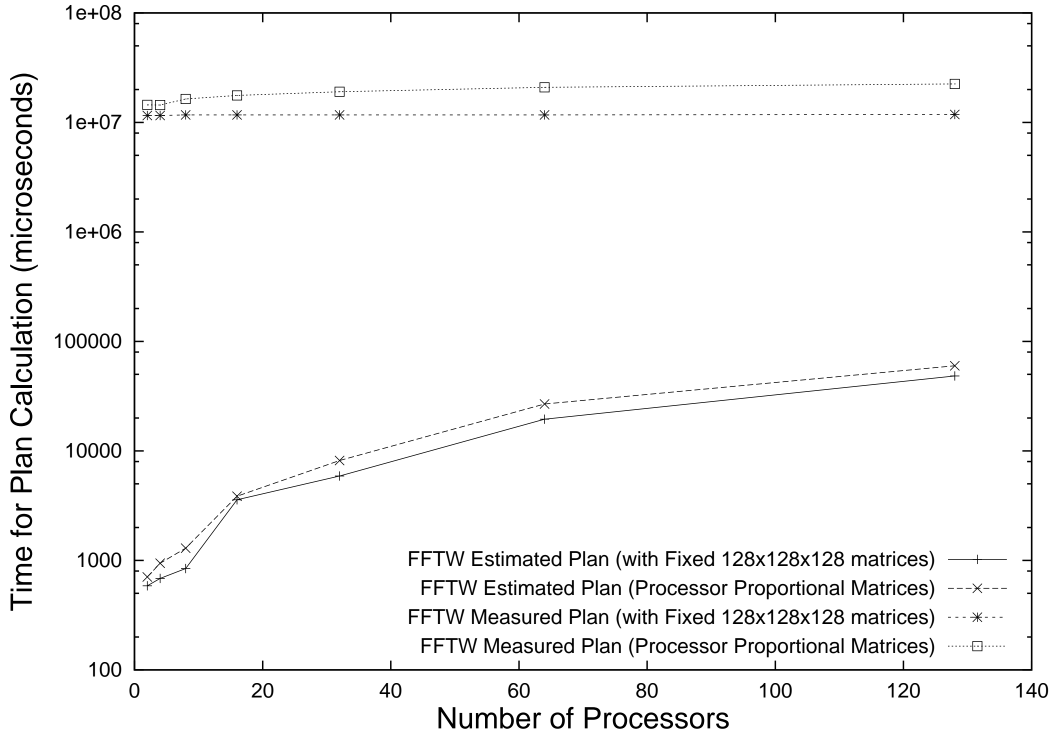

Figure 13: Time to compute FFTW plans with a global matrix of size $128 \times 128 \times 128$

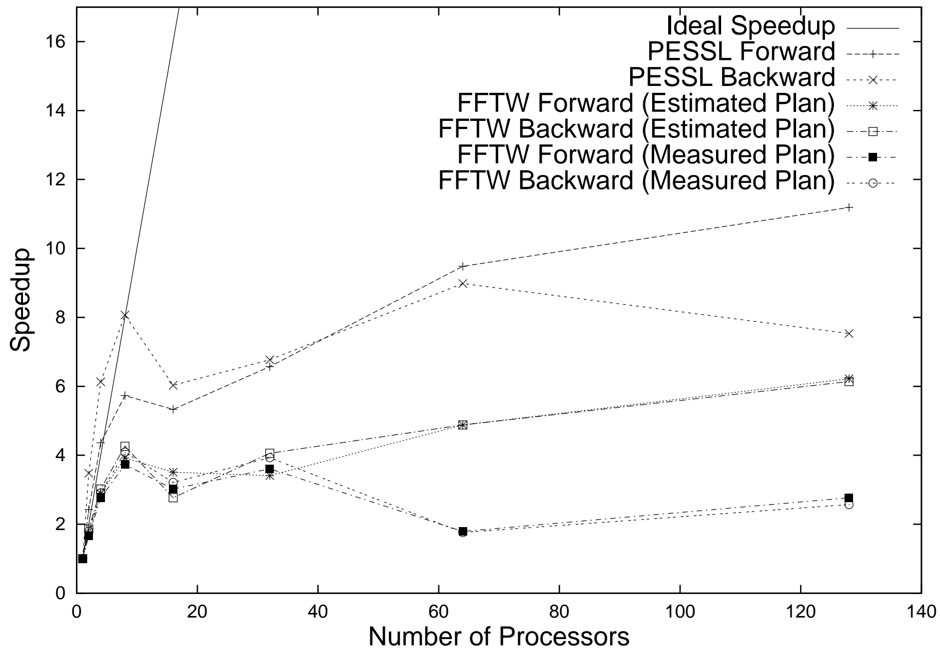

Figure 14: Speed-up of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 128^3

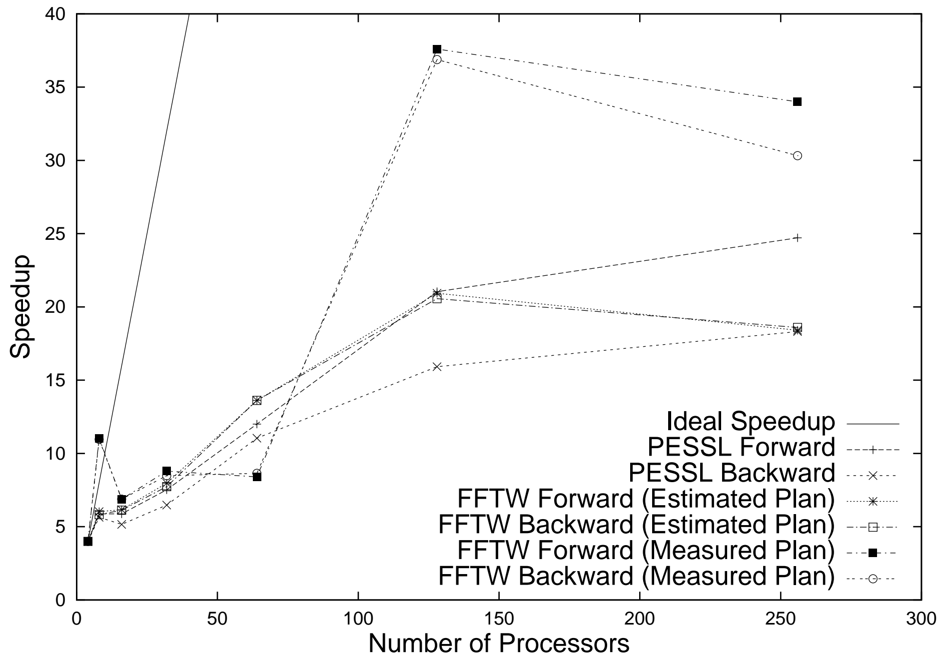

Figure 15: Speedup of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 256^3

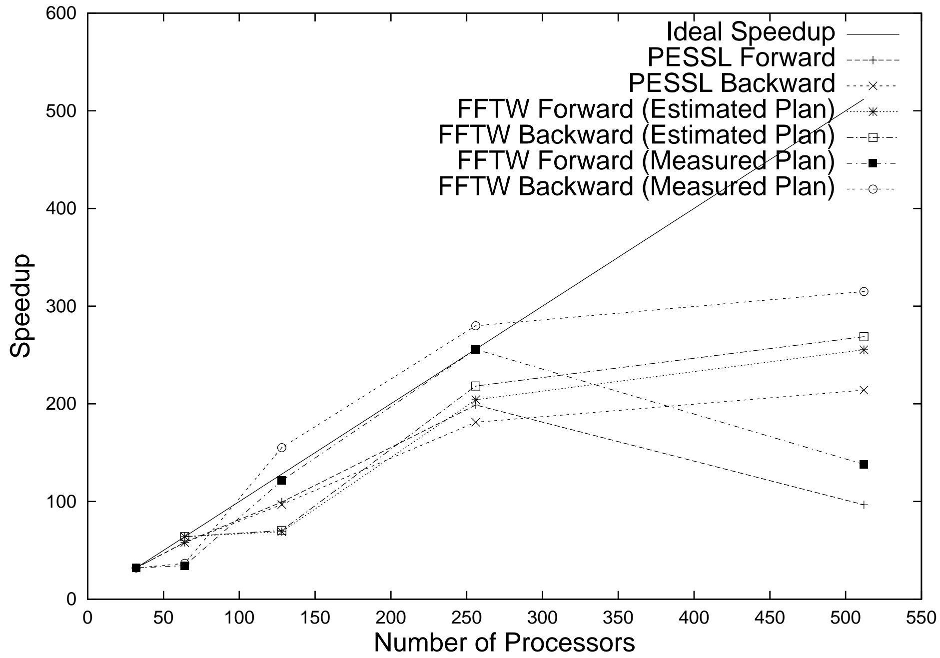

Figure 16: Speedup of FFT calculation (showing FFTW measured and estimated plans) using a global matrix of size 512^3

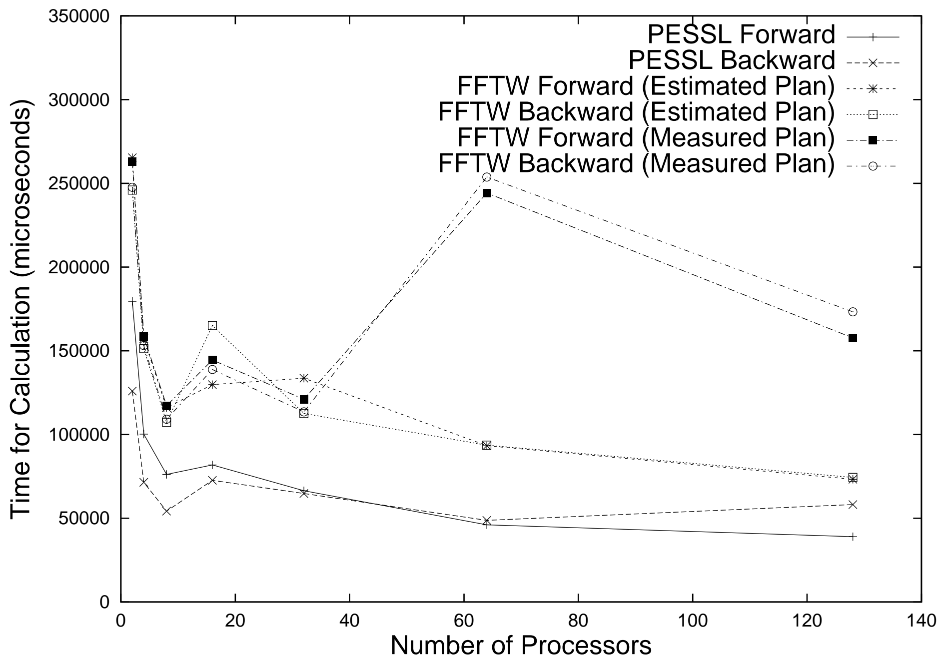

Figure 17: Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 128^3

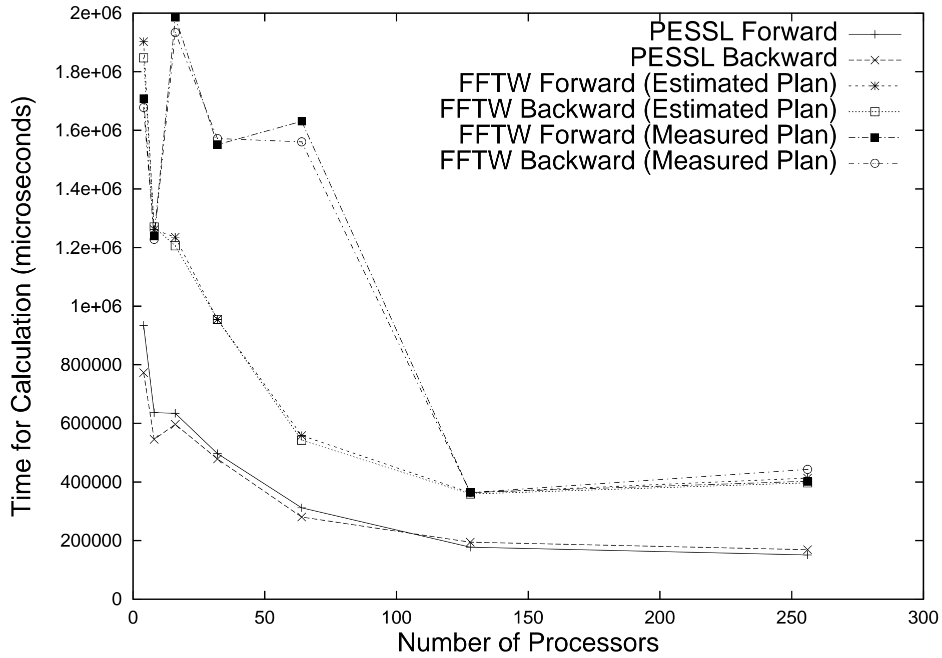

Figure 18: Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 256^3

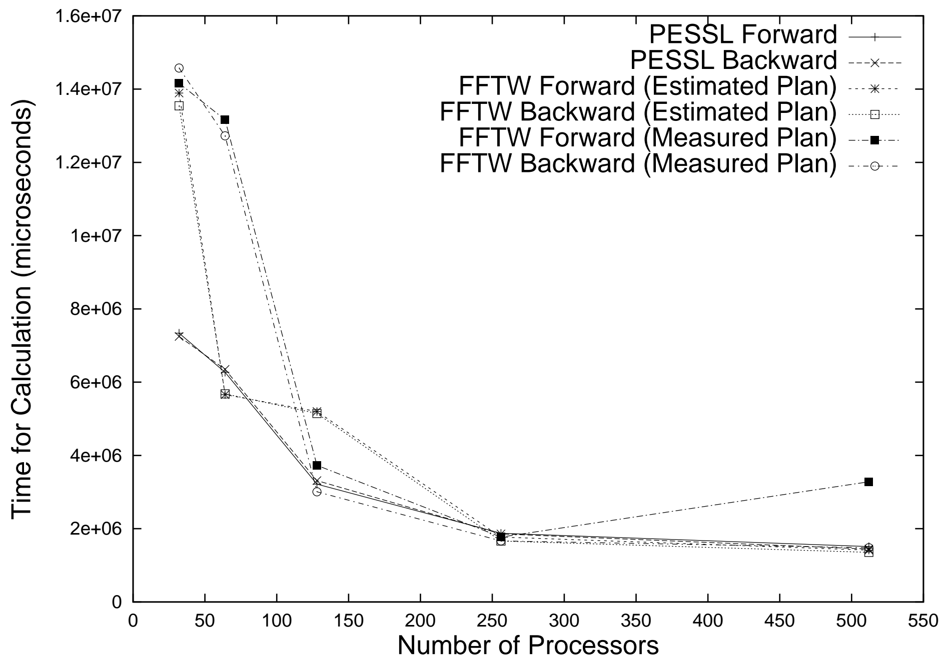

Figure 19: Times for FFT calculation for (showing FFTW measured and estimated plans) using a global matrix of size 512^3

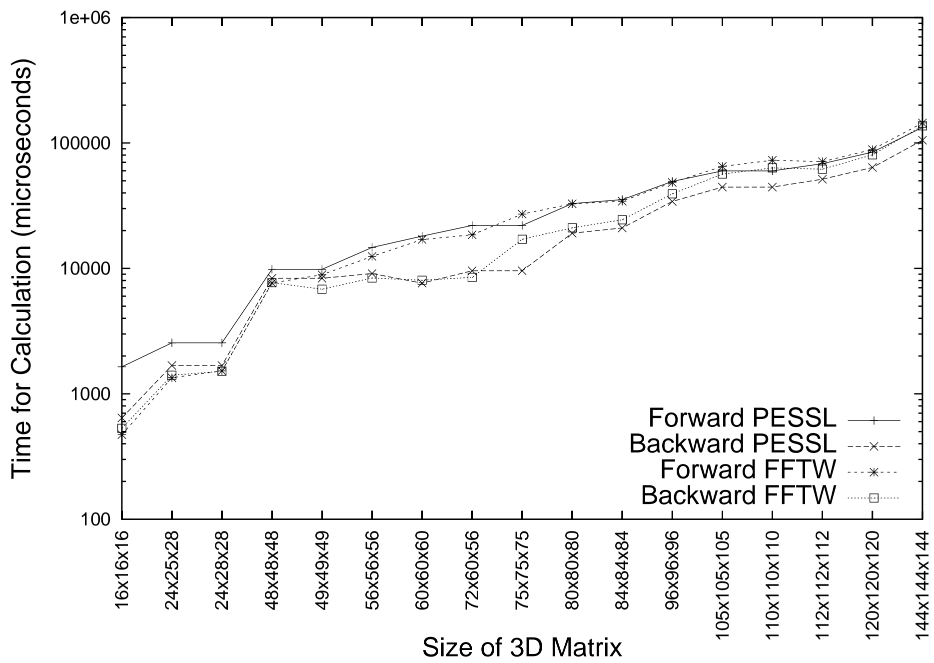

Figure 20: Times for FFT calculations using non-power-of-2 matrices and 8 processors

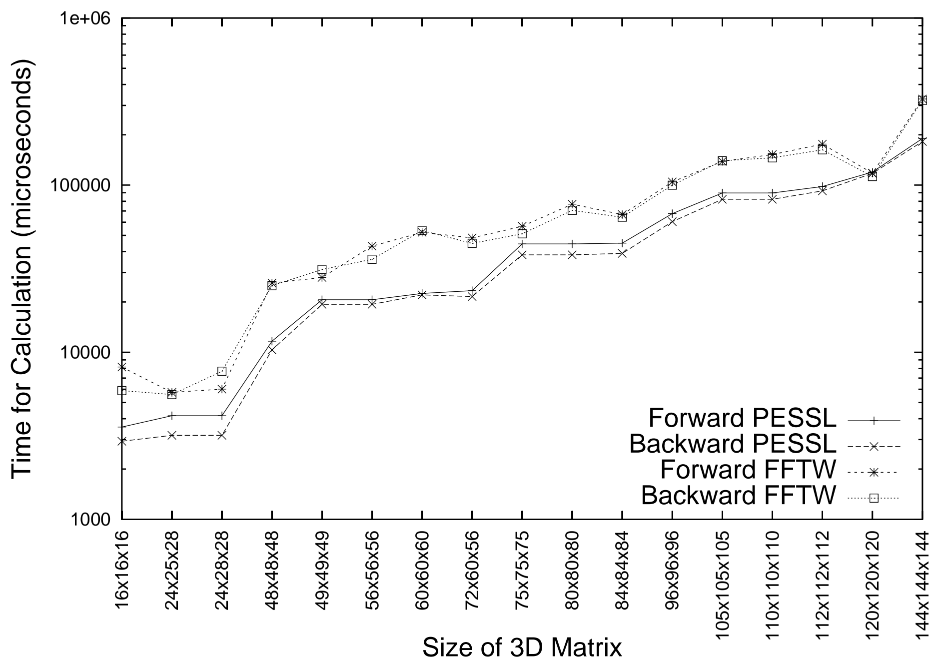

Figure 21: Times for FFT calculations using non-power-of-2 matrices and 16 processors

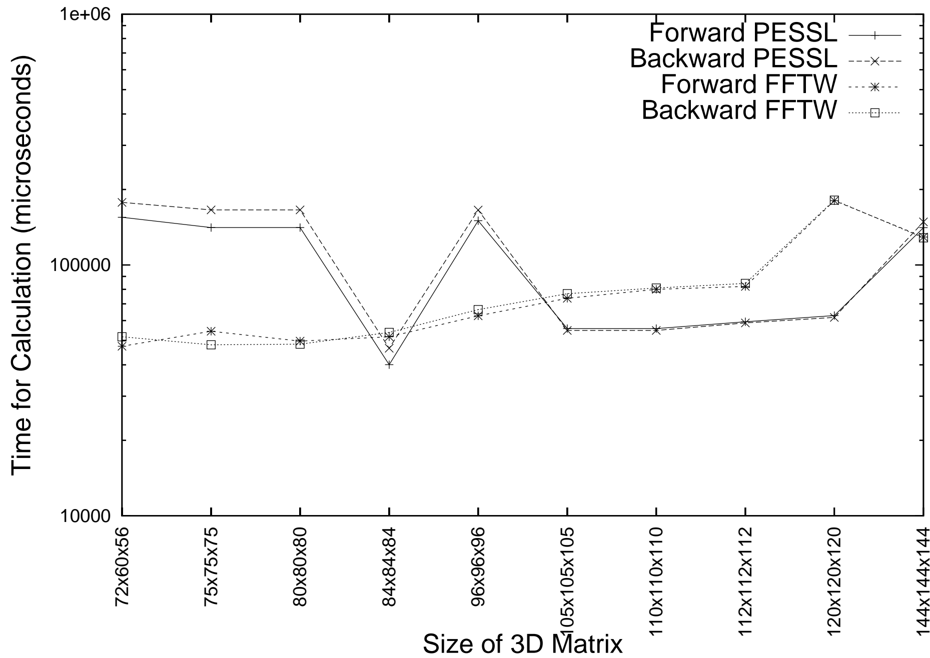

Figure 22: Times for FFT calculations using non-power-of-2 matrices and 64 processors

5 Summary

For serial programs the choice of ESSL or FFTW will not seriously effect your performance, as both giving similar performance, with FFTW showing slightly better performance for small matrices and ESSL shows slightly better performance for larger matrices.

Regarding parallel performance, PESSL is usually better, being, generally slightly faster than FFTW. For scaling, however, FFTW is significantly better than PESSL, especially for large processor numbers.

Also, it must be noted that FFTW is much more portable than PESSL, so the choice of library must depend on the specifics of each individual code.

Finally, both libraries lack the facility to decompose data in more than one-dimension, however, such a decomposition can be built by the user out of the serial FFT routines.

5.1 Timings

As with any benchmarking performed on HPCx, there is a wide variation in the results for a single program. This effect is most apparent when using more than one LPAR (i.e. when data is being sent across the interconnect), where the time taken for two independent runs of the same test, lasting less than one second, can differ by as much as 30%. This variation is partly due to fluctuations in the use of the interconnect (as it is shared with other users), and partly due to operating system issues. Each LPAR runs its own copy of the operating system, which takes some of the compute resources of the LPAR. If an LPAR's operating system is using significant resources this will slow the operation of that LPAR, and consequently of the whole parallel job (as parallel jobs take as long as the slowest process). It is possible that if only 7 out of the 8 processors in an LPAR was used to perform the benchmark tests, then the performance and scaling of the tested libraries would have improved.

It should also be noted that an anomalously large time for all the FFT libraries in a benchmarking run (usually the first of the executables in the batch script). Whilst this was not a frequent occurrence, when it did happen the time to compute the FFT was dramatically increased (i.e. taking 100 times longer than normal). This occurred more commonly using the PESSL libraries, and could possibly be

attributed to the dynamic nature of the the library (PESSL can only be linked dynamically). However, this phenomena could also be associated with the operating system overhead associated with each LPAR.

5.2 Memory usage

Both PESSL and FFTW routines require sufficiently large memory structures for the computation of FFTs. Whilst in most cases this can be achieved by allocating a local memory structure the same size as the processors local data set, for large processor number this may not be sufficient. Problems may arises when the number of processors is larger than some of the dimensions of the global matrix. Given than only one dimension is decomposed over processors, and if that dimension is larger than (or equal to) the number of processor, then one might expect that a local array should have the extent $(x/\text{nproc}, y, z)$ (where the global array has extent (x, y, z).

However, when the number of processors is larger than y or z, FFTW and PESSL will crash if this local array size is used. This is because parallel FFTs are usually performed using a number of single dimension FFTs. To achieve this with a distributed data set requires some movement of data between processors for the FFTs, generally an array *transpose* from one dimension to another. This requires each processor to have sufficiently large memory arrays to be able to store arrays in both normal and transpose forms. Whilst this is an issue, the problem was only discovered when performing the *same size* benchmarks which, as previously stated, use an unusual matrix size. It is unlikely that users would encounter this problem in practice.

NOTE: FFTW provides a useful routine (`fftwnd_mpi_local_size`) that will give you the total size that the local array should be. If this problem occurs for PESSL then it can be fixed by increasing the size of the local memory structures so they can cope with the parallel matrix transposes.

5.3 Environment variable OMP_NUM_THREADS

Finally, the PESSL library performance appears to be affected by the value of the system variable `OMP_NUM_THREADS`. This variable is primarily used to specify the number of OpenMP threads a program will spawn, but also controls a number of other system threads. By default OMP_NUM_THREADS is undefined, however, to get the best performance for the codes employed here which employed PESSL and MPI, it should be set: `OMP_NUM_THREADS = 1`. This will restrict the system from spawning any unnecessary threads which would impact the performance of a users code. This variable does not appear to affect the performance of FFTW, which would suggest that it is controlling thread spawning in either PESSL or BLACS, rather than MPI (as FFTW uses MPI). However, whilst setting this variable to 1 will improve the performance of PESSL/BLACS, it may reduce the performance of any other libraries used within a program, so it should be used with care.

5.4 Future Work

Both FFTW and PESSL can be further optimised and modified using various library options (i.e. strided ESSL/FFTW, IP(1)=0/FFTW_TRANSPOSED_ORDER, etc), alternative compiler flags (i.e. fast, -O5, etc) and system variables (i.e. MP_EAGER_LIMIT, MP_POLLING_INTERVAL, etc). Further work is needed to investigate whether the performance of these libraries can be improved, both in terms of computation time and scaling. Methods for optimising both libraries are discussed in their associated documentation.

Benchmarking FFTW using an optimised MPI implementation may improve the performance of FFTW parallel FFTs. Also, if we use fewer processors per LPAR, then the parallel FFTs may give better scaling for both libraries.

The performance of the libraries using very large numbers of processors (> 512) and very large data sets also need to be further investigated. Also, it would be beneficial to benchmark other libraries

that perform FFTs, especially if they allow data to be decomposed over more that one dimension.

Often, parallel FFTs are computed using serial FFT libraries and performing the necessary communication within the calling program (i.e. manually performing a number of serial FFTs and swapping data between processors). It would be interesting to benchmark FFTW and ESSL when used in this style and compare the performance to PESSL and the parallel routines given by FFTW.

Finally, it maybe of interest to compare the performance of the underlying communication libraries, namely MPI vs BLACS.

References

[1] **ESSL for AIX: Guide and Reference**, Version 3.0, IBM Document Number: SA22-7272-04

[2] **Parallel ESSL for AIX Guide and Reference**, Version 2.0, IBM Document Number: SA22-7273-04

[3] **FFTW Manual for Version 2.1.4**, Frigo, M., Johnson, S. G., http://www.fftw.org/fftw2_doc/, March 2003.

[4] **A User's Guide to the BLACS**, Dongarra, J. J., van de Geijn, R. A., Whaley, R. C., Oak Ridge National Laboratory, Oak Ridge, Tennessee, June 1994.

[5] **Basic Linear Algebra Subprograms: A Quick Reference Guide**, Dongarra, J. J., et al., University of Tennessee, Oak Ridge National Laboratory, Numerical Algorithms Group Ltd, May 1997.

[6] **LAPACK Users' Guide**, Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen D., Third Edition, SIAM, Philadelphia, August 1999.

[7] **FFTW: An Adaptive Software Architecture for the FFT**, Frigo, M., Johnson, S. G., http://www.fftw.org/fftw-paper-icassp.pdf.

A Code fragments

A.1 ESSL code fragments

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include<essl.h>
#include<dfftw.h>

#define nx 8
#define ny 8
#define nz 8

#define naux 100 + nx*ny*nz

  double aux[naux];
  double size = nx*ny*nz;
  dcmplx ***t;
```

```
t = (dcmplx ***)malloc(nx*sizeof(dcmplx **));
for(i=0; i<nx; i++){
  t[i] = (dcmplx **)malloc(ny*sizeof(dcmplx *));
  for(j=0; j<ny; j++){
    t[i][j] = (dcmplx *)malloc(nz*sizeof(dcmplx));
  }
}

// Do the forward FFT
isign = 1;
scale = 1.0;
dcft3(t,nx,nx*ny,t,nx,nx*ny,nx,ny,nz,isign,scale,aux,naux);

// Do the backward FFT
isign = -1;
scale = 1.0/size;
dcft3(t,nx,nx*ny,t,nx,nx*ny,nx,ny,nz,isign,scale,aux,naux);
```

A.2 Serial 3D FFTW code fragments

```
fftw_complex *t;
fftwnd_plan forwardplan, backwardplan;

t = (fftw_complex *)malloc((nx*ny*nz)*sizeof(fftw_complex));

/* printf("Initialising the FFTW plans\n");*/
forwardplan = fftw3d_create_plan(nx,ny,nz,FFTW_FORWARD,FFTW_IN_PLACE);
backwardplan = fftw3d_create_plan(nx,ny,nz,FFTW_BACKWARD,FFTW_IN_PLACE);

// pack the t array

for(i=0; i<nx; i++){
  for(j=0; j<ny; j++){
    for(k=0; k<nz; k++){
      t[k + nz * (j + ny * i)].re = b[i][j][k];
      t[k + nz * (j + ny * i)].im = 0.0;
    }
  }
}

// Do the forward FFT
scale = 1.0;
fftwnd_one(forwardplan,&t[0],NULL);

// Do the backward FFT
scale = 1.0/size;
fftwnd_one(backwardplan,&t[0],NULL);

fftwnd_destroy_plan(forwardplan);
fftwnd_destroy_plan(backwardplan);
fftw_free(t);
```

A.3 PESSL code fragments

```
int icontext;
size_t lorder;
int prow,pcol,bnum;
dcmplx t[(int)ceil((double)nx/(double)procs)][ny][nz];

blacs_pinfo(&bnum,&pcol);
lorder = strlen(order);
blacs_get(&intzero, &intzero, &icontext);
prow = 1;
blacs_gridinit(&icontext,&order,&prow,&pcol,lorder);
blacs_gridinfo(&icontext,&nrow,&ncol,&myrow,&mycol);

for(i=0; i<localnx; i++){
  for(j=0; j<ny; j++){
    for(k=0; k<nz; k++){
      RE(t[i][j][k]) = b[i][j][k];
      IM(t[i][j][k]) = 0.0;
    }
  }
}

// Do the forward FFT
isign = 1;
scale = 1.0;
ip[0] = 0;
ip[1] = 1;
ip[19] = 0;
ip[20] = 0;
ip[21] = 0;
ip[22] = 0;
pdcft3(t,t,nx,ny,nz,isign,scale,icontext,ip);

// Do the backward FFT
isign = -1;
scale = 1.0/size;
ip[0] = 0;
ip[1] = 1;
ip[19] = 0;
ip[20] = 0;
ip[21] = 0;
ip[22] = 0;
pdcft3(t,t,nx,ny,nz,isign,scale,icontext,ip);

blacs_exit(0);
```

A.4 Parallel 3D FFTW code fragments

```
int localnx,localxstart,localny,localystart,totallocalsize;

fftw_complex *t;
```

```
fftw_complex a;
fftwnd_mpi_plan forwardplan, backwardplan;

forwardplan = fftw3d_mpi_create_plan(MPI_COMM_WORLD,nx,ny,nz,FFTW_FORWARD,FFTW_ESTIMATE);
backwardplan = fftw3d_mpi_create_plan(MPI_COMM_WORLD,nx,ny,nz,FFTW_BACKWARD,FFTW_ESTIMATE);

fftwnd_mpi_local_sizes(forwardplan, &localnx, &localxstart, &localny, &localystart, &totallocalsize);

t = (fftw_complex *)malloc(totallocalsize*sizeof(fftw_complex));

for(i=0; i<localnx; i++){
  for(j=0; j<ny; j++){
    for(k=0; k<nz; k++){
      b[i][j][k] = b[i][j][k] - zerofactor;
      t[k + nz * (j + ny * i)].re = b[i][j][k];
      t[k + nz * (j + ny * i)].im = 0.0;
    }
  }
}

// Do the forward FFT
fftwnd_mpi(forwardplan,1,&t[0],NULL,FFTW_NORMAL_ORDER);

// Do the backward FFT
fftwnd_mpi(backwardplan,1,&t[0],NULL,FFTW_NORMAL_ORDER);

fftwnd_mpi_destroy_plan(forwardplan);
fftwnd_mpi_destroy_plan(backwardplan);
fftw_free(t);
```

B Matix sizes for benchmarks

Matrix Size
$5 \times 5 \times 5$
$6 \times 6 \times 6$
$7 \times 7 \times 7$
$8 \times 8 \times 8$
$9 \times 9 \times 9$
$10 \times 10 \times 10$
$11 \times 11 \times 11$
$12 \times 12 \times 12$
$13 \times 13 \times 13$
$14 \times 14 \times 14$
$15 \times 15 \times 15$
$16 \times 16 \times 16$
$24 \times 25 \times 28$
$24 \times 28 \times 28$
$48 \times 48 \times 48$
$49 \times 49 \times 49$
$56 \times 56 \times 56$
$60 \times 60 \times 60$
$72 \times 60 \times 56$
$75 \times 75 \times 75$
$80 \times 80 \times 80$
$84 \times 84 \times 84$
$96 \times 96 \times 96$
$105 \times 105 \times 105$
$110 \times 110 \times 110$
$112 \times 112 \times 112$
$120 \times 120 \times 120$
$144 \times 144 \times 144$

Table 3: Sizes for Serial non-power-of-2 FFT benchmarks

Matrix Size
$16 \times 16 \times 16$
$24 \times 25 \times 28$
$24 \times 28 \times 28$
$48 \times 48 \times 48$
$49 \times 49 \times 49$
$56 \times 56 \times 56$
$60 \times 60 \times 60$
$72 \times 60 \times 56$
$75 \times 75 \times 75$
$80 \times 80 \times 80$
$84 \times 84 \times 84$
$96 \times 96 \times 96$
$105 \times 105 \times 105$
$110 \times 110 \times 110$
$112 \times 112 \times 112$
$120 \times 120 \times 120$
$144 \times 144 \times 144$

Table 4: Sizes for 8 and 16 processor non-power-of-2 FFT benchmarks

Matrix Size
$72 \times 60 \times 56$
$75 \times 75 \times 75$
$80 \times 80 \times 80$
$84 \times 84 \times 84$
$96 \times 96 \times 96$
$105 \times 105 \times 105$
$110 \times 110 \times 110$
$112 \times 112 \times 112$
$120 \times 120 \times 120$
$144 \times 144 \times 144$

Table 5: Sizes for 64 processor non-power-of-2 FFT benchmarks