Optimizing parallel 3D Fast Fourier Transformations for a cluster of IBM POWER5 SMP nodes

Ulrich Sigrist

August 24, 2007

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2007
Abstract

The Fast Fourier Transformation (FFT) of three-dimensional (3D) data is of particular importance for many numerical simulations used in High Performance Computing codes. The parallelization of the 3D FFT is usually done using a one-dimensional decomposition of the input data. This approach imposes limitations in terms of scalability to large numbers of processors. This dissertation presents the results of our investigations on how to optimize the parallel 3D FFT using the two-dimensional (2D) data decomposition on HPCx, a cluster of POWER5 SMP nodes. We examined the properties of the IBM HPS interconnect on HPCx with respect to the All-to-All communication pattern which is crucial for the parallel FFT. We present the results from our analysis on how the mapping of the virtual 2D processor grid to the processors in the SMP nodes affects the overall performance of the parallel 3D FFT. We further present experimental results which demonstrate good scaling of the 2D data decomposition approach up to 1024 processors for problem sizes between $128^3$ and $512^3$ on HPCx. Furthermore, we compare the performance of the two different MPI All-to-All communication routines and the impact of using derived data types compared to manual buffer packing. And finally, we present the results of our performance comparisons between the 1D FFT implementations of the open source Fastest Fourier Transformation in the West (FFTW) library and IBMs Engineering Scientific Subroutine Library (ESSL).
## Contents

Chapter 1 Introduction ........................................................................................................ 1

Chapter 2 Parallel Fast Fourier Transformation ................................................................. 3
  2.1 Fast Fourier Transformation of three-dimensional data ............................................. 3
  2.2 Parallel three-dimensional FFT ................................................................................ 5
    2.2.1 Parallel 3D FFT using 1D decomposition ....................................................... 5
    2.2.2 Parallel 3D FFT using 2D decomposition ....................................................... 6
  2.3 Implementation and verification ............................................................................... 7
    2.3.1 Implementation of the test program .............................................................. 7
    2.3.2 Timing and measurement considerations .................................................... 8
    2.3.3 Verification of the FFT results ..................................................................... 9

Chapter 3 Parallel FFT on HPCx ..................................................................................... 12
  3.1 HPCx hardware ...................................................................................................... 12
    3.1.1 Cluster components .................................................................................... 12
    3.1.2 Cluster interconnect .................................................................................. 13
  3.2 Bandwidth measurements on HPCx ...................................................................... 16
    3.2.1 Single node (intra-node) bandwidth properties .......................................... 17
    3.2.2 Two node (inter-node) bandwidth properties ............................................. 20
3.3 Task placement for parallel 3D FFT on HPCx .......................................................23

3.3.1 Problem size, local memory and send size considerations ............................25

3.3.2 Properties of All-to-All within the row- and column communicators.........27

3.3.3 All-to-All within the row communicators ......................................................28

3.3.4 All-to-All within the column communicators ................................................30

3.3.5 Combination of All-to-All in Row- and column communicators..............36

3.3.6 Node placement on the HPCx network ........................................................38

Chapter 4 Performance of parallel 3D FFT on HPCx .............................................42

4.1 Factors influencing the performance of the parallel 3D FFT .........................42

4.1.1 MPI All-to-All APIs .....................................................................................43

4.1.2 Derived data types .......................................................................................49

4.1.3 Fast Fourier Transformation libraries .......................................................51

4.2 Parallel performance and speedup ....................................................................55

Chapter 5 Conclusions .............................................................................................59

References ...............................................................................................................66
List of Tables

Table 1: Latency times (in μs) between nodes in different frames on HPCx .......... 15
Table 2: Two-node All-to-All timings (in μs) for various node combinations relative to send size for send sizes between 2 and 512 bytes ................. 22
Table 3: Amount of data stored on each processor locally (DP) for P > N and problem sizes N^3 (with N=N_x=N_y=N_z) ........................................ 26
Table 4: Send sizes (in bytes) for row- and column All-to-All for 256 processors and problem size N=128 for all possible virtual processor grid layouts ............................................................... 27
Table 5: Virtual processor grid layout (Pr x Pc) for P≥32 used for the performance measurements shown in Figure 36 ........................................ 58
Table 6: Average total communication (row- + column communication) time (in s) for All-to-All communications on 256 processors for various problem sizes and virtual processor grid layouts (details to Figure 22) ........................................ 62
Table 7: Average times (in s) for All-to-All within the row communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 14) ......................................................... 63
Table 8: Minimum times (in s) for All-to-All within the row communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 14) ......................................................... 63
Table 9: Average times (in s) for All-to-All within the columns communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 19) .......................................................... 63
Table 10: Minimum times (in s) for All-to-All within the columns communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 19) ......................................................... 63
Table 11: Detail average times (in s) for 1D FFT comparison between ESSL and FFTW (Figure 34 and Figure 35) .......................................................... 64
Table 12: Detail average times (in s) for scaling test in Figure 36 on 1 to 1024 processors using different problem sizes from 32^3 to 512^3 ................. 64
Table 13: Detail times (in s) for the comparison between `MPI_Alltoall` using manual buffer packing (2 versions) and `MPI_Alltoall` using a derived data type (Figure 33)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual buffer packing (2 versions)</td>
<td></td>
</tr>
<tr>
<td>Derived data type</td>
<td></td>
</tr>
</tbody>
</table>

Figure 33: Description of the figure showing the comparison.
List of Figures

Figure 1: Processing steps for parallel 3D FFT using 1D (slab) data decomposition........................................................................................................5

Figure 2: Processing steps for parallel 3D FFT using 2D data decomposition ........7

Figure 3: Overview HPCx cluster components....................................................12

Figure 4: Simplified HPS Switch Board schema ...............................................13

Figure 5: Two-plane HPS network topology.......................................................14

Figure 6: Sample latency times between nodes in frames f403 and f405 and other nodes in the Capacity region of HPCx.........................15

Figure 7: Single node Multi-PingPong bandwidth (minimum and average values) using 16 processors (8 groups of 2 processors) for various message sizes and eager limit settings of 64 KB and 4 KB ................18

Figure 8: Single node All-to-All bandwidth relative to send size for eager limits 64 KB and 4 KB. Boundaries of the cache lines are shown as reference.................................................................19

Figure 9: Two-node Multi-PingPong bandwidth between nodes in frames connected through either one NSB (f403-f403 and f403-f404) or two NSBs and one ISB (f403-f405)...............................20

Figure 10: Two-node All-to-All bandwidths between nodes in frames connected through either one NSB (f403-f403 and f403-F404) or two NSBs and one ISB (f403-f405)..................................................22

Figure 11: Mapping of 32 physical processors in two SMP nodes to virtual two-dimensional Cartesian processor grid ........................................24

Figure 12: Mapping of the 2D virtual processor grid to the 3D data and creation of row- and column communicators for the All-to-All communication from the Cartesian grid........................................25

Figure 13: Location of physical processors in row-/column communicator for 32 processors using a 4x8 virtual processor grid.........................28
Figure 28: Bandwidth comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 16 processors in one node for send sizes from 4 bytes to 8 MB using Intel MPI Benchmarks................................................................................. 45

Figure 29: Time comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 32 processors (two full nodes) for send sizes between 4 bytes and 8 MB (16 processor \textit{MPI\_Alltoall} times shown as reference).......................... 46

Figure 30: Time comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 64 processors (4 full nodes) for send sizes between 64 bytes and 512 KB (32 processor \textit{MPI\_Alltoall} times shown as reference).............................. 47

Figure 31: Total communication time (row comm. + column comm.) comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 64 processors using the parallel FFT test program for problem sizes of $8^3$ to $256^3$ ................................. 48

Figure 32: Total communication time (row comm. + column comm.) comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 256 processors using the parallel FFT test program for problem sizes of $64^3$ (send size 1024) to $512^3$ (send size 524288).............................................................................. 49

Figure 33: Comparison between \textit{MPI\_Alltoall} using manual send buffer packing (2 versions) and \textit{MPI\_Alltoall} using a derived data type as send data type on 16 processors for different send sizes (normalized times relative to faster manual buffer packing)............................................. 51

Figure 34: Performance comparison between ESSL and FFTW non-strided FFT implementations for different problem sizes (relative to the performance of ESSL)................................................................. 53

Figure 35: Performance comparison between ESSL and FFTW strided FFT implementations for different problem sizes (relative to the performance of ESSL)........................................................................ 55

Figure 36: Performance measurements for the parallel 3D FFT using 2D data decomposition for problem sizes from $32^3$ to $512^3$ on up to 1024 processors (average times) ................................................................................................. 56

Figure 37: Speedup for the parallel 3D FFT using 2D data decomposition for problem sizes from $32^3$ to $512^3$ on up to 1024 processors (using timings from Figure 36)...................................................................................... 57

Figure 38: Alternative bisectional bandwidth graph (using average times) for All-to-All communication within columns using 256 processors and different virtual processor grid layouts................................................................. 62
Acknowledgements

I would like to thank my project supervisor Dr Joachim Hein for his support and guidance throughout this project.

I would also like to thank Dr Kenton D'Mellow for his help during the final stages of the project.

Most importantly, I would like to thank Yvonne for her invaluable moral support during my time in Edinburgh.

Special thanks go to Hans-Peter, Ruedi, Markus and Peter for making this experience possible for me.
Chapter 1

Introduction

The Fast Fourier Transformation (FFT) is an important computational kernel for many scientific applications. The FFT of three-dimensional (3D) input data is of particular importance to many numerical simulations and is widely used in High Performance Computing (HPC) codes running on large numbers of processors. Unfortunately the parallel FFT is, due to its high communication demands, often the limiting factor preventing the efficient usage of larger numbers of processors for a given problem size.

To address these scalability issues with the most commonly used parallelization strategy for parallel 3D FFT, the one-dimensional (1D), or slab data decomposition, an alternative approach based on a two-dimensional (2D) data decomposition was discussed for the IBM BlueGene/L supercomputer in [1]. Both decomposition strategies are described in Chapter 2. Furthermore, a recent MSc dissertation presented by H. Jagode [2] has shown the importance of task placement for 3D parallel Fast Fourier Transformations on the IBM BlueGene/L system. The overall performance of the parallel 3D FFT was improved by choosing a task placement which takes the characteristics of the BlueGene/L torus network into account.

The hardware architecture of HPCx is completely different from the BlueGene/L architecture. The Symmetric Multi Processor (SMP) cluster nodes on HPCx are connected via a hierarchical network with different latency and bandwidth properties for communications within an SMP node and between nodes connected through one or more network hierarchy levels.
In order to explore the potential of the parallel 3D FFT using 2D data decomposition on HPCx, we investigate the hardware properties of the SMP nodes and the network with respect to the communication patterns associated with the parallel FFT using standard benchmarks (Multi-PingPong and All-to-All benchmarks) and our own implementation of the parallel 3D FFT using 2D data decomposition.

Based on the knowledge of the communication properties we then examine the scope for task placement and its impact on the performance of the parallel 3D FFT on HPCx. We present a comparison between different mappings of the 2D data decomposition on the SMP nodes for different problem sizes and task placements.

Finally, we look at other factors that affect the performance of the parallel 3D FFT. We compare the performance of different All-to-All communication routines as well as the influence the FFT library has on the overall performance.

The rest of this document is structured as follows. Chapter 2 contains the mathematical information on the Discrete Fourier Transformation followed by a discussion of the parallelization strategies for the Fast Fourier Transformation and the description of the test program. In Chapter 3, we start with an overview of the HPCx cluster components and present the results from the bandwidth measurements. We then present the task mapping strategies and discuss the performance impact of different mappings on the overall performance of the parallel 3D FFT. In Chapter 4 we look at the factors other than the All-to-All communication affecting the performance of the parallel 3D FFT. We compare the performance of different All-to-All communication routines and investigate how the usage of derived data types affect performance in comparison with the manual packing of the send buffer in the All-to-All routines. Finally, we look at how the performance of the open source Fastest Fourier Transform in the West (FFTW) library compares with the proprietary Engineering Scientific Subroutine Library (ESSL) provided by IBM. In Chapter 5 we summarize the results and draw our conclusions.
Chapter 2

Parallel Fast Fourier Transformation

2.1 Fast Fourier Transformation of three-dimensional data

The Fast Fourier Transformation is an algorithm for computing the discrete Fourier Transformation (DFT) efficiently. The forward DFT of a complex one-dimensional input array \( X \) of size \( N \) to a complex one dimensional output array \( Y \) of size \( N \) is defined by

\[
Y(k) = \sum_{j=0}^{N-1} X(j) e^{-2\pi i \frac{j k}{N}}, \quad 0 \leq k < N
\]

The backward discrete Fourier Transformation for the one-dimensional case using the same definitions as in (1) is defined by

\[
X(j) = \frac{1}{N} \sum_{k=0}^{N-1} Y(k) e^{2\pi i \frac{j k}{N}}, \quad 0 \leq j < N
\]

The runtime complexity for an algorithm to evaluate the sums in (1) and (2) directly is \( O(N^2) \). A Fast Fourier Transformation algorithm can compute the same result in \( O(N \log N) \) operations. For implementation details of the FFT algorithm see [3] or [4].

The forward DFT of a three-dimensional complex input array \( X = \{X(0 : N_x -1, 0 : N_y -1, 0 : N_z -1)\} \) (size \( N_x \times N_y \times N_z \)) to a complex three-dimensional output array \( Y = \{Y(0 : N_x -1, 0 : N_y -1, 0 : N_z -1)\} \) is defined as
The corresponding backward DFT using the same definitions as in (3) is defined as

\[
X(j_x, j_y, j_z) = \frac{1}{N_x N_y N_z} \sum_{j_x=0}^{N_x-1} \sum_{j_y=0}^{N_y-1} \sum_{j_z=0}^{N_z-1} Y(k_x, k_y, k_z) e^{-2\pi i \left( \frac{k_x j_x}{N_x} + \frac{k_y j_y}{N_y} + \frac{k_z j_z}{N_z} \right)}, \quad 0 \leq j_x < N_x, \quad 0 \leq j_y < N_y, \quad 0 \leq j_z < N_z.
\]

Equation (3) can be rewritten in the following form to demonstrate that the three-dimensional case can be calculated as three separate sets of one-dimensional DFT:

\[
Y(k_x, k_y, k_z) = \sum_{j_x=0}^{N_x-1} \left( \sum_{j_y=0}^{N_y-1} \left( \sum_{j_z=0}^{N_z-1} X(j_x, j_y, j_z) e^{-2\pi i \left( \frac{k_x j_x}{N_x} \right)} \right) e^{-2\pi i \left( \frac{k_y j_y}{N_y} \right)} \right) e^{-2\pi i \left( \frac{k_z j_z}{N_z} \right)}.
\]

The first step calculates the DFT over the \( z \)-dimension. This has to be done by calculating \( N_x \times N_y \) one-dimensional DFTs, each of \( N_z \) elements. This leads to the intermediate result

\[
Y_z(j_x, j_y, k_z) = \sum_{j_z=0}^{N_z-1} X(j_x, j_y, j_z) e^{-2\pi i \left( \frac{k_z j_z}{N_z} \right)} \quad 0 \leq j_x < N_x, \quad 0 \leq j_y < N_y.
\]

The same procedure has to be applied for the \( y \)-dimension by calculating \( N_x \times N_z \) one-dimensional DFTs, each of \( N_y \) elements, leading to the intermediate result

\[
Y_{yz}(j_x, k_y, k_z) = \sum_{j_y=0}^{N_y-1} Y_z(j_x, j_y, k_z) e^{-2\pi i \left( \frac{k_y j_y}{N_y} \right)} \quad 0 \leq j_x < N_x, \quad 0 \leq k_z < N_z.
\]

And finally for the \( x \)-dimension by calculating \( N_y \times N_z \) one-dimensional DFTs, each of \( N_x \) elements

\[
Y(k_x, k_y, k_z) = \sum_{j_x=0}^{N_x-1} Y_{yz}(j_x, k_y, k_z) e^{-2\pi i \left( \frac{k_x j_x}{N_x} \right)} \quad 0 \leq k_y < N_y, \quad 0 \leq k_z < N_z.
\]
(eq. (5), (6), (7), (8) using conventions similar to [1]). The order in which the three steps are calculated is not relevant for the final result.

### 2.2 Parallel three-dimensional FFT

In order to calculate the Fast Fourier Transformation of a three-dimensional array of complex numbers, the input array has to be decomposed among the available processors. There are two different decomposition strategies used for the parallel 3D FFT. The more commonly used 1D or slab decomposition and the 2D decomposition also known as **volumetric** or **pencil** decomposition as discussed in [1].

#### 2.2.1 Parallel 3D FFT using 1D decomposition

The 1D decomposition is the easiest and most common decomposition technique used in existing parallel FFT libraries. The 3D input data array is decomposed along one axis into slabs. Each of the available processors P gets the data of one slab in its local memory. For a problem size of \( N_x \times N_y \times N_z \) the data is split along the \( x \)-axis in \( P \) slabs with each slab containing \( \frac{N_x}{P} \times N_y \times N_z \) complex numbers. With the slabs distributed on the processors each processor can then compute step 1 and 2 (eq. (6) and (7)) on the data in its local memory without having to communicate with the other processors.

![Processing steps for parallel 3D FFT using 1D (slab) data decomposition](image)

Figure 1: Processing steps for parallel 3D FFT using 1D (slab) data decomposition

Figure 1 shows the slab decomposition for a \( 4 \times 4 \times 4 \) problem using 4 processors. The coordinate system for the data array is rooted in the bottom left corner and the memory
locations are contiguous along the initial $z$-axis assuming C array memory layout (row-major order with the ‘rightmost’ dimension contiguous in memory). Each cube represents a memory location. The numbers on the cubes are the global index of the memory location again assuming C array notification.

The main problem with the slab decomposition can be seen in Figure 1. The maximum number of processors is limited by the maximum number of available slabs. For the example in Figure 1 we cannot use more than 4 processors. This causes a major limitation in terms of scalability for large problem sizes once the number of processor exceeds the number of possible slabs.

2.2.2 Parallel 3D FFT using 2D decomposition

The 2D decomposition addresses the scalability issue of the slab decomposition by mapping the 3D input data on a 2D processor grid. For a problem size of $N_x \times N_y \times N_z$, the data is distributed on a $P_x \times P_y$ processor grid. Each processor now holds

$$\frac{N_x}{P_x} \times \frac{N_y}{P_y} \times N_z$$

complex numbers in its local memory. Each processor calculates

$$\frac{N_x}{P_x} \times \frac{N_y}{P_y}$$

one-dimensional FFTs over the $z$-axis resulting in $Y_z$ (see eq. (6)). The processors within the same row of the virtual processor grid then perform an All-to-All communication to swap the $y$- and $z$-axis. In step 2 each processor then calculates

$$\frac{N_x}{P_x} \times \frac{N_z}{P_z}$$

one-dimensional FFTs over the $y$-axis resulting in $Y_{yz}$ (see eq. (7)). In preparation for the final step, the processors within the same column of the virtual processor grid perform an All-to-All communication to swap the $y$- and $x$-axis. The final result is then calculated by performing

$$\frac{N_y}{P_y} \times \frac{N_z}{P_z}$$

one-dimensional FFT over the $x$-axis. The procedure is shown in Figure 2. Each processor initially holds a pencil style piece of the entire data with the same colour. The All-to-All within the rows moves the data horizontally (between $P_0$ and $P_1$ and between $P_2$ and $P_3$) whereas the All-to-All within the columns moves the data vertically (between $P_0$ and $P_2$ and between $P_1$ and $P_3$).
It is easy to see from Figure 2 that a maximum of \( \min(N_x \times N_y, N_x \times N_z, N_y \times N_z) \) processors can be used to compute the 3D FFT of the \( N_x \times N_y \times N_z \) problem. This comes at the cost of one extra All-to-All communication. Since All-to-All communication is in most cases slower than rearranging the data in local memory, the 2D decomposition approach pays off only if there are more processors available than we can utilize using the slab decomposition.

### 2.3 Implementation and verification

#### 2.3.1 Implementation of the test program

We implemented a modular program as a test bed for performance evaluations of the parallel 3D Fast Fourier Transformation using the 2D data composition approach described in section 2.2.2. The program is written in C and we use C conventions for 3D array memory layout throughout this document.

The complex 3D input data – using double precision complex numbers – for the parallel FFT algorithm is constructed on every processor by using an easy to verify trigonometric function (see 2.3.3 for details). The values for the complex input array are calculated based on the coordinates of the processor in the virtual 2D processor
grid. Our timing measurements do not include the initialization of the input data, simulating an environment where the input data is already distributed on the processor grid. The initialization phase is concluded with a global barrier in order to prevent distortion of the timing measurements.

2.3.2 Timing and measurement considerations

As we have seen in Figure 2, the algorithm for the parallel 3D forward FFT using 2D data decomposition is divided into the following steps:

1. One dimensional FFT(s) in $z$ direction
2. Transposition of the $z$- and $y$-axis using All-to-All within the rows
   a. pack All-to-All send buffer
   b. perform the All-to-All
   c. unpack the All-to-All receive buffer
3. One dimensional FFT(s) in $y$ direction
4. Transposition of the $y$- and $x$-axis using All-to-All within the columns
   a. pack All-to-All send buffer
   b. perform the All-to-All
   c. unpack the All-to-All receive buffer
5. One dimensional FFT(s) in $x$ direction

The structure for the backward transformation is identical but in reverse order (5, 4a, 4b, 4c, 3, ..., 1). In order to allow detailed investigations of the runtime behaviour of our test program, we instrumented the code with a series of timers to measure the time it takes to execute each of the steps above separately. The forward and the backward transformation are timed independently. There is an additional timer, called forward, (backward for the backward transformation) to measure the total time it takes to execute steps 1 to 5.

The timers have a constant overhead of ~1μs for a complete start/stop sequence. Overhead from nested timers is not removed from the results. This is not considered a problem since the timing overhead is constant and very small. However, for small problem sizes, computed on relatively large numbers of processors, the timing
overhead does become significant. Timing results with values close to the timer resolution are either marked with error indicators or discarded.

The timings are taken from a number of iterations, between 5 and 500 depending on the problem size, each performing a complete FFT. A number of iterations at the beginning of each measurement are considered warm-up runs and are not included in the results. At the beginning of each iteration (before step 1 and before the forward timer is started) the processes are synchronized at a global barrier to avoid distortion of the timing results at the first collective communication. A second barrier is used to synchronize the processes globally after step 5 and after the forward timer is stopped. The same applies to the backward transformation correspondingly.

In each iteration we collect the time for each of the step timers separately from all processors on processor 0 using a global sum reduction. This value (divided by the number of processors) is the average time across all processors for one iteration.

Based on this time the program reports for each timer two different values:

1. The minimum time is the time of the fastest iteration. The fastest iteration is determined by the forward timer. The values of all timers for the iteration with the best forward time are reported as min value.

2. For the average time we keep a running total for each timer over all relevant iterations. The running total divided by the number of relevant iterations is reported as avg value.

Using the difference between the min time and the avg time allows us to establish a measure of the scattering of our timing results. It has to be noted that we do not divide the results of the global sum reduction by the number of processors at each iteration. Instead we work with the sum values and calculate the necessary division only for reporting. This is to avoid rounding errors in the timing results.

2.3.3 Verification of the FFT results

The 3D FFT using 2D data decomposition code requires a lot of copy operations within the local memory and communication between processors. It is therefore essential that
we are able to verify the results of the transpositions and transformations directly in the test program. We have implemented two different verification routines. The two routines are applied at different stages in every FFT iteration. The first verification is done on the result of the forward FFT. The frequency domain data is verified against the results we expect from the given input data. This requires the input data to be created using a function with well known properties with respect to the Fourier Transformation. The trigonometric function used to create the input data is

\[
X(j_x, j_y, j_z) = \sin \left( \frac{2\pi j_y}{N_y} + \frac{2\pi j_z}{N_z} \right) 0 \leq u < N_x, k \in \mathbb{Z} \\
\quad \quad \quad \quad \quad \quad \quad 0 \leq u < N_y, l \in \mathbb{Z} \\
\quad \quad \quad \quad \quad \quad \quad 0 \leq w < N_z, m \in \mathbb{Z}
\]

The values for \(u\), \(v\) and \(w\) are randomly chosen for each program run. This function has the property (see full mathematical details in [2]) that only two values in the frequency domain are expected to be nonzero. The coordinates of the two peak points for the DFT of \(X(j_x, j_y, j_z)\) are given by (as shown in [2])

\[
Y(k_x, k_y, k_z) = \begin{cases} 
\frac{1}{2} \cdot \sqrt{-1} \cdot N_x N_y N_z & \text{if } k_x = u, k_y = v, k_z = w \\
\frac{1}{2} \cdot \sqrt{-1} \cdot N_x N_y N_z & \text{if } k_x = N_x - u, k_y = N_y - v, k_z = N_z - w \\
0 & \text{else}
\end{cases}
\]

Because the verification is done on the transposed data, the coordinates of the peak values, determined by the factors \(u\), \(v\) and \(w\), have to be transposed as well. The order of the transpositions in the test program is as follows:

1. Create input data \(\rightarrow X(j_x, j_y, j_z)\)
2. First FFT in z-axis \(X(j_x, j_y, j_z) \rightarrow Y_z(j_x, j_y, k_z)\)
3. Transposition of y- and z-axis \(Y_z(j_x, j_y, k_z) \rightarrow Y_z(j_x, k_z, j_y)\)
4. Second FFT for y-axis \(Y_z(j_x, k_z, j_y) \rightarrow Y_{yz}(j_x, k_z, k_y)\)
5. Transposition of y- and x-axis \(Y_{yz}(j_x, k_z, k_y) \rightarrow Y_{yz}(k_x, k_z, j_x)\)
6. Third FFT in x-axis

\[ Y_{yz}(k_y, k_z, j_x) \rightarrow Y(k_y, k_z, k_x) \]

The peak values from (10) have to be at \( Y(v, w, u) \) and at \( Y(N_y - v, N_z - w, N_x - u) \).

The second verification is executed in the time domain after the backward transformation. Since we do not perform any manipulations (not even the normalization) on the data in the frequency domain, the result after the backward transformation has to be the same as the original input data with the exception of the normalization factor of \( N_x \times N_y \times N_z \). Both verification routines are implemented to allow for numerical errors. For single precision complex numbers the maximum relative error is defined as \( 1.0 \times 10^{-5} \), for double precision it is \( 1.0 \times 10^{-11} \).
Chapter 3

Parallel FFT on HPCx

3.1 HPCx hardware

3.1.1 Cluster components

HPCx (Phase 3) is a large SMP cluster system built from 160 IBM eServer 575 computing nodes and two IBM eServer 575 service nodes. Each computing node contains 16 1.5 GHz IBM POWER5 64-bit RISC processors resulting in a total of 2560 processors delivering a peak performance of 15.3 Teraflop/s or 12.9 Teraflop/s [5] sustained on Linpack [6]. Each POWER5 chip contains two cores and the 1.92 MB L2 cache. Each core has its own L1 cache – 64 KB for instructions and 32 KB for data. The 16 processors of each computing node are located on 8 Dual Chip Modules (DCM). Each DCM contains one dual core chip and one 36 MB L3 cache chip. The L2- and L3 caches are shared between the two cores of each chip. There is a total of 32 GB main memory available on each computing node.

Figure 3: Overview HPCx cluster components
A frame contains 8 computing nodes. HPCx phase 3 has a total of 20 frames available for batch processing (see Figure 3 for clarification).

Each node of the HPCx phase 3 cluster is configured as a 16-way Logical Partition (LPAR) running its own copy of the IBM AIX (version 5.3) operating system. The node names follow a consistent naming schema in the form of \[1[1−8]F[401−422].\]

Frames f401 and f402 contain the login, interactive and I/O nodes and frames f403-f422 host the computing nodes. The machine is divided into different regions. The two main regions are the Capacity region for jobs requiring up to 128 processors, and the Capability region for jobs requiring more than 128 processors.

### 3.1.2 Cluster interconnect

The HPCx compute nodes are connected through the IBM High Performance Switch (HPS). HPS – also know as ‘Federation’ – is IBMs 4th generation switch and adapter technology first introduced in 2003 [7]. HPS provides a fast multistage bidirectional connection from every node to every other node in the system. The main component of the HPS network is the Switch Board (see Figure 4). A Switch Board consists of 8 logical Switch Chips that are connected as a 4x4 crossbar switch and 32 ports, controlled by Link Driver Chips. The ports of a Switch Board are either connected to a Switch Network Interface (SNI) – the network adapter of an eServer node – or to another Switch Board [8].

![Simplified HPS Switch Board schema](image)

Figure 4: Simplified HPS Switch Board schema

A HPS network is constructed by cascading Switch Boards to form a multistage hierarchical topology. In HPS terminology the Switch Boards are named according to
their position in the network hierarchy. A Switch Board that has direct connections to
nodes is called Node Switch Board (NSB) and a Switch Board that is only connected to
other Switch Boards is called Intermediate Switch Board (ISB) [7]. The end to end
latency of a connection is proportional to the number of Switch Chips traversed since
each Switch Chip traversal adds ~59ns of latency [7].

Each HPC\textsuperscript{x} node is connected to the network through two Switch Network Interface
(SNI) adapters, plugged into the node’s GX I/O bus slot. Each SNI provides two
physical links for a total of four links per node [5]. A HPS link has a peak bandwidth of
2 GB/s for unidirectional- and 4 GB/s for bidirectional communications [7].

Although the details of the HPC\textsuperscript{x} phase 3 HPS connectivity are not publicly available,
the assumption is that the two-plane configuration (see Figure 5) described in [9] for
HPC\textsuperscript{x} Phase 2a is used for Phase 3 as well. The two-plane configuration provides
connections between all nodes using two identical but independent switch networks.
This is achieved by connecting each of the two available SNI of each node to an NSB
in a different switch plane.

![Two-plane HPS network topology](image)

**Figure 5:** Two-plane HPS network topology

Using this topology, every node can communicate with every other node in the system
through a maximum of 3 switch boards (2 NSBs and one ISB). Due to the fixed amount
of latency each Switch Board traversal adds to the total latency, the structure of the
network can be determined by measuring the latency between pairs of nodes. The
latency timings were gathered using the Presta\textsuperscript{laten} [10] benchmark. Presta uses zero
length MPI [11] Send-Receive (Ping-Pong) between pairs of processors to measure latency.
Table 1:  Latency times (in μs) between nodes in different frames on HPCx

Table 1 shows the latency timings measured between two nodes in several frames on HPCx.

![Table 1](image)

Figure 6:  Sample latency times between nodes in frames f403 and f405 and other nodes in the Capacity region of HPCx

Figure 6 illustrates the latencies measured from nodes in frames f403 and f405 (1 and 2 in Table 1) to other nodes in the Capacity region of HPCx. From Table 1 and Figure 6, we can see that communications between neighbouring pairs of frames show the same latency properties of ~5μs. This is due to the fact that pairs of nodes share one Node Switch Board in each plane. This means that the pairs of processors located in the different nodes within the two neighbouring frames are able to communicate with each other through only one NSB hop. For communication paths spanning more than two neighbouring frames, the latency measured is ~0.5μs higher than the latency between nodes in neighbouring frames. The fact that we measure latency values on only two different levels, at ~5.1μs and ~5.6μs, indicates that there is only one Intermediate
Switch Board level used to connect all the frames in the Capacity region on HPCx. Since an NSB-ISB-NSB connection adds at most 4 Switch Chip traversals – only 3 additional traversals are required if the nodes are connected on the same side of the Switch Board (see. Figure 4) – the maximal latency difference between the single NSB connection and the NSB-ISB-NSB connection, considering only the Switch Chip traversals, would be $4 \cdot \frac{59 \mu s}{4} = 239 \mu s$. We measure roughly double that difference most likely caused by other network components along the longer communication path.

### 3.2 Bandwidth measurements on HPCx

As illustrated in the previous section, there is a noticeable structure in the HPCx network from a latency point of view. However, in the context of parallel FFT the All-to-All bandwidth properties of the network are more important than latency. It is important to determine whether the structure of the HPCx communication network has an affect on the bandwidth between nodes in different frames in the same way it has an affect on the latency.

The most important difference in communication bandwidth is given by the fact that HPCx is an SMP cluster. The 16 processors within one node can communicate with each other by means of shared memory access or through memory copy operations. As a result, intra-node communications should be considerably faster than inter-node message exchange. Therefore, we have examined the bandwidth properties for intra- and inter-node communications separately. We used the Multi-PingPong benchmark, which is part of the Intel MPI Benchmarks (IMB) [12] suite, to establish the baseline performance characteristics. The Multi-PingPong benchmark performs the standard PingPong (using $MPI_{Send}/MPI_{Recv}$) on pairs of processors in parallel. For example, a Multi-PingPong on 16 processors performs 8 standard two processor PingPong measurements in parallel. The IMB Multi-PingPong bandwidth ([13]) is calculated relative to the amount of data sent from one processor as

$$B_{\text{MPP}} = \frac{\text{Send size}}{t} \quad (11)$$

The second test used was the All-to-All bandwidth test in which we used the All-to-All implementation of our FFT test program in order to measure the All-to-All bandwidth.
for a single node and across two nodes. In order to compare the two bandwidth values we define the All-to-All bandwidth as

\[ B_{A2A} = \frac{P \cdot \text{Send size}}{t} \]  

The All-to-All results from our test program were verified using the IMB All-to-All benchmark.

### 3.2.1 Single node (intra-node) bandwidth properties

The single node Multi-PingPong bandwidth measurements (Figure 7) show that the bandwidth is strongly correlated to the cache sizes. The cache limits (vertical lines in Figure 7) are drawn using the send size as a reference. L2 and L3 cache sizes are drawn at half the actual cache size because these caches are shared between the two cores on a POWER5 chip. The maximum bandwidth measured using the Multi-PingPong benchmark was ~3.6 GB/s for a send size of 256 KB. However, we can see that there is a large difference between maximum- and average bandwidth for send sizes fitting in either L2 or L3 cache. These two caches are shared between the two cores of a POWER5 chip. If the two processors communicating with each other in the Multi-PingPong benchmark are located on the same chip, the message exchange can happen within the shared caches. Since the difference between maximum- and average bandwidth is very large, we assume that this is normally not the case. The average bandwidth – which reaches the maximum of ~2 GB/s for send sizes of 1 MB (close to the L2/2 cache size) – is therefore of more practical significance. For very large messages the bandwidth levels out at ~1.5 GB/s.

The bandwidth drop at send sizes of 64 KB is caused by the fact that 32 KB is the size of the L1 (data-) cache. 64 KB is the first send size that does not fit in L1 and therefore requires access to the slower L2 cache. To demonstrate the influence of the eager limit, we ran the benchmark using an eager limit of 4 KB. The corresponding graph in Figure 7 shows the expected effect on the bandwidth as there is a bandwidth decrease at a send size of 8 KB due to the protocol change. However, for the eager limit setting of 64 KB (which is the recommendation in the HPCx Users Guide [5]) it is difficult to
differentiate the effects of the bandwidth drop at the L1 boundary from the effects of the protocol change.

![Figure 7: Single node Multi-PingPong bandwidth (minimum and average values) using 16 processors (8 groups of 2 processors) for various message sizes and eager limit settings of 64 KB and 4 KB](image)

The intra-node All-to-All bandwidth graph (Figure 8) shows similar properties as was discussed earlier for the Multi-PingPong benchmark. The graph in Figure 8 is drawn using the send size as a reference. The send size is the amount of data each processor sends to every other processor in the All-to-All communication. This means that the amount of data sent in the All-to-All benchmark is by a factor of 16 larger than for the Multi-PingPong benchmark shown in Figure 7. This is reflected in a shift of the cache size boundaries (vertical lines in Figure 8). The 32 KB L1 data cache for example can store the send- or receive buffer for send sizes up to 2 KB.

We observe in Figure 8 that the bandwidth drops occur before the send size of 2 KB. This might be due to the fact that the L1 cache is used for data other than the message buffer in the All-to-All communication routine. The graph for the All-to-All communication using an eager limit setting of 4 KB shows the expected effect on the individual messages as we observe a levelling effect in the bandwidth between the send size of 4 KB and 8 KB caused by the protocol change. However, we are unable to
distinguish effects from an eager limit setting of 64 KB from the effects of the L2 cache boundary as both are expected to happen between send sizes of 64 KB and 128 KB.

Figure 8: Single node All-to-All bandwidth relative to send size for eager limits 64 KB and 4 KB. Boundaries of the cache lines are shown as reference.

The maximum bandwidth for intra-node All-to-All is ~1.2 GB/s using a send size of 512 KB (8 MB data sent by each processor). Comparing the All-to-All measurements (Figure 8) with the Multi-PingPong measurements (Figure 7) at the high points around the cache limits, we can see that the All-to-All bandwidth is about ½ of the Multi-PingPong bandwidth for send sizes larger than the L1 cache. For these larger messages, the contention of the memory bus is likely to be the limiting factor. Since the All-to-All involves more messages to be sent than the Multi-PingPong we would expect this behaviour. The All-to-All graph also shows a bandwidth decrease for send sizes larger than 4 MB whereas the Multi-PingPong bandwidth increases again after the L3 drop. This indicates that All-to-All saturates the memory bus with these large messages while the Multi-PingPong benchmark does not.

For the largest message that fits in the L1 cache, the All-to-All bandwidth is only ¼ of the Multi-PingPong value. This is most likely due to latency effects – the intra-node latency is ~2 μs – which affect the All-to-All communication more because the number of messages exchanged is significantly larger.
3.2.2 Two node (inter-node) bandwidth properties

By measuring the bandwidth between two nodes, we want to establish the bandwidth limits for the row- and column transforms in the parallel 3D FFT. The Intra-node bandwidth measurements were performed on 32 processors (2 full nodes). We used the same technique to allocate pairs of nodes in different frames on the Capacity region of HPCx as were used for the latency measurements (Table 1 and Figure 6). By using this approach, we can compare the bandwidth results with the results from the latency measurements. If the network structure does affect the bandwidth, the differences should be visible in the bandwidth measurements between two nodes in different frames.

Figure 9: Two-node Multi-PingPong bandwidth between nodes in frames connected through either one NSB (f403-f403 and f403-f404) or two NSBs and one ISB (f403-f405)

Figure 9 shows the results of the IMB Multi-PingPong benchmark between nodes in different frames. The results presented are average values from a range of measurements using the same nodes on the three frames. The values measured for a single message size, especially for messages > 16 KB, show large variations due to network contention caused either by the benchmark itself or by other applications running on other nodes on the same NSB or ISB.
We have seen in Figure 7 that the intra-node bandwidth drops considerably at the boundaries of the different cache levels. This effect is less pronounced for the inter-node case. Although we observe performance drops close to the cache size boundaries, it is difficult to determine whether these drops are directly linked with the caches. The maximum bandwidth measured is in the range of ~270 MB/s. This bandwidth is reached by all the measured configurations for the send size of 128 KB.

We have shown (see Figure 6) that the frames f403 and f404 are connected though a single NSB (in each plane), and that frames f403 and f405 are connected through two NSBs and one ISB. It is essential to determine if the structure of the network has an effect on the bandwidth properties between nodes on frames f403, f404 and f405. We can see that there is very little difference in the Multi-PingPong bandwidth between the various frame combinations for send sizes < 8 KB. This is the send size range that is more affected by the latency difference. The measurements between f403 and f405 show that the multi switch board configuration yields slightly better bandwidth for send sizes > 128 KB. The reason for this might be that the Multi-PingPong itself causes more network contention if both nodes communicate through a single Node Switch Board. Frames f403 and f405 are connected through two NSBs and one ISB. Distributing the network traffic on more Switch Boards could therefore be advantageous in terms of bandwidth for large send sizes.

Considering the scattering of the benchmark data, it is difficult to establish whether the network structure has an influence on the bandwidth or not. However, the differences in bandwidth measured between the single NSB combinations compared with the NSB/ISB/NSB combination, indicate that we may achieve better bandwidth by using nodes that are connected through more than one NSB.

Figure 10 shows the All-to-All bandwidth graphs for the same set of two-node configurations used in the Multi-PingPong measurements. As we have seen from the Multi-PingPong results, there is very little difference in the All-to-All bandwidth between the different node combinations except for send sizes > 1 MB. For small send sizes however, the latency differences (see Figure 6) should have an impact on the bandwidth between nodes on different Node Switch Boards. This is not the case.
Figure 10: Two-node All-to-All bandwidths between nodes in frames connected through either one NSB (f403-f403 and f403-F404) or two NSBs and one ISB (f403-f405). Table 2 shows the timings for send sizes from 2 to 512 bytes (same data as Figure 10). There is very little difference between nodes communicating through one NSB (f403-f403 and f403-f404) and nodes communicating through two NSB and one ISB (f403-f405) for send sizes up to 256 KB. For send sizes between 512 KB and 2 MB we observe the same advantage for the f403-f405 combination we have discussed for the Multi-PingPong benchmark. The results for send sizes > 4 MB however, show the best performance for the configuration with both communicating nodes in the same frame.

<table>
<thead>
<tr>
<th>Send size (bytes)</th>
<th>Communicating nodes in frames</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f403-f403</td>
</tr>
<tr>
<td>2</td>
<td>30.86</td>
</tr>
<tr>
<td>4</td>
<td>31.32</td>
</tr>
<tr>
<td>8</td>
<td>34.05</td>
</tr>
<tr>
<td>16</td>
<td>36.04</td>
</tr>
<tr>
<td>32</td>
<td>37.96</td>
</tr>
<tr>
<td>64</td>
<td>43.21</td>
</tr>
<tr>
<td>128</td>
<td>62.65</td>
</tr>
<tr>
<td>256</td>
<td>83.59</td>
</tr>
<tr>
<td>512</td>
<td>114.92</td>
</tr>
</tbody>
</table>

Table 2: Two-node All-to-All timings (in μs) for various node combinations relative to send size for send sizes between 2 and 512 bytes.

It should to be noted that the link between send size and bandwidth for the All-to-All communication is based on the assumption that the MPI library does send the data as
individual messages over the network. This might or might not be true for an SMP aware MPI implementation. It is possible that the MPI library packs messages for distinct processors in larger messages targeted for distinct nodes, and unpacks and distributes the data within the target node. This would result in larger messages sent over the network and would also minimize latency effects associated with small messages. The All-to-All timings are, compared to the Multi-PingPong measurements, much more stable. This is most likely due to the collective nature of the All-to-All communication which has a levelling effect on the measurements. The maximum bandwidth for the two-node All-to-All is ~260 MB/s for send sizes of 512 KB. This is very close to the two-node bandwidth measured using the IMB Multi-PingPong benchmark. The factor of ½ we have observed in the intra-node measurements does not appear in the two-node measurements. This is most likely because in the intra-node case the network is the limiting factor. Both benchmarks manage to saturate the network and therefore we do not observe the factor we have measured in the intra-node benchmark.

As we have seen from the Multi-PingPong measurements, there seems to be no significant difference in bandwidth between the different layers of the HPS interconnects. As we have concluded for the Multi-PingPong case before, we can see no advantage present in placing tasks on nodes in the same frame or in frames connected through a single NSB. However, we have seen that for large send sizes the opposite is true. Our measurements show better bandwidth performance for large send sizes if the nodes are connected through more than one NSB. For send sizes < 1 MB the network has the same inter-node All-to-All bandwidth properties irrespective of the location of the communicating nodes in the network. The considerable bandwidth difference between intra- and inter-node All-to-All communications however, provides some potential for optimizations of the 3D FFT using 2D data decomposition.

3.3 Task placement for parallel 3D FFT on HPCx

The allocation of tasks is done per node (multiples of 16 processors). Which task is placed on which physical processor within a node is determined by the operating system. Although explicit task binding is possible on HPCx, we only discuss results without explicit task binding using the affinity settings recommended in the HPCx
Users Guide [5]. We can assume that tasks are migrated between processors by the operating system. A batch job can ask for a specific node by using the LoadLeveler requirements keyword to specify the machine name(s) of the required node(s). This approach was used to gather intra-node latency and bandwidth measurements which have been presented above. Since we use a virtual two dimensional processor grid to compute the FFT of the 3D input data we have a certain degree of freedom on how we map the rows and columns of the virtual processor grid on the processors in the SMP nodes.

It is important to understand the different topologies used in the mapping process. The entire process is explained using an example mapping on 32 processors ($P = 32$), an $8 \times 4$ ($P_x = 8, P_y = 4$) virtual processor grid and an $8^3$ problem size ($N_x \cdot N_y \cdot N_z = 8^3 = 512$ values). The physical processors are structured in SMP nodes with 16 processors per node ($P_{No} = 16$) as described in chapter 3.1. The virtual processor grid is a logical structure which arranges the physical processors into a virtual two dimensional grid. Each processor is assigned a unique location in this grid by means of Cartesian coordinates. The virtual processor grid is created using the \texttt{MPI\_Cart\_create} API. The first step in the mapping process is shown in Figure 11.

![Virtual processor grid diagram](https://via.placeholder.com/150)

Figure 11: Mapping of 32 physical processors in two SMP nodes to virtual two-dimensional Cartesian processor grid

There is no specific order in which the MPI library has to assign ranks to processors in an SMP cluster. For this discussion we assume that ranks are assigned in such a way that MPI ranks of the processors in the same node are contiguous, with the first processor on each node having a rank number which is a multiple of the number of processors per SMP ($P_{No}$). Our tests to determine the rank to node mappings (using
MPI_Get_processor_name) show that this is the standard behaviour of the IBM MPI library on HPCx.

Figure 12: Mapping of the 2D virtual processor grid to the 3D data and creation of row- and column communicators for the All-to-All communication from the Cartesian grid

The next step is to map the 2D virtual processor grid to the 3D data grid (Figure 12 left). For this mapping the Cartesian grid is rotated to match the coordinate system of the 3D data introduced in Figure 2. Each of the 32 processors is mapped to a 3D sub-array of size $[1,2,8]$ of the $8^3$ data array. As discussed before the 3D data array used C conventions with the z-dimension contiguous in memory. The last step in the mapping process is to create the row- and column communicators for the All-to-All data exchange using the MPI_Cart_sub API (Figure 12 right). For our example, this creates 8 independent row communicators ($R_0$ to $R_7$) for the All-to-All within the rows of the virtual processor grid with each communicator spanning 4 processors ($P_r = 4$) and 4 independent column communicators ($C_0$ to $C_3$) with each spanning 8 processors ($P_c = 8$) for the data exchange within the columns.

3.3.1 Problem size, local memory and send size considerations

Assuming that for a given problem size $N^3$ (using the restriction $N=N_x=N_y=N_z$) the 1D decomposition using $P = N$ processors is the fastest possible parallel FFT, we are mainly interested in cases where $P > N$. For these cases, the 2D decomposition is the
only option to further improve scalability to a large number of processors. It is useful to examine the amount of data stored on each processor and the send sizes involved in All-to-All communications for cases where $P > N$.

Assuming we use double precisions floating point numbers, each complex number, corresponding to two double values in C, is 16 bytes long. The total amount of memory used for a problem of size $N_x \cdot N_y \cdot N_z (= N^3$ in our case) is then given by

\[(13) \quad D_T = 16 \cdot N_x \cdot N_y \cdot N_z\]

The amount of data stored on each processor is then given by

\[(14) \quad D_p = \frac{D_T}{P}\]

Using equations (13) and (14), Table 3 shows the amount of memory used on each processor for various values for $P$ and $N$, where $P > N$. The shaded area marks $N / P$ combinations where a 1D data decomposition is applicable and the use of a 2D data decomposition has no advantages.

<table>
<thead>
<tr>
<th>Dim. size</th>
<th>Number of processor $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>512</td>
<td>2048 KB</td>
</tr>
<tr>
<td>256</td>
<td>256 KB</td>
</tr>
<tr>
<td>128</td>
<td>32 KB</td>
</tr>
<tr>
<td>64</td>
<td>4 KB</td>
</tr>
<tr>
<td>32</td>
<td>512 B</td>
</tr>
<tr>
<td>16</td>
<td>256 B</td>
</tr>
<tr>
<td>8</td>
<td>128 B</td>
</tr>
</tbody>
</table>

Table 3:  Amount of data stored on each processor locally ($D_p$) for $P > N$ and problem sizes $N^3$ (with $N=N_x=N_y=N_z$)

Send sizes (message sizes) used in the All-to-All communication calls depend on the number of processors in the row- and column communicators ($P_r$ and $P_c$). Since all the data stored on each processor ($D_p$) has to be transposed in both All-to-All communications, the send size for the All-to-All communications within the row communicators is given by

\[(15) \quad S_r = \frac{D_p}{P_r}\]
The send size for All-to-All communications within the column communicators is defined accordingly as

\[(16) \quad S_c = \frac{D_p}{P_c}\]

As an example the range of send sizes \((S_r\) and \(S_c\)) for all possible configurations of the virtual processor grid for \(P=256\) and \(N=128\) is shown in Table 4.

<table>
<thead>
<tr>
<th>(P_r \times P_c)</th>
<th>Send size within row communicator</th>
<th>Send size within column communicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 128</td>
<td>65536</td>
<td>1024</td>
</tr>
<tr>
<td>4 x 64</td>
<td>32768</td>
<td>2048</td>
</tr>
<tr>
<td>8 x 32</td>
<td>16384</td>
<td>4096</td>
</tr>
<tr>
<td>16 x 16</td>
<td>8192</td>
<td>8192</td>
</tr>
<tr>
<td>32 x 8</td>
<td>4096</td>
<td>16384</td>
</tr>
<tr>
<td>64 x 4</td>
<td>2048</td>
<td>32768</td>
</tr>
<tr>
<td>128 x 2</td>
<td>1024</td>
<td>65536</td>
</tr>
</tbody>
</table>

Table 4: Send sizes (in bytes) for row- and column All-to-All for 256 processors and problem size \(N=128\) for all possible virtual processor grid layouts

### 3.3.2 Properties of All-to-All within the row- and column communicators

Looking at the location of the physical processors in the row-/column communicators for the chosen virtual grid layout (Figure 13), we make the following observations:

1. The size of the first dimension \((P_y)\) in the virtual grid defines the number of row communicators and the number of processors in each column communicator \((P_c = P_y)\).

2. The size of the second dimension \((P_r)\) in the virtual grid defines the number of column communicators and the number of processors in each row communicator \((P_r = P_y)\).

3. All-to-All communications within the row communicators can be done without sending data over the network (intra-node All-to-All), as long as the number of processors in each row communicator is smaller or equal the number of processors per SMP \((P_r \leq P_{No})\).
4. All-to-All communications within the column communicators always involves the network.

Figure 13: Location of physical processors in row-/column communicator for 32 processors using a 4x8 virtual processor grid

Based on observation 3 and 4 and the difference between intra- and inter-node All-to-All bandwidth we discussed in section 3.2, we expect that all virtual processor grid layouts which use only intra-node communications for the All-to-All within the rows, layouts with $P_r \leq P_{No}$, result in a better overall performance than mappings with $P_r > P_{No}$.

We will now look at the two communication phases in more detail. For the analysis we use average time (as defined in section 2.3.2) from a set of test runs on 256 processors, using all the possible virtual processor grid layouts. We have selected the average iteration times for this comparison because we want to use timing values closest to what can be expect in a real application. Our measurements show that using larger problem sizes on the nodes of the Capability region lead to results which are less affected by noise on the machine and consequently more stable and reproducible. The minimum times (see 2.3.2) are shown as reference.

3.3.3 All-to-All within the row communicators

Let us consider the communication within the row communicators first. From observation 3 and the results of the bandwidth measurements of intra- and inter- node All-to-All communications (see Figure 8 and Figure 10), we can assume that All-to-All within the row communicators performs best with mappings having $P_r \leq P_{No}$.
We can see from Figure 14 that our assumption is correct for all measured problem sizes (details in Appendix B). The vertical line marks the SMP boundary with virtual processor grid layouts with $P_r \leq P_{No}$ on the right hand side. We can also see that the communication time decreases, as the value for $P_r$ gets smaller. Since the send size is inverse proportional to the number of processors in the row communicator (see eq. (15)), the send sizes become larger as $P_r$ gets smaller. We have seen in Figure 10 that the intra-node All-to-All performance is tightly coupled with the send size due to cache effects. Using this relation, we can explain the increase in the All-to-All time between the values for $P_r=4$ and $P_r=2$ for $N=128$ in Figure 14. Using eq. (13), (14) and (15) we get a send size of 32 KB for $P_r=4$ and a send size of 64 KB for $P_r=2$. The 32 KB L1 data cache is not shared between processors and therefore one 32 KB message for $P_r = 4$ fits into the L1 cache on each processor and can be sent without touching other caches. This is not the case however for the 64 KB message size for $P_r = 2$. We assume that this may be the reason for the slowdown as this is the only case in Figure 14 where the row communication time increases for both the minimal and the average iteration time.
3.3.4 All-to-All within the column communicators

Let us now consider the more complex communication within the column communicators. Since the All-to-All exchange within the columns always includes the network (see observation 4 above), we need to examine how the layout of the virtual processor grid affects the amount of data sent over the network. We first investigate a two-node layout with \( P = 32 \) and a 16x2 virtual processor grid layout \((P_r=16 \text{ and } P_c=2)\). The mapping of the processors in the first row- and column communicator \((R_0 \text{ and } C_0)\) on the two SMP nodes is shown in Figure 15. As illustrated above, the All-to-All between the 16 processors within the two independent row communicators \(R_0\) and \(R_1\) does not involve the network.

![Figure 15: Location of processors in the first row communicator \(R_0\) and the first column communicator \(C_0\) on 32 processors using a 16x2 virtual processor grid](image)

An All-to-All within \(C_0\) requires each processor to send \( S_c = \frac{D_r}{P_c} = \frac{D_r}{2} \) to all processors in the same column communicator. Let us consider only the communication in one direction, from Node_0 to Node_1. There is one processor on Node_0 and one processor on Node_1 within \(C_0\). This means that \(1 \cdot S_c\) data has to be sent over the network for the All-to-All in \(C_0\). The same happens for all 16 column communicators in parallel. The total amount of data sent from Node_0 to Node_1 \((D_{NC})\) is given by

\[
D_{NC} = 16 \cdot \frac{D_r}{2}
\]

Using the definition of \(D_P\) (see eq. (14)) this results in

\[
D_{NC} = 16 \cdot \frac{D_r}{P \cdot P_c} = 16 \cdot \frac{D_r}{32 \cdot 2} = \frac{D_r}{4}
\]
For our example layout, \( \frac{1}{4} \) of the total data associated with a given problem size has to be sent over the network from Node\(_0\) to Node\(_1\). Simultaneously, the same amount of data has to be sent in the other direction, from Node\(_1\) to Node\(_0\).

If we change the virtual grid layout to \( P_r = 8 \) and \( P_c = 4 \) (Figure 16) we have the situation that 2 of the 4 processors within \( C_0 \) are on Node\(_0\), and that the other 2 processors are on Node\(_1\). For an All-to-All within \( C_0 \) each of the 2 processor on Node\(_0\) sends \( 2 \cdot S_c \) data over the network to the 2 processors on Node\(_1\). This again happens in the 8 column communicators in parallel resulting in

\[
D_{NC} = 8 \cdot 2 \cdot 2 \cdot S_c = 32 \frac{D_T}{P \cdot P_c} = 32 \frac{D_T}{32 \cdot 4} = \frac{D_T}{4}
\]

The All-to-All in columns for the 8x4 mapping also require messages to be sent to processors in the same node. Since an MPI All-to-All communication call does not finish until the data to and from all processors is sent and received, the total time of the All-to-All will be dominated by the time it takes to communicate through the network with the processors on the other node.

From eq. (18) and (19) we can see that the amount of data sent over the network for the All-to-All in columns using 2 nodes does not depend on the layout of the virtual processor grid. We can therefore develop a model that ignores the processors inside an SMP node and only considers the two (virtual) nodes and the network (Figure 17).
Using this approach we can say that the amount of data stored on VN0 is given by

\[ D_{VN} = \frac{D_T}{2} \]  

Irrespective of the virtual processor grid layout half of this data has to be sent to VN1. The amount of data sent over the network at the bisection \( D_{BI} \) is then given by

\[ D_{BI} = \frac{D_{VN}}{2} = \frac{D_T}{4} \]  

Using this bisection approach we can now map larger number of nodes to the two virtual nodes. For \( P=64 \) using a \( P_r=16 \) and \( P_c=4 \) (16x4) virtual processor grid layout the mapping is shown in Figure 18.

---

Figure 17: Virtual node mapping of the column communicator for 32 processors (2 nodes) and an 8x4 virtual processor grid layout

Figure 18: Virtual node mapping for the column communicator for 64 processors (4 nodes) and a 16x4 virtual processor grid layout
The virtual node mapping for 64 processors (Figure 18) has the same properties in terms of $D_{Bl}$ as the virtual node mapping for 32 processors. The All-to-All communication between the two nodes in each virtual node also involves the network. However, the amount of data sent over the network between VN$_0$ and VN$_1$ is always larger (double the amount if there is at least one processor per column communicator on every node) than the amount of the data sent between the two nodes within VN$_0$ or VN$_1$. The bisectional bandwidth of the network between the two virtual nodes is therefore the dominant factor for the time spent in the All-to-All communication within the column communicators.

Figure 19 shows the times measured for All-to-All within the columns using the same set of measurements as for the row communications above (details in Appendix B). Since the amount of data sent over the network remains constant for a given $N$, we can say that a lower communication time translates to a better bandwidth. From this point of view the results in Figure 19 matches to a large extent our expectations from what we have seen from the inter-node bandwidth measurements (Figure 10). The column communication is faster for small values of $P_c$, resulting in larger send sizes.

![Figure 19: Average and minimum time (in s) for All-to-All communications within the column communicators on 256 processors for various problem sizes and virtual processor grid layouts](image-url)
The differences between the average time (solid lines) and the time for the fastest iteration (dotted lines) show that there is a considerable amount of scattering involved in the measurements. This is obvious from the two results at $P_c=32$ for $N=128$ and $N=256$, which seem to be out of order, and are not confirmed in the minimum time graphs. Repeated runs of the test program have shown the same type of unexpected slow-downs for all problem sizes and for different virtual grid layouts. Whether these network stalls are caused by other applications contending for the network, or by saturation effects caused by the test application itself, is difficult to determine.

By applying the virtual node mapping model, introduced in Figure 17 and Figure 18, to the 256 processor partition and using the timing data for the column communications ($t_c$) presented in Figure 19, we can calculate the bisectional bandwidth ($B_{BI}$) for a given problem size and virtual processor grid layout by

$$B_{BI} = \frac{D_{BI}}{t_c} = \frac{D_T}{4\cdot t_c}$$

The results for the bisectional bandwidth calculations using the average times (solid lines in Figure 19) are shown in Figure 20.
All the bandwidth measurements we presented so far have show that the bandwidth on the HPCx interconnect increases with the amount of data sent in one message (send size). We can see from Figure 21 that the 128x2 virtual processor grid mapping in our model is equivalent to a Multi-PingPong between VN0 and VN1.

![Virtual node mapping of the column communicator for 256 processors (16 nodes) and a 128x2 virtual processor grid layout](image)

Figure 21: Virtual node mapping of the column communicator for 256 processors (16 nodes) and a 128x2 virtual processor grid layout

For our model to be coherent with what we have seen so far, the bandwidth between the two virtual nodes should have the same properties as the inter-node Multi-PingPong bandwidth measurements we presented in Figure 9. This is not the case. We do not have any data from our previous measurements that would explain the difference between the bandwidth for the problem size 128^3 and 512^3 on the 182x2 virtual processor grid. We also have not seen any indications that would explain the large fluctuations in the bandwidth graphs. A word about the reproducibility of the bisectional bandwidth graphs. We have only a limited number of measurements on 256 processors. As we have seen for all measurements so far, there are significant variations in the times we measure between different runs that are unavoidable. Although the individual measurements differ from each other, the general behaviour is quite stable. The large differences in bisectional bandwidth for the 128x2 layout and the peaks at either the 16x16 layout or at the 8x32 layout are common to all measurements (see the additional graph in Appendix A). From that we can only conclude that the bisectional bandwidth model is not applicable for the All-to-All within the columns. Within the limited time available for this project we were not able to define a model explaining the behaviour of the All-to-All communication within the columns on the HPCx network.
3.3.5 Combination of All-to-All in Row- and column communicators

We have seen in section 3.3.3 that All-to-All within the row communicators performs best when we used a virtual processor grid layout with \( P_r \leq P_{No} \). From the analysis of the All-to-All within the column communicators (Figure 19) we have seen that the virtual processor grid layouts with \( P_r \leq P_{No} \) (and therefore \( P_c > P_{No} \)) result in a less than optimal performance. Optimizing the virtual processor grid for one All-to-All communication type will result in bad performance for the other All-to-All communication. Considering the combined row- and column (Figure 22) communication times we try to establish guidelines on which optimization is more likely to result in the best overall performance.

The total communication for the average times shown in Figure 22 (details in Table 6 in Appendix A) show that for the problem sizes \( N=512 \), \( N=256 \) and \( N=128 \), the best total communication time is at \( P_r=8 \) or \( P_r=16 \). Considering the minimum timings, printed as dotted lines, we can see that for problems \( \geq 256^3 \) the layouts with \( P_r \leq P_{No} \) are almost identical.

![Figure 22: Average and minimum total communication (row- + column communication) time (in s) for All-to-All communications on 256 processors for various problem sizes and virtual processor grid layouts](image)

For \( N=128 \) on the other hand, the 16x16 layout is clearly the best performing mapping. These results are within our expectations given that the intra-node All-to-All performance is considerably better than the inter-node All-to-All.
Since we are interested in problem sizes where the 2D data decomposition improves scalability ($P > N$) we have a closer look at the problem size $N=64$. Figure 23 shows the total communication time as well as the individual times for row- and column All-to-All for $N=64$ on 256 processors. We can see that for this problem size the best performance was achieved using a 32x8 virtual processor grid ($P_r=32$ and $P_c=8$). This is surprising as for this virtual processor grid layout the network is involved for the All-to-All in the rows and in the columns. However, this behaviour is not completely stable for every run of the same configuration, but can be reproduced for the majority of our measurements.

![Graph showing communication times](image)

Figure 23: Details of row- and column All-to-All communication times for $N=64$ on 256 processors using various virtual processor grid layouts

The increase in overall communication time for the layouts with $P_r \leq 16$ is caused by the poor performance of the All-to-All within the columns. The send sizes for the column All-to-All for these layouts is between 256 bytes for the 64x4 layout and 1 KB for the 16x16 layout. The poor performance for sending these small messages over the network does outweigh the performance advantage of a purely intra-node All-to-All within the rows in this particular case. Using the minimum time from Figure 22 as an additional reference (Table 6 in Appendix A), we observe that the 32x8 layout leads to the fastest minimum time for the $64^3$ problem size as well. This shows that the average times do not suffer from exceptionally slow iterations.

We see these results as an indicator that for large scaling factors, a layout using $P_r > P_{N\sigma}$ can achieve better performance than all layouts using $P_r \leq P_{N\sigma}$. We note in this
context that the number of valid mappings decreases as we use more processors for a given problem size. Although we can use $N^2$ processors to solve a problem of size $N^3$, there is only a single valid virtual processor grid layout ($P_r = P_c = N$) if we do so.

Based on our measurements, we can conclude that for most problem sizes a virtual processor grid layout with $P_r = P_{No}$ is generally a good choice. Since the intra-node All-to-All performance varies with the send size, choosing a layout with $P_r = \frac{P_{No}}{2}$ can result in faster row communication times. For large scaling factors, where $P \geq 4 \cdot N$, we have seen that a grid layout with $P_r = 2 \cdot P_{No}$ can result in the fastest total communication time.

### 3.3.6 Node placement on the HPC$x$ network

So far we have not consider the structure of the HPS interconnect for our investigation of the column All-to-All on 256 processors. We have seen from the inter-node Multi-PingPong (Figure 9) and the inter-node All-to-All (Figure 10) benchmarks, that the bandwidth seems to remain largely unaffected regardless whether the two nodes are connected through a single NSB or through an NSB-ISB-NSB combination with a slight advantage for the NSB-ISB-NSB combination (see Figure 5). Running our test program on 256 processors, we can extent the investigations from two nodes to two full frames.

We are able to allocate all 16 nodes in 2 frames on the Capability region of HPC$x$ by using the same technique used for the latency and inter-node bandwidth tests. Using this technique, we can run the test program on two frames connected through a single NSB, two frames connected via two NSBs and an ISB and 16 nodes on random frames (no requirements specified). We can then apply the bisectional bandwidth (see eq. (22)) for All-to-All within the column communicators as a metric in order to compare the different node placements. Although the bisectional bandwidth model is not applicable to explain the behaviour of the All-to-All within the columns, we can still use it as a relative measure between different frame configurations.

We use the average time for the column communication as a metric for the sustained bandwidth (avg), and the minimum time as a metric for the maximum (best) bandwidth between the nodes of the two frames. However, using the best iteration from a set of up
to 500 does not imply that this is the fastest possible time. Allocating all nodes in two frames does not guarantee that there is no contention from other applications on the NSBs. Since we use All-to-All communications intensively, any contention is most likely caused by the test application itself. We use the problem sizes $N=256$ and $N=128$ for our measurements.

Figure 24: Bisectional bandwidth (average and best) for All-to-All communication within columns using 256 processors on 2 full frames connected through a single NSB.

Figure 24 shows the bisectional bandwidth graphs for the average column communication time (solid lines) and the column communication time of the fastest iteration (dotted lines) for the frames connected through a single NSB.

Figure 25 shows the corresponding graphs for nodes connected through two NSBs and one ISB. The comparison shows that, although the maximum values (dotted lines) are in a similar range, the bisectional bandwidth for the 2 NSB case seems to be slightly better. This is especially true for $N=256$ where the measured bisectional bandwidth is considerably better, with the exception of the 8x32 virtual processor grid layout. The measurement for this configuration is the worst for both problem sizes.

Both configurations show the same tendency of decreasing bisectional bandwidth with small message sizes, but this behaviour is more pronounced for the configuration using a single NSB.
Figure 25: Bisectional bandwidth (average - and best iteration times) for All-to-All communication within columns using 256 processors on 2 full frames connected through 2 NSBs and 1 ISB.

We would like to note that the measurements are taken from a single measurement. This is because asking for 2 full frames at specific locations on HPCx can take up to two days for a job to be started depending on how busy the machine is.

Figure 26: Bisectional bandwidth (average - and best iteration times) for All-to-All communication within columns using 256 processors using 16 random nodes (default behaviour on HPCx)

We use the bidirectional bandwidth metric on a set of measurements using the default node allocation with no requirements specified. (Figure 26). We can see that the
general structure of the bisectional bandwidth graphs is the same as shown in Figure 25, and that the performance drop is at the virtual processor gird layout 16x16 instead of 8x32. This behaviour is, as we have discussed before, difficult to explain but we assume it is more likely that the drops are caused by random stalls due to contention in a network component.

From our results of running the FFT test program on full frames at different locations of the Capability region of HPCx we can see no advantages in placing the processes on specific nodes or frames within the HPS interconnect. However, we do observe a difference when we use all 16 nodes in 2 frames which are connected through a single NSB. In this case we see a decrease of the bisectional bandwidth, especially for large send sizes. This is an indication that the default node allocation scheme used on HPCx provides the best bandwidth properties for the All-to-All within the columns for the 3D parallel FFT using 2D data decomposition.
Chapter 4

Performance of parallel 3D FFT on HPCx

4.1 Factors influencing the performance of the parallel 3D FFT

In Chapter 3 we discussed the properties of the All-to-All communication for the 3D parallel FFT using 2D data decomposition in the previous chapter. However, the All-to-All communication is not the only factor affecting the performance of the 3D parallel FFT.

Figure 27: Time distribution between the 9 steps of the parallel forward parallel 3D FFT using 2D data distribution on 256 processors in an 8x32 virtual processor grid for a $128^3$ problem size using MPI_Alltoall and manual pack-/unpack routines

Figure 27 shows an example of the relative time distribution between the 9 steps (see section 2.3.2) for a single forward parallel 3D FFT using 2D data decomposition. The sample data is taken from a run on 256 processors for a $128^3$ problem size using an
8x32 virtual processor grid, which is the best performing grid layout for this problem size. We can see that only ~20% of the time for a forward 3D FFT is spent in the Fourier Transformation code. Selecting the best virtual processor grid layout is one way to increase this ratio. In this particular example, more than half of the time was spent in the two communication steps. However, when implementing the parallel 3D FFT, we can choose from various options on how to implement each of the steps shown in Figure 27. In this chapter, we want to look at these options and illustrate how they affect the overall performance of the parallel 3D FFT.

4.1.1 MPI All-to-All APIs

The MPI standard [11] defines two different All-to-All methods, `MPI_Alltoall` and `MPI_Alltoallv`. `MPI_Alltoall` is the simpler, but more restrictive of the two communication methods. All measurements we discussed in Chapter 3 are based on `MPI_Alltoall`.

`MPI_Alltoall` is used to send the same amount of data to every processor participating in the communication. Unless we use derived data types, the data being sent to every processor has to be contiguous in memory. Deferring the discussion of derived data types to section 4.1.2 below, we briefly discuss the implications of this restriction. Considering an All-to-All between 4 processors, a send size of 4 and a send data type with size of 1 byte, the send- and receive buffers used for the `MPI_Alltoall` call have to be of size 4·4·1 bytes. Each processor sends 4 bytes to all processors in the communicator (including itself). Each processor sends bytes 0-3 of its send buffer to the processor with rank 0 (within the communicator used for the All-to-All), bytes 4-7 send to processor with rank 1 and so on. In case of the parallel 3D FFT using 2D data decomposition, where each processor holds a pencil shaped part of the entire problem, the data intended for the processors in the same row- or column communicator is not contiguous in memory (unless \( D_P \) is of size \([1,1,P_r] \), requiring \( P_r = P_c \)). This means that the data has to be rearranged (packed) from the work buffer to the send buffer before calling the `MPI_Alltoall` API. When the call returns, the data is arranged in blocks of 4 bytes, according to the amount of data received from each processor, in the receive buffer. We have to rearrange (unpack) the data back to the structure required by the
FFT library. This last step is only necessary if we do not use strided FFT which will be discussed in section 4.1.3 below.

With respect to the 3D parallel FFT, using \textit{MPI\_Alltoall} imposes restrictions on the input data. Since each processor sends and receives the same amount of data, the three dimensions of the input data have to be divisible by the number of processors in the two dimensions of the virtual processor grid ($P_r$ and $P_c$).

The vector variant of the MPI All-to-All communication routines, \textit{MPI\_Alltoallv}, lifts some of the restrictions of \textit{MPI\_Alltoall} which we just discussed. Instead of specifying one send and one receive count, \textit{MPI\_Alltoallv} allows us to specify vectors for displacements and counts for the All-to-All communication. The four vectors for the send/receive counts and send/receive displacements are of the same length as there are processors participating in the collective communication. By using \textit{MPI\_Alltoallv} we can send a different amount of data to each processor and receive a different amount of data from each processor. Although we can specify send lengths and displacements into the send/receive buffer for every processor, the data intended for one processor still has to be contiguous in memory (unless we used derived data types). Using \textit{MPI\_Alltoallv} as the All-to-All API for the 3D FFT allows us to handle problems with uneven load distributions.

We compared the performance of the two All-to-All APIs in two different ways. First we used the Intel MPI Benchmarks [12] to measure the baseline performance for the intra-node case (16 processors) and for the inter-node case for two and four nodes. For the second test we used our FFT test program which can be configured to use either \textit{MPI\_Alltoall} or \textit{MPI\_Alltoallv} as the communication routine. Using a synthetic benchmark allowed us to compare the performance for specific send sizes whereas the comparison using the FFT test program is restricted to the send sizes given by the $N^3$ problem sizes (see chapter 3.3.1). The test program is not designed to use \textit{MPI\_Alltoallv} with arbitrary dimension sizes.

Figure 28 shows the results of the comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} for send sizes of 4 bytes to 8MB on a single 16 processor node. The bandwidth graph shows the cache boundaries as vertical lines. Interestingly, \textit{MPI\_Alltoallv} seems to take no advantage of the L1 cache as the performance of \textit{MPI\_Alltoallv} in the send size
range of L1, up to 2 KB, is exceptionally bad. For messages larger than 2 KB the time

differences between the two API methods are in the range of ~2-5%, with

*MPI_Alltoallv* performing slightly better for most send sizes.

![Bandwidth comparison between MPI_Alltoall and MPI_Alltoallv on 16 processors in one node for send sizes from 4 bytes to 8 MB using Intel MPI Benchmarks](image)

Figure 28: Bandwidth comparison between *MPI_Alltoall* and *MPI_Alltoallv* on 16 processors in one node for send sizes from 4 bytes to 8 MB using Intel MPI Benchmarks

The results for 32 processors (on two full nodes) are shown in Figure 29. *MPI_Alltoallv* is considerably slower for send sizes smaller than 1 KB. Compared to *MPI_Alltoall* we observed the same poor performance for send sizes that do not fit in the L1 cache. Since the amount of data sent over the network is the same for both APIs we assume that the performance difference is linked with the L1 cache size and how *MPI_Alltoall* and *MPI_Alltoallv* are implemented. The difference between the two All-to-All routines becomes smaller for send sizes larger than 1 KB. In contradiction to what we observed for the intra-node case, *MPI_Alltoallv* is the faster routine for most send sizes larger than 8 KB.

Although the Intel MPI Benchmark averages measurements from a large number of iteration, 1000 iterations for send sizes up to 32 KB, the timings between individual runs show variations of up to 20% for the same send sizes. However, the overall trend shown in Figure 29 is consistent between measurements, especially the poor performance we observed using *MPI_Alltoallv* for small send sizes.
Figure 29: Time comparison between MPI_Alltoall and MPI_Alltoallv on 32 processors (two full nodes) for send sizes between 4 bytes and 8 MB (16 processor MPI_Alltoall times shown as reference)

The last test using the Intel MPI Benchmarks suite was performed on 64 processors (4 full nodes) on a reduced set of send sizes from 64 bytes to 512 KB. This is the relevant range of send sizes for the parallel 3D FFT. Our results are shown in Figure 30. We observed the same behaviour for small messages on 64 processors as we have seen for the 32- and 16 processor cases above. MPI_Alltoall performs considerably better for send sizes up to 512 bytes. For larger messages the difference between the two communication routines becomes smaller but it is evident that MPI_Alltoall is on average 20% faster than MPI_Alltoallv. This is again different from what we have seen for 32 processors. Although there is a variation of the absolute times we measured between different runs of the benchmark on 64 processors, the relative performance difference between MPI_Alltoall and MPI_Alltoallv is reproducible.

We have seen from all measurements that MPI_Alltoallv takes approximately the same time for all send sizes < 1 KB. This could be an indication that MPI_Alltoallv uses message aggregation and the smallest possible send size is 1 KB.
Our next step was to verify the results of the synthetic benchmarks using our parallel FFT test program. We therefore ran the test program on 64 processors to compare the results with those in Figure 30.

Figure 31 shows the total communication times (the sum of row- and column communication times) for the forward parallel 3D FFT using 2D data decomposition on 64 processors. The 64 processor test was run on an 8x8 virtual processor grid in order to have the same send sizes for the row and column communications.

The results from the measurements using the parallel FFT test program are consistent with the results using the Intel MPI Benchmark in terms of the relative performance between MPI_Alltoall and MPI_Alltoallv. We observed that MPI_Alltoall is on average 10% faster for larger messages. For small messages the advantage for MPI_Alltoall is more pronounced.

The difference in absolute times between Figure 30 and Figure 31 stems from the fact that the test program measures the sum of one All-to-All within the rows and one within the columns. Since we use an 8x8 processor grid an All-to-All within the rows results in $P_r^2$ messages sent within each of the 8 row communicators. The same is true...
for the All-to-All within the columns. The results in Figure 1 are therefore the times to send $8 \cdot P_r^2 + 8 \cdot P_c^2 = 1024$ messages whereas the IMB All-to-All results are from an All-to-All between all 64 processors resulting in $64^2=4096$ messages. Since we use an 8x8 virtual processor grid the All-to-All within the rows does not involve the network which further reduces the total time compared to the result of the IMB benchmark.

Figure 31: Total communication time (row comm. + column comm.) comparison between \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 64 processors using the parallel FFT test program for problem sizes of $8^3$ to $256^3$.

Figure 32 shows the comparison of \textit{MPI\_Alltoall} and \textit{MPI\_Alltoallv} on 256 processors by using the grid layout which resulted in the lowest total communication time. With the exception of the $256^3$ problem size \textit{MPI\_Alltoall} is the faster API. However \textit{MPI\_Alltoallv} performs ~10% better than \textit{MPI\_Alltoall} for $N=256$. The exceptional result for $N=256$ was achieved using a 16x16 virtual processor grid for \textit{MPI\_Alltoallv}, resulting in send sizes of 64 KB for both row- and column communications. For \textit{MPI\_Alltoall} a 32x8 virtual processor grid was used, resulting in send sizes of 32 KB for the row communications and 128 KB for the column communication. This is an exceptional result because the 32x8 virtual processor layout involves the network for the All-to-All within the rows. The measurement for the 16x16 layout for this particular configuration suffers from very poor performance for the All-to-All within the columns.
We can conclude that, for small send sizes up to 2 KB, *MPI_Alltoall* is the faster API for all configurations which we examined. For configurations which result in larger send sizes we have seen some exceptional cases where *MPI_Alltoallv* provides slightly better performance. However, in the majority of cases, we observed better performance using *MPI_Alltoall*. The choice of whether to use *MPI_Alltoall* or *MPI_Alltoallv* depends not only on performance considerations but also on application requirements. If an application uses arbitrary sized 3D data arrays, *MPI_Alltoallv* is the only available option. For evenly sized input arrays *MPI_Alltoall* is in most cases the better choice for the parallel 3D FFT.

### 4.1.2 Derived data types

As we have discussed in the previous section, we need to manually pack and unpack the send and receive buffers for the All-to-All routines as long as we use MPI basic data types (like *MPI_DOUBLE*) as the send and receive types. The manual packing and unpacking can be avoided by using derived data types as send- and receive data types. Derived data types can be used for both version of the MPI All-to-All communication APIs. Using derived data types leaves the task of assembling and disassembling the individual messages to the MPI library. In order to compare the performance of the
manual buffer copy with the performance of the derived data type implementation in the MPI library, we implemented a derived data type as the send type for the communication within the rows. The send data type is created in three stages. First we define a derived data type that represents a complex number as two contiguous double values (MPI_DOUBLE). Next, based on this derived data type for a complex number, we define a vector type that specifies the data to be sent to every processor. This vector type essentially describes the copy operations required to assemble the message to be sent to the receiving processor. The third step is to wrap the vector data type in a struct data type (using MPI_Type_struct). The struct data type is required for the All-to-All communication because we need a way to tell the MPI library that the vector has to be used in an overlapping instead of a sequential manner. We can now use the struct data type as the send type in the All-to-All communication routine, and provide the work buffer (the output of the first 1D FFT) directly as send buffer to the MPI_Alltoall or MPI_Alltoallv library call.

Comparing the performance of the manual buffer copy with the performance of the derived data type is only indirectly possible because the copy operations associated with the derived data type are executed within the MPI_Alltoall or MPI_Alltoallv call. We defined separate timers for the manual pack operation and for the All-to-All API call. The only way we can compare the performance of the derived data type is by comparing the sum of the time spent for the manual pack operation and the MPI All-to-All routine using the basic data types with the time for the MPI All-to-All routine using the vector data type. However, by using this approach, the difference between the two times also includes the time difference from the All-to-All communication. We have seen throughout this project that communication times can vary quite dramatically. We used timings on a single node to avoid effects from network contention in the results. Performing an All-to-All using all processors in a single node however, creates a lot of traffic on the memory subsystem which affects the timing results too. Figure 33 shows the results of the comparison between the manual buffer copy and the derived data type implementation for different send sizes on 16 processors. Our comparison (Figure 33) shows that the derived data type is faster than the manual buffer packing for all send sizes > 8 KB (details in Appendix B).
Since these results were quite surprising, we implemented an optimized packing algorithm which uses sequential reads, instead of sequential writes, to take advantage of the data prefetching to L1. Despite that improvement, the All-to-All using the struct/vector data type is still the better choice for most send sizes we measured. However, we do not claim that the improved manual buffer packing algorithm is the fastest possible solution for the POWER5 processor. We did not attempt to further improve the manual pack algorithm. In order to minimize memory and cache contention it would be better to run this comparison with only two processors in a single node participating in the All-to-All communication. Based on the result from this comparison, it would be interesting to investigate the performance of a parallel 3D FFT implementation using exclusively derived data types for the All-to-All in the communication routines.

4.1.3 Fast Fourier Transformation libraries

When it comes to the actual Fast Fourier Transformation code one can choose from a large selection of free and commercial FFT implementations. The free (and/or open source) FFT libraries are usually written in a portable way so that they can be used on a broad range of platforms. The commercial libraries are mainly written by hardware vendors and provide highly optimized FFT implementations for a specific platform and
are therefore usually not portable. Considering this basic classification, we would expect that the vendor provided libraries have a measurable performance advantage over the portable libraries.

The most widely used free FFT library is the Fastest Fourier Transform in the West (FFTW) [15] library. FFTW is the de-facto standard FFT library in the high performance computing community due to its good performance, its portability and the fact that it is free software (under the terms of the GNU General Public License). There are two versions, 2.1.5 and 3.0.1, of the FFTW library installed on HPCx. The older version is used by many codes running on the service. Unfortunately the API of version 3 is not compatible with version 2.5.1 which prevents an easy upgrade path to the new version. The reason we selected version 2.5.1 for the comparison is because it is easier to use. FFTW is implemented in C but can be used from Fortran codes as well. It uses empirical optimization techniques to tune its performance for a particular platform.

The commercial library available on HPCx is the IBM Engineering Scientific Subroutine Library (ESSL) [16]. ESSL provides not only implementations for the Fast Fourier Transformation, it also implements a wide range of other mathematical subroutines including Linear Algebra subprograms, Matrix Operations, Sorting and Searching subroutines and others. ESSL is highly optimized for the IBM POWER architecture and can be used from codes written in C, C++ and Fortran.

As we have seen in Figure 27 the time spent in the FFT library code is – depending on the problem size – in the range of 10% to 20% of the total time for one forward parallel 3D Fourier transformation. This means that performance differences between the FFT libraries have to be quite substantial to make a difference on the overall algorithm. However, it is useful to compare the performance of a vendor provided proprietary implementation with the free and portable open source library. The versions used for this comparison are FFTW 2.1.5 and ESSL 4.2.0 as installed on HPCx.

The time spent in the FFT libraries depends only on the amount of data being processed. For each of the three FFT steps, the entire local buffer on each processor has to be processed (see 3.3.1 for details on local memory sizes). We can therefore compare the performance of the FFT libraries using a single node. The library methods used for the comparison are `fftw()` for the FFTW library and `dcft()` for ESSL. Both library
methods allow for performing multiple 1D FFTs using only a single method call. This is important because each processor (usually) holds data spanning more than one row and/or column. For example, in the case of a $4^3$ problem running on 4 processors, each processor holds a 2x2x4 partition of the problem. This requires 4 independent 1D FFT calculations, each on of size 4 (see Figure 2). Using a single method call to compute all 4 1D FFTs is likely to provide better performance than 4 individual method calls. This is especially true for large problem sizes.

The FFT preparation routines, `fftw_create_plan()` for FFTW and `dcft()` with the `init` argument set to 1, are not included in the performance measurements. The FFTW plan was created using the `FFTW_MEASURE` flag which should result in the best performing plan for the given problem size and platform.

![Figure 34](image_url)

Figure 34: Performance comparison between ESSL and FFTW non-strided FFT implementations for different problem sizes (relative to the performance of ESSL).

Figure 34 shows the performance of the non-strided versions of the FFT routines (details in Appendix B). The FFT is considered non-strided if the complex numbers in the input- and output arrays are contiguous in memory (stride 1 between elements and offset 1 between individual arrays). In context of the 3D FFT using 2D data decomposition, this means that the FFT input array has to be rearranged after each All-to-All communication step to match this requirement. We can see that the ESSL implementation is generally faster for small problem sizes.
In order to have a broader spectrum of problem sizes, we ran the measurements using a number of problem sizes with $N$ not being a power of two. This means that the input array sizes for the 1D FFT that are also not powers of two. It seems that ESSL handles this type of array size better than the FFTW library. ESSL is on average 20-25% faster than FFTW. The only exception for array sizes relevant to the parallel 3D FFT is the array size of 16 K complex numbers where FFTW is slightly faster than ESSL.

Using strided FFT, we can omit the manual rearranging step after the All-to-All communication. We can use the receive buffer from the All-to-All directly as input for the FFT method. We used the strided version for the FFT in $x$-direction, the final step in the parallel 3D FFT. We can see in Figure 27 that the unpacking and rearranging step after the All-to-All in the columns is the second longest single step after the column All-to-All itself. This is because of how a C array is stored in memory. In order to transpose the $x$- and $y$-axis we need to rearrange the elements of the first column ($x$-axis) in such a way that they are contiguous in memory. The first element, $[0, 0, 0]$, is stored at source location 0 and has to be copied to target location 0. The second element, $[0, z, y]$, considering that the $z$- and $y$-axis were transposed in the row All-to-All step, is stored at memory location $y \cdot z \cdot 1$ and has to be copied to target location 1. The third element has to be copied from memory location $2 \cdot z \cdot y$ to target location 2 and so on. Applying these copy operations to the entire local buffer requires frequent access to distant memory locations, leading to poor usage of caches and prefetch streams. This explains why the rearrangement step in Figure 27 takes so long. However, we should point out that the column rearrangement code was not optimized for serial performance and most likely has further potential for improvement.

By implementing the strided FFT for the transformation in $x$-axis, we delegate the expensive operation to the FFT library. Since we measure the time for the manual rearrangement, we can analyze the two approaches by comparing the time of the strided FFT library method with the sum of the time spent to manually rearrange the data and the time spent in the non-strided FFT method. Figure 35 shows the times relative to the time of the strided FFT implementation in the ESSL.
We can see from Figure 35 (details in Appendix B) that the performance of the strided transformation of FFTW relative to that of ESSL depends highly on the problem size. The strided \texttt{fftw()} is \textasciitilde20\% faster than the strided version of \texttt{dcft()} (ESSL) for power of two problem sizes up to 16 K elements. For problem sizes \textasciitilde128 K elements, the strided versions of both libraries perform about the same. However, the non-strided version of \texttt{dcft()} using the manual rearranged data is the fastest variant up to array sizes of \textasciitilde400 K elements. This is far beyond the local data size relevant to the parallel 3D FFT using 2D data decomposition, as a \textasciitilde512^3 problem computed on 1024 processors translates into an FFT array size of 131072 complex numbers.

From our measurement we can say that the FFT implementation in the ESSL library is on average 10\%-15\% faster than the FFTW implementation for input array length of up to 128 K elements. If the input array length is not a power of two, the ESSL library is up to 40 \% faster. FFTW shows equal or slightly better performance if the input arrays are larger than 128 K elements.

### 4.2 Parallel performance and speedup

Using the information of which of the options for the individual steps yields the best performance we can now examine the scaling properties of the parallel 3D FFT using
2D data decomposition on HPCx. The results presented in this section were gathered using the following configurations:

- **MPI_Alltoall** as the All-to-All communication routine
- The non-strided version of `dcft()` (ESSL)
- The **struct/vector** derived data type for layouts which result exclusively in send sizes > 8 KB
- The manual buffer copy implementation for layouts resulting in send sizes ≤ 8 KB

The virtual processor grid layout was chosen according to our conclusions from section 3.3.5. We chose either $P_{No} \times \frac{P}{P_{No}}$ or a $\frac{P_{No}}{2} \times \frac{2P}{P_{No}}$ virtual processor grid. For small problem sizes and large scaling factors ($N \geq 4P$) we also considered the virtual processor grid mapping $2 \cdot P_{No} \times \frac{P}{2 \cdot P_{No}}$ which requires both All-to-All communications, within the columns and within the rows, to send messages over the network.

![Figure 36](image)

**Figure 36:** Performance measurements for the parallel 3D FFT using 2D data decomposition for problem sizes from $32^3$ to $512^3$ on up to 1024 processors (average times)
Figure 36 shows the results of the performance measurements for problem sizes from $32^3$ to $512^3$ (details in Appendix B). We used the average time for one forward parallel 3D FFT using 2D data decomposition (except for $P=1$ and $P=2$). We can see that the program performs reasonably well for problem sizes with $N \geq 128$ on up to 1024 processors.

For problem sizes with $N \leq 64$ the performance improvement not as good. The speedup graph (Figure 37) using the same data as Figure 36 shows the limited scalability for small problem sizes more clearly. For $N=32$ the speedup graph already flattens using 16 processors. Although we still see an improvement using up to 256 processors, we can see that our implementation does not scale well for small problem sizes.

For problem sizes of $N=64$ the program scales reasonably well. Although we observe a small performance hit at $P=64$ the runtime still improves noticeably up to 512 processors. Considering that the 1D decomposition limits the parallelization of this problem size to 64 processors, we consider the achievable improvement a good result. As we have seen before, the 3D FFT using 2D data decomposition scales well for problem sizes $\geq 128^3$.

The results in Figure 36 and Figure 37 were achieved using the virtual processor grid layouts in Table 5. For $P=128$ and $N=256$ the 4x32 virtual processor grid layout
resulted in the best performance. This layout was only considered because we observed
an unusual performance drop in the speedup graph at this particular combination.

<table>
<thead>
<tr>
<th>Processors</th>
<th>32$^3$</th>
<th>64$^3$</th>
<th>128$^3$</th>
<th>256$^3$</th>
<th>512$^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>8 x 8</td>
<td>16 x 4</td>
<td>16 x 4</td>
<td>8 x 8</td>
<td>16 x 4</td>
</tr>
<tr>
<td>128</td>
<td>16 x 8</td>
<td>16 x 8</td>
<td>8 x 16</td>
<td>4 x 32</td>
<td>8 x 16</td>
</tr>
<tr>
<td>256</td>
<td>16 x 16</td>
<td>16 x 16</td>
<td>16 x 16</td>
<td>8 x 32</td>
<td>8 x 32</td>
</tr>
<tr>
<td>512</td>
<td>16 x 32</td>
<td>16 x 32</td>
<td>32 x 16</td>
<td>16 x 32</td>
<td>8 x 64</td>
</tr>
<tr>
<td>1024</td>
<td>32 x 32</td>
<td>16 x 64</td>
<td>16 x 64</td>
<td>16 x 64</td>
<td>8 x 128</td>
</tr>
</tbody>
</table>

Table 5: Virtual processor grid layout (P$_r$ x P$_c$) for $P \geq 32$ used for the performance measurements shown in Figure 36.

The 256$^3$ problem size on 128 processors results in $D_T = 2$MB. Using $P_c=32$ results in send sizes of 64 KB, which is the eager limit setting we used. Smaller values for $P_c$ therefore result in send sizes > 64 KB. Whether this is the reason for the unexpected behaviour or not is difficult to determine.

The dark shaded areas mark configurations where a row communicator spanning more than one node resulted in the best performance. For $N=32$ and $P=1024$ using a 32x32 virtual processor layout is the only available option. For $N=128$ and $P=512$ this is within our conclusions from section 3.3.5 that for cases with $P \geq 4 \cdot N$ the configuration spanning two nodes should be considered as well. We have discussed this surprising behaviour of the configuration $N=64$ and $P=256$ in Figure 23. We can see in Table 5 that for the measurements runs we used for the scaling graphs, the best grid layout for $N=64$ and $P=256$ was 16x16. This is one of the exceptions we mentioned when we discussed the stability of the $P_r>P_{No}$ performance. For this particular measurement the standard case using only intra-node communication for the row communication performed faster.
Chapter 5

Conclusions

The scaling properties of the parallel 3D Fast Fourier Transformation can be improved by applying the two-dimensional decomposition of the input data. We have shown that by using this approach, we can achieve good scalability for problem sizes between $128^3$ and $512^3$ on up to 1024 processors on HPCx. The scalability for smaller problem sizes is limited to smaller numbers of processors. For a problem size of $64^3$ we observed performance improvements on up to 512 processors.

We investigated the All-to-All communication properties of the HPS interconnect on HPCx in order to further improve the scaling properties by placing processes on specific nodes within the HPCx cluster. Communication between nodes that are connected through only one layer of the hierarchical HPS network profit from 10 % lower latency times than communications between nodes connected through all three layers of the network. Our measurements show that this advantage in latency does not translate to an improvement of the All-to-All bandwidth. We have seen that placing the processes explicitly on nodes connected through only one Node Switch Board can reduce the bandwidth for the All-to-All communication due to extensive load on a single NSB. Our results indicate that there is no advantage in placing nodes on specific frames and that the default node allocation on HPCx, resulting in random node placement, provides the best performance with respect to the All-to-All communication pattern.

We examined the influence of different 2D virtual processor grid layouts to map the 3D input data on the physical processors in the SMP nodes. We have shown that we can improve the overall performance by selecting an appropriate processor mapping. The
optimal mapping depends on the problem size and the number of available processors. We have found that mappings which only use communications within one SMP node for the All-to-All within the rows communication result, for most cases, in lower total communication times than mappings which involve the network for both All-to-All steps. To our surprise we found that mappings which result in network communications for both All-to-All steps \((P > P_{No})\) can be beneficial if we process relatively small problem sizes on large numbers of processors \((P \geq 4 \cdot N)\). The key for this unexpected behaviour lies in the All-to-All communication within the columns. Our attempt to use a bisectional bandwidth model to gain a better understanding of this communication step did not succeed. The exact behaviour of the HPCx interconnect with respect to the All-to-All communication within the columns remains unclear.

We examined the performance impact of other factors than the all important All-to-All properties of the network. Comparing the two MPI All-to-All communication routines shows that \texttt{MPI\_Alltoall} results on average in 10 % faster communication times than \texttt{MPI\_Alltoallv}. This difference is even larger for small send sizes (< 1 KB) which we typically see if we use large numbers of processors on relatively small problem sizes. We have shown that using derived data types for the All-to-All communication can yield better performance compared to the manual buffer packing and unpacking. However, this result depends highly on the efficiency of the manual buffer packing algorithm and more research needs to be done on how the two approaches compare. Finally, we showed that the 1D FFT implemented by the IBM Engineering Scientific Subroutine Library (ESSL) provides better performance than the 1D FFT of the Fastest Fourier Transform in the West (FFTW) library. This comparison was done using the easier to use version 2.1.5 of the FFTW library. The ESSL implementation is particularly faster for non power of two input array sizes. When we compared the non-strided FFT implementations, which require manual rearranging of the receive buffers from the All-to-All communication, with the strided FFT implementations, it became evident that manually rearranging the data, and using the non-strided versions, results in the best overall time for the 1D FFTs.

We have shown that applying the 2D data decomposition has great potential to improve the scalability for applications using the parallel 3D FFT on HPCx. Although we could not establish a thorough understanding on how the All-to-All communications within
the rows and columns behaves, we were able to define guidelines that lead to good
results in terms of scalability. As a next step, it would be interesting to compare the
performance of the parallel 3D FFT using 2D data decomposition with the performance
of the 3D FFT using 1D data decomposition for the cases where both decompositions
are applicable. Since the 2D decomposition allows us to perform the All-to-All within
the rows without network communication, we would expect that the penalty for using a
2D data decomposition to be smaller on a cluster of SMP nodes than on other
platforms.
Appendix A

Figure 38: Alternative bisectional bandwidth graph (using average times) for All-to-All communication within columns using 256 processors and different virtual processor grid layouts

<table>
<thead>
<tr>
<th>N</th>
<th>128x2</th>
<th>64x4</th>
<th>32x8</th>
<th>16x16</th>
<th>8x32</th>
<th>4x64</th>
<th>2x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 (avg.)</td>
<td>0.236539</td>
<td>0.203571</td>
<td>0.196895</td>
<td>0.173991</td>
<td>0.148933</td>
<td>0.172421</td>
<td>0.181578</td>
</tr>
<tr>
<td>512 (min.)</td>
<td>0.117331</td>
<td>0.127497</td>
<td>0.140470</td>
<td>0.106615</td>
<td>0.106893</td>
<td>0.097418</td>
<td>0.098201</td>
</tr>
<tr>
<td>256 (avg.)</td>
<td>0.020162</td>
<td>0.021028</td>
<td>0.019270</td>
<td>0.017233</td>
<td>0.035824</td>
<td>0.022307</td>
<td>0.017250</td>
</tr>
<tr>
<td>256 (min.)</td>
<td>0.015035</td>
<td>0.015427</td>
<td>0.013773</td>
<td>0.012852</td>
<td>0.011755</td>
<td>0.011593</td>
<td>0.011830</td>
</tr>
<tr>
<td>128 (avg.)</td>
<td>0.002655</td>
<td>0.002543</td>
<td>0.002568</td>
<td>0.001587</td>
<td>0.004142</td>
<td>0.002908</td>
<td>0.003765</td>
</tr>
<tr>
<td>128 (min.)</td>
<td>0.002535</td>
<td>0.002215</td>
<td>0.001947</td>
<td>0.001402</td>
<td>0.001508</td>
<td>0.002509</td>
<td>0.002778</td>
</tr>
<tr>
<td>64 (avg.)</td>
<td>0.000309</td>
<td>0.000289</td>
<td>0.000347</td>
<td>0.000368</td>
<td>0.000369</td>
<td>0.000369</td>
<td>0.000369</td>
</tr>
<tr>
<td>64 (min.)</td>
<td>0.000281</td>
<td>0.000278</td>
<td>0.000321</td>
<td>0.000333</td>
<td>0.000351</td>
<td>0.000351</td>
<td>0.000351</td>
</tr>
</tbody>
</table>

Table 6: Average total communication (row- + column communication) time (in s) for All-to-All communications on 256 processors for various problem sizes and virtual processor grid layouts (details to Figure 22)
Appendix B

<table>
<thead>
<tr>
<th>P</th>
<th>128x2</th>
<th>64x4</th>
<th>32x8</th>
<th>16x16</th>
<th>8x32</th>
<th>4x64</th>
<th>2x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.135643</td>
<td>0.086895</td>
<td>0.062823</td>
<td>0.018077</td>
<td>0.014101</td>
<td>0.015509</td>
<td>0.013341</td>
</tr>
<tr>
<td>256</td>
<td>0.012843</td>
<td>0.009834</td>
<td>0.007072</td>
<td>0.004506</td>
<td>0.002961</td>
<td>0.001596</td>
<td>0.001107</td>
</tr>
<tr>
<td>128</td>
<td>0.002044</td>
<td>0.001382</td>
<td>0.000874</td>
<td>0.000273</td>
<td>0.000032</td>
<td>0.000159</td>
<td>0.000315</td>
</tr>
<tr>
<td>64</td>
<td>0.000195</td>
<td>0.000111</td>
<td>0.000044</td>
<td>0.000037</td>
<td>0.000034</td>
<td>0.000034</td>
<td>0.000034</td>
</tr>
</tbody>
</table>

Table 7: Average times (in s) for All-to-All within the row communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 14)

<table>
<thead>
<tr>
<th>P</th>
<th>128x2</th>
<th>64x4</th>
<th>32x8</th>
<th>16x16</th>
<th>8x32</th>
<th>4x64</th>
<th>2x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.074225</td>
<td>0.058432</td>
<td>0.046595</td>
<td>0.011817</td>
<td>0.010636</td>
<td>0.010636</td>
<td>0.009964</td>
</tr>
<tr>
<td>256</td>
<td>0.009676</td>
<td>0.007223</td>
<td>0.004911</td>
<td>0.003484</td>
<td>0.00174</td>
<td>0.001132</td>
<td>0.000837</td>
</tr>
<tr>
<td>128</td>
<td>0.001961</td>
<td>0.001267</td>
<td>0.000769</td>
<td>0.000249</td>
<td>0.000185</td>
<td>0.000146</td>
<td>0.000288</td>
</tr>
<tr>
<td>64</td>
<td>0.000186</td>
<td>0.000109</td>
<td>0.000042</td>
<td>0.000037</td>
<td>0.000034</td>
<td>0.000034</td>
<td>0.000034</td>
</tr>
</tbody>
</table>

Table 8: Minimum times (in s) for All-to-All within the row communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 14)

<table>
<thead>
<tr>
<th>P</th>
<th>128x2</th>
<th>64x4</th>
<th>32x8</th>
<th>16x16</th>
<th>8x32</th>
<th>4x64</th>
<th>2x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.100896</td>
<td>0.116676</td>
<td>0.130472</td>
<td>0.155914</td>
<td>0.134832</td>
<td>0.156912</td>
<td>0.168237</td>
</tr>
<tr>
<td>256</td>
<td>0.007319</td>
<td>0.011194</td>
<td>0.010198</td>
<td>0.012727</td>
<td>0.032863</td>
<td>0.020711</td>
<td>0.016143</td>
</tr>
<tr>
<td>128</td>
<td>0.000611</td>
<td>0.001161</td>
<td>0.001694</td>
<td>0.001314</td>
<td>0.003822</td>
<td>0.002749</td>
<td>0.003450</td>
</tr>
<tr>
<td>64</td>
<td>0.000114</td>
<td>0.000178</td>
<td>0.000303</td>
<td>0.000295</td>
<td>0.000317</td>
<td>0.000331</td>
<td>0.000335</td>
</tr>
</tbody>
</table>

Table 9: Average times (in s) for All-to-All within the columns communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 19)

<table>
<thead>
<tr>
<th>P</th>
<th>128x2</th>
<th>64x4</th>
<th>32x8</th>
<th>16x16</th>
<th>8x32</th>
<th>4x64</th>
<th>2x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.043106</td>
<td>0.069065</td>
<td>0.093875</td>
<td>0.094798</td>
<td>0.096405</td>
<td>0.086782</td>
<td>0.088237</td>
</tr>
<tr>
<td>256</td>
<td>0.005359</td>
<td>0.008204</td>
<td>0.008662</td>
<td>0.009368</td>
<td>0.010015</td>
<td>0.010461</td>
<td>0.010993</td>
</tr>
<tr>
<td>128</td>
<td>0.000574</td>
<td>0.000948</td>
<td>0.001178</td>
<td>0.001153</td>
<td>0.001323</td>
<td>0.002363</td>
<td>0.002490</td>
</tr>
<tr>
<td>64</td>
<td>0.000095</td>
<td>0.000169</td>
<td>0.000279</td>
<td>0.000295</td>
<td>0.000317</td>
<td>0.000317</td>
<td>0.000317</td>
</tr>
</tbody>
</table>

Table 10: Minimum times (in s) for All-to-All within the columns communicators on 256 processors using various problem sizes and virtual processor grid layouts (data to Figure 19)
<table>
<thead>
<tr>
<th>Problem size</th>
<th>ESSL non-strided</th>
<th>FFTW non-strided</th>
<th>ESSL strided</th>
<th>FFTW strided</th>
<th>ESSL non-strided + copy</th>
<th>FFTW non-strided + copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.000002</td>
<td>0.000003</td>
<td>0.000005</td>
<td>0.000003</td>
<td>0.000003</td>
<td>0.000002</td>
</tr>
<tr>
<td>24</td>
<td>0.000006</td>
<td>0.000015</td>
<td>0.000013</td>
<td>0.000015</td>
<td>0.000009</td>
<td>0.000002</td>
</tr>
<tr>
<td>32</td>
<td>0.000015</td>
<td>0.000022</td>
<td>0.000039</td>
<td>0.000025</td>
<td>0.000022</td>
<td>0.000004</td>
</tr>
<tr>
<td>48</td>
<td>0.000079</td>
<td>0.000117</td>
<td>0.000125</td>
<td>0.000146</td>
<td>0.000118</td>
<td>0.000018</td>
</tr>
<tr>
<td>64</td>
<td>0.000198</td>
<td>0.000187</td>
<td>0.000468</td>
<td>0.000407</td>
<td>0.000377</td>
<td>0.000029</td>
</tr>
<tr>
<td>96</td>
<td>0.000787</td>
<td>0.001004</td>
<td>0.001510</td>
<td>0.001757</td>
<td>0.001312</td>
<td>0.000156</td>
</tr>
<tr>
<td>128</td>
<td>0.002081</td>
<td>0.002412</td>
<td>0.012866</td>
<td>0.012702</td>
<td>0.012778</td>
<td>0.000365</td>
</tr>
<tr>
<td>160</td>
<td>0.004166</td>
<td>0.005987</td>
<td>0.017227</td>
<td>0.017635</td>
<td>0.011301</td>
<td>0.001550</td>
</tr>
<tr>
<td>192</td>
<td>0.011183</td>
<td>0.011415</td>
<td>0.051636</td>
<td>0.050686</td>
<td>0.040380</td>
<td>0.013036</td>
</tr>
<tr>
<td>256</td>
<td>0.054077</td>
<td>0.040270</td>
<td>0.208727</td>
<td>0.207046</td>
<td>0.241949</td>
<td>0.013122</td>
</tr>
<tr>
<td>384</td>
<td>0.133264</td>
<td>0.124624</td>
<td>0.853670</td>
<td>0.849163</td>
<td>0.900775</td>
<td>0.040261</td>
</tr>
<tr>
<td>512</td>
<td>0.369120</td>
<td>0.388514</td>
<td>2.670609</td>
<td>2.727210</td>
<td>2.963480</td>
<td>0.226971</td>
</tr>
</tbody>
</table>

Table 11: Detail average times (in s) for 1D FFT comparison between ESSL and FFTW (Figure 34 and Figure 35)

<table>
<thead>
<tr>
<th>P</th>
<th>512³</th>
<th>256³</th>
<th>128³</th>
<th>64³</th>
<th>32³</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.422040</td>
<td>7.924243</td>
<td>0.599342</td>
<td>0.048054</td>
<td>0.002614</td>
</tr>
<tr>
<td>2</td>
<td>44.012531</td>
<td>4.829876</td>
<td>0.222333</td>
<td>0.020038</td>
<td>0.001251</td>
</tr>
<tr>
<td>4</td>
<td>19.109078</td>
<td>2.236013</td>
<td>0.103806</td>
<td>0.007938</td>
<td>0.000663</td>
</tr>
<tr>
<td>8</td>
<td>9.333622</td>
<td>0.952405</td>
<td>0.050862</td>
<td>0.003496</td>
<td>0.000382</td>
</tr>
<tr>
<td>16</td>
<td>5.356105</td>
<td>0.526056</td>
<td>0.027752</td>
<td>0.002084</td>
<td>0.000240</td>
</tr>
<tr>
<td>32</td>
<td>2.884616</td>
<td>0.204429</td>
<td>0.017169</td>
<td>0.001385</td>
<td>0.000196</td>
</tr>
<tr>
<td>64</td>
<td>1.477321</td>
<td>0.123126</td>
<td>0.007362</td>
<td>0.001061</td>
<td>0.000171</td>
</tr>
<tr>
<td>128</td>
<td>0.710503</td>
<td>0.059033</td>
<td>0.004111</td>
<td>0.000571</td>
<td>0.000153</td>
</tr>
<tr>
<td>256</td>
<td>0.344872</td>
<td>0.030940</td>
<td>0.002131</td>
<td>0.000413</td>
<td>0.000140</td>
</tr>
<tr>
<td>512</td>
<td>0.235160</td>
<td>0.015548</td>
<td>0.001584</td>
<td>0.000366</td>
<td>0.000201</td>
</tr>
<tr>
<td>1024</td>
<td>0.190792</td>
<td>0.008541</td>
<td>0.001506</td>
<td>0.000555</td>
<td>0.000314</td>
</tr>
</tbody>
</table>

Table 12: Detail average times (in s) for scaling test in Figure 36 on 1 to 1024 processors using different problem sizes from 32³ to 512³
<table>
<thead>
<tr>
<th>Send size</th>
<th>Manual buffer copy + send (serial read)</th>
<th>Struct/Vector derived data type</th>
<th>Manual buffer copy + send (serial write)</th>
</tr>
</thead>
<tbody>
<tr>
<td>432</td>
<td>0.000011</td>
<td>0.000014</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>0.000015</td>
<td>0.000017</td>
<td>0.000016</td>
</tr>
<tr>
<td>3456</td>
<td>0.000043</td>
<td>0.000042</td>
<td>0.000046</td>
</tr>
<tr>
<td>8192</td>
<td>0.000067</td>
<td>0.000067</td>
<td>0.000074</td>
</tr>
<tr>
<td>27648</td>
<td>0.000175</td>
<td>0.000145</td>
<td>0.000197</td>
</tr>
<tr>
<td>65536</td>
<td>0.000837</td>
<td>0.000353</td>
<td>0.000887</td>
</tr>
<tr>
<td>221184</td>
<td>0.001515</td>
<td>0.001256</td>
<td>0.001796</td>
</tr>
<tr>
<td>524288</td>
<td>0.003266</td>
<td>0.002883</td>
<td>0.004733</td>
</tr>
<tr>
<td>1024000</td>
<td>0.006165</td>
<td>0.005970</td>
<td>0.007720</td>
</tr>
<tr>
<td>1769472</td>
<td>0.015638</td>
<td>0.011516</td>
<td>0.020083</td>
</tr>
<tr>
<td>4194304</td>
<td>0.054435</td>
<td>0.039225</td>
<td>0.060618</td>
</tr>
<tr>
<td>14155776</td>
<td>0.158735</td>
<td>0.126129</td>
<td>0.200560</td>
</tr>
</tbody>
</table>

Table 13: Detail times (in s) for the comparison between MPI_Alltoall using manual buffer packing (2 versions) and MPI_Alltoall using a derived data type (Figure 33)
References

[1] M. Eleftheriou, J. E. Moreira, B. G. Fitch, R. S. Germain, 
*A Volumetric FFT for BlueGene/L*, Lecture Notes in Computer Science, 
volume 2913, 2003, pp. 194-203

EPCC, The University of Edinburgh  
(http://www.epcc.ed.ac.uk/msc/dissertations/dissertations-0506/hjagode.pdf)

[3] A. Grama, A. Gupta, G. Karypis, V. Kumar,  


[5] HPCx website  
http://www.hpcx.ac.uk

[6] LINPACK Benchmarks website  
http://www.netlib.org/benchmark/hpl

*IBM eServer pSeries High Performance Switch*  
Performance White Paper  

[8] O. Lascu, Z. Borgosz, J. D. S. Davis, P. Pereira, A. Socoliuc,  
*An Introduction to the New IBM eServer pSeries High Performance Switch*  
IBM Redbooks  

The HPCx building block  
http://www.hpcx.ac.uk/about/events/annual2005/talks/Follows_IBM.pdf

[10] The Presta Stress Benchmark Code v1.2  
http://www.llnl.gov/asc/computing_resources/purple/rfp/benchmarks/limited/presta

[12] Intel® MPI Benchmarks 3.0

[13] Intel® MPI Benchmarks 3.0
Users Guide and Methodology Description

[14] LoadLeveler for AIX 5L and Linux V3.2
Using and Administering

[15] Fastest Fourier Transform in the West (FFTW)
http://www.fftw.org/fftw2_doc/

[16] IBM Engineering Scientific Subroutine Library (ESSL)
http://www-03.ibm.com/systems/p/software/essl.html