Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Bachelor Thesis

Benchmarking of Non-Blocking Input/Output on Compute Clusters

Name: David Büttner
Betreuer: Prof. Dr. Thomas Ludwig
Abgabe Datum: April 24, 2007

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

.....

Date of Submission: April 24, 2007

Acknowledgements

My special thanks to Prof. Dr. Thomas Ludwig for his guidance and support during the last months and for the opportunity to write this thesis. Also I would like to thank very much Julian Martin Kunkel for his support with PVFS2 and the trace tools used as well as for his unending patience with all my questions. Furthermore my thanks go to our cluster administrators for their help in setting up the test environments.

Last but not least I want to thank my flat mates for their patience with me and for all the meals prepared when my time was dedicated to writing this paper.

1 Abstract

As high performance computing becomes more important and supercomputers grow bigger and faster, the I/O part of applications can become a real problem in regard to overall execution times. While use of new and specialized hardware and tuning of parallel file systems help a lot in the struggle to minimize I/O times, adjustment of the execution environment is not the only option to improve overall application behavior.
It is not always possible to run applications on parallel computers which are especially adabted to that kind of applications and one has to use the system settings provided. Not only the system administrator and the developers of hardware or software components can help to reduce application execution times but also the application programmer can help by making use of non-blocking I/O operations.
To see if and how much performance gains can be achieved by making use of non-blocking I/O operations this thesis discusses non-blocking I/O operations in detail, proposes a benchmark to measure performance gains when switching from blocking I/O operations to their non-blocking counter parts and presents the results of different series of test runs.

Contents

2 General goals of the thesis

Execution times of applications run on computer clusters in the area of high performance computing can often increase a lot when a lot of I/O operations have to be executed. In order to minimize I/O as a bottleneck in the computer system a lot of optimizations in different software layers of the execution environment are applied and specialized hardware also helps to reduce I/O times. While these improvements are very important they are not the only way to reduce the time an application needs on a given computer system to terminate faster.
The idea of non-blocking I/O operations also suggests the possibility to improve application behavior and to reduce the impact I/O has on the overall program execution.

This thesis first talks about high performance computing in chapter 3 and will discuss the idea of benchmarking in chapter 4. Furthermore some existing benchmarks for parallel computing, especially for parallel I/O systems, are presented.
Then the idea of non-blocking I/O operations is presented and the theory behind possible performance gains by switching from blocking I/O operations to non-blocking I/O operations is discussed in detail in chapter 5.
In order to see if the potential performance gains suggested in theory can actually be achieved, a benchmark is proposed and described in chapter 6 and the results of a series of test runs made is presented in chapter 7.
Finally chapter 8 compares the results with the expectations established when discussing the theory behing non-blocking I/O operations in order see what can actually be achieved.

In addition to this a few extra test runs will be presented in chapter 9, for which the assumption is made that the computer cluster does not have a dedicated I/O part and that the application making use of the I/O system is run on the same nodes as the I/O servers of the parallel file system used.

3 High performance computing

performance [...] the ability to operate efficiently, react quickly [6]

computing [...] the operation of computers [6]

3.1 The TOP500 - a history of HPC

While a regular personal computer or notebook is enough for most users, there are groups for whom the computing power of these is not enough. Quite a lot of fields need more computing power for their needs as one regular PC can provide, even when used efficiently. This might be for a lot of reasons including applications having to do so many calculation that termination of the application would be way too long to reasonably be waiting for. Other applications need more than the average main memory provided by regular PCs. Therefore `supercomputers` have been designed and built during the last years to grow in nearly every aspect to satisfy those needs.

Used in a lot of areas of research, especially Weather and Climate Research, but also outside of research in production systems in the areas of Finance, Geophysics, Energy, and others [5], supercomputers provide ever greater computing power measured in `floating point operations per second` also known as `Flops`. While the developements over the recent years provided more and more resources, the race towards the bigger and faster supercomputer seems to go on without an end in sight.

When the first TOP500 list was published in June 1993, the number one of the list was located in the Los Alamos National Laboratory, USA, and had 1024 processors [5, TOP500-List June 1993]. The peak performance of this Supercomputer was 131 GFlops. Measured with the Linpack benchmark the 500 fastest supercomputers have been ranked every 6 month since then and the trend is clear: bigger and faster.
This is not only shown by looking at how the peak performance of the number one of the list grew from 131 GFlops in 1993 to 367000 GFlops in November 2006 [5, TOP500-List November 2006], but also by the fact that the current number 500 with its 4896 GFlops outperforms the number one of 1993 by quite a margin.

This growth goes hand in hand with the improvement and extension of all different parts and resources of the supercomputers. As an example Figure 3.1 shows the distribution of number of processors per supercomputer for the years 1993, 1999 and 2006.[5] One can easily see the trend to more CPUs per supercomputer.

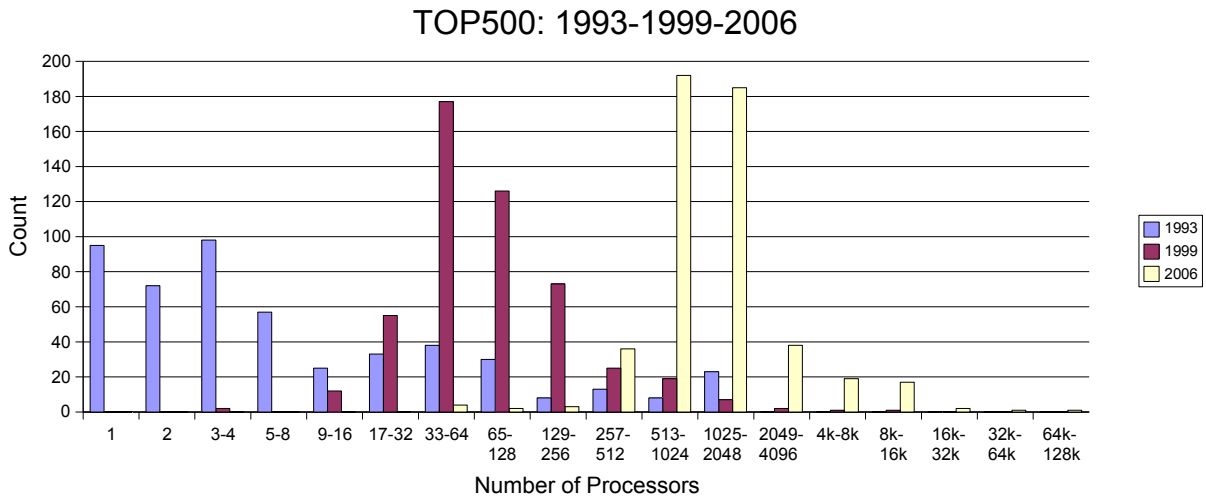

Figure 3.1: Shift of number of CPUs of TOP500 supercomputers

3.2 HPC on computer clusters

"**cluster** `[...]  a number of things of the same kind growing closely together [...]`" [6]

3.2.1 Computer clusters

As the supercomputers grew bigger and faster they not only provided more computing power and main memory, but also grew more and more expensive. Often hand built with specially developed parts and in very low quantities millions of dollars are necessary to buy one. Furthermore most of the different supercomputer architectures have their own instruction set and run a different operating system. Programmers need to adjust to individual systems and in most cases could not benifit from experience. [16, Foreword]

In the early 1990's a different kind of supercomputer was born: the computer cluster. A lot of homogeneous compute resources, mostly regular PC architectures, connected by some kind of network in order to be able to communicate between the so called `nodes`. A supercomputer for everyone.
As the computer marked grew and prices for desktop PCs fell it suddenly became possible to build a supercomputer using parts one was accustomed to and which were available at very low cost. In addition to this the cluster could run commodity software and provided the option to develop tools which now are very portable. A lot of groups like the Beowulf Cluster Project worked hard to create those in order to provide everybody with the needs to operate their own supercomputer. [16, Chapter 1]
Users are now able to use the aggregated compute power of their cluster nodes for performance or have additional fault tolerance in their system. But in order to be able to do so, programs as yet run as sequential programs need to be adjusted to run in the parallel compute environment provided by the clusters.
The idea is to split up the calculation into different parts which can be executed in parallel and

distribute them across the compute nodes of the cluster. While in some cases this might be enough, most applications regularly need to have global intermediate results of the calculation in order to go on. Therefore it is necessary for the different parts of the program, executed as different processes on the different nodes, to communicate with each other. As said before network connections between the nodes are provided in order to enable this.

3.2.2 MPI, MPI-2 and MPICH2

In order to run parallel programs on clusters and be able to have the processes communicate, one needs to compile the program, distribute it to all the nodes, start the processes there and have some way to organize communication. While in the beginning different groups had similiar but different ideas about how to accomplish this in the best way, it soon became clear that some kind of standard was needed.
What was common in all ideas was that the communication of the processes had to be based on some kind of message passing since no shared memory is available in clusters, at least not between the individual compute nodes. In some cases clusters are built using nodes with more than one CPU, so called `symmetric multiprocessors (SMP)`.

Since November 1992 the Message Passing Interface Forum members met to discuss this topic and to publish the first `message-passing interface standard` in 1994 [1]. This and the following revised versions of MPI 1.x only included communication [1].
1997 the Message Passing Interface Forum published the `Extensions to the Message Passing Interface`: MPI-2. The MPI-2 paper does not only include extensions to the MPI-1 standard, but also addresses topics which include process creation and management, one-sided communication and external interfaces. Most important for this thesis it also includes I/O functionality for parallel programming. [2]
The standard is a message-passing library specification but not a programming language. It provides routines for these tasks of communication, I/O, etc., which are implemented in the programming language used, such as C, C++ or Fortran and are called by the client program. [16, Chapter 8]
Implementations exist in different languages and more than once in those. Different groups or hardware vendors implement their version, often to fit the underlying hardware architecture. Some of the implementations are BeoMPI, MPICH-GM, MPICH-G2, MPI-Madeleine, MPICH-V, MPI/GAMMA, MPI/Pro, MP-MPICH, MVABICH, MVICH, ScaMPI, LAM/MPI or LAM/MPI's next generation MPI implementation - Open MPI. Open MPI tries to combine the ideas of FT-MPI, LA-MPI, LAM/MPI and PACX-MPI and wants to provide the best MPI implementation available [3] [16, Chapter 8].
While a lot of implementations are out there, sometimes freely available and even as open source software, this thesis is done using the MPI implementation MPICH2. MPICH2 is the reimplementation of MPICH and is developed at the Argonne National Laboratories, Argonne, USA. Its aim is to provide a MPI implementation for important platforms, especially for clusters. The MPICH2 developers also had an implementation in mind which can be used in research for MPI implementations and can help to improve the developement of any aspect of parallel programming. It is freely available as open source software at [4].

3.3 HPC and I/O

3.3.1 Overview

In the beginning of High Performance Computing, I/O was not much of a worry. It was now possible to do fast calculations of big problems, reducing very long calculation times to relatively short ones. One would load a program, execute it and in the end write the results to disk. Programs with still bigger execution times sometimes would have to write checkpoints during execution in order to be able to restart execution there in case of system failures. But with the evolution of supercomputers and ever faster processors, I/O became a bigger bottleneck in comparison to calculation and communication. [16, Chapter 19] Users were forced to write checkpoints less frequently in order to overcome this bottleneck.
Also users now were able to use the speedup in supercomputing to create more and more useful data and therefore had the wish to save this data. I/O got and still gets more and more important.
But the amount of data to be written or read is not the only problem arising when changing to parallel programming environments. Now one not only has one process executing a sequential program which is accessing the data, but suddenly has a lot of processes executing who need to access the data concurrently. Also the data is most likely not stored in local disks but distributed somewhere in the cluster network or on a connected storage server.

To address these issues the MPI-2 standard adresses I/O in an analog way to communication. An interface for I/O operations is defined and collective and asynchronous I/O is introduced. Also the very important concepts of derived datatypes, file views and different kinds of file pointers are presented. [2] [9, MPI-IO] Files are opened collectively by all processes which are given access to the file, which enables MPI to implement the support for individual or collective file access. Every process sets the datatype expected in the file, creates its own file view and is then able to access the data in the file inside their file view space. These operations also provide the possiblilty to access non contiguous data with only one I/O operation call.
Most importantly the definition of the standard provides the possibility for the MPI implementation to optimize the internal operations to actually manipulate the data for the underlying system without the need for programmer to change the application when switching to a new system. [16, Chapter 9&19] [2]

3.3.2 Parallel file systems

While the definitions in 3.3.1 solve the problems of concurrent access of multiple processes to a single file located somewhere in network, it does not necessarily solve the problem of the great amounts of data needed to be manipulated.

What is left is a series of issues that needs to be addressed. Files might be too large to fit on singel hard disk. The time to access the data is still too big. More hardware used to address these problems leads to more failures and the need of backup systems.
Also a very important issue is the question of data consistency since one file is not only accessed by one client, but multiple clients at the same time.

These issues are addressed by `Parallel File Systems`.

In [16, Chapter 19], Walt Ligon and Rob Ross give three characteristics which describe a parallel I/O system, in which a parallel file system is used:

- multiple hardware I/O resources on which data will be stored,
- multiple connections between these resources and compute resources, and
- high-performance concurrent acces to these I/O resources by numerous compute resources.

A parallel file system is a software layer operating on a parallel I/O system. It provides a global namespace for different physical I/O devices and can be seen as an abstraction of where which part of the data is stored.
To higher level software layers it provides an interface presenting the data as a directory hierarchie and allowing concurrent access to the data by more than one process. It provides a consistent few of the data even in parallel environments. The intefaces provided by parallel file systems often are not only limited to UNIX file system semantics but also provide a MPI-IO file view compliant interface to be able to optimize data access. [16, Chapter 19]

Its role is also to organize distribution of data over physically distributed I/O resources and therefore is able to provide faster acces to the data through use of aggregated throughput and bandtwidth of I/O servers, hard disks and interconnects between I/O devices and compute resources. This is made possible by striping the data across the I/O servers in a similar fashion as done in RAID systems. [16, Chapter 19] [8]

PVFS2

The parallel file system used in this thesis is the `Parallel Virtual File System 2 (PVFS2)`. It is an open source parallel file system designed as a client-server architecture. It aims to provide the possiblilty of multiple processes operating on multiple nodes of a cluster to access large amounts of data located on some subpart of the cluster. It strives to be very scalable.
PVFS2 consists of two parts: pfvs2-server or pvfs2-client. A pvfs2-server can have one of or both of two roles: data server or metadata server. Data servers are run on the I/O nodes of the cluster and manage the data stored locally on that node. Metadata servers store and manage all object attributes. Furthermore the second part is the PVFS2 client. Client applications can use the provided userlevel-Interface to access the data.[15, 7]
While some parallel file systems do provide file consistency using for example locking mechanisms, PVFS2 was designed using relaxed semantics and defining data access semantics without data locking. This and a careful design and ordering of metadata operations provide consistency. This is done because locking mechanisms always lead to performance loss with a rising number of accessing processes and soon become a bottleneck in the I/O system. [13]
For more information on PVFS2 and an in depth look at this parallel file system see [15, 7] or refer to the PVFS homepage at `www.pvfs.org`.

4 Benchmarking

4.1 Introduction to benchmarking

When writing programs, whether sequential programs or programs to be executed in parallel environments, the progammer soon has to make quite a few decisions. How to organize the program, which part in which order and, in case of parallel programs, which process takes on which part of the problem. Furthermore at some point decisions are made on which hardware to deploy, which software tools and operating system to use. In short: one has to decide on the execution environment.
Furthermore given problems and possible solutions implicate different behavior of the program and this causes different performance behavior in different environments.

Since not everybody has the resources to buy and configure new computers, supercomputers and clusters for each new type of application, most institutions have to decide which hardware and software to use. Since this is done more or less once for a long time, one has to find a way of deciding which system would be best.
One way would be to take the applications which will be run on the system and make test runs before buying a system. But in most cases it is not one or only a few applications which will be run and, as is the case especially with supercomputers, more than one person or even group will be using the system. Therefore a system has to be found to meet the needs of all those users, which turns the idea of trying out a system at the vendor with all the expected applications into an impossible task. Another method is needed: a benchmark.

> **benchmark** [...] an example of sth which is used as a standard or point of reference for making comparisons [6]

One solution to this problem is writing a program which can be used in order to make comparisons between the different systems available. A benchmark should represent the needs of expected applications, their workload and execution behavior. The development of a benchmark should be done having in mind which values are important and what exactly is to be measured. Nearly always a benchmark is created and run in order so see which scenario provides the highest performance and minimizes execution times and to find out how given applications will perform on different systems. Looking at existing benchmarks one can see a lot of different ideas about how to write them, about what they should do and how they should do it. On the other hand a lot of similarities are apparent throughout a lot of groups of benchmarks.
General differences for example include ideas about benchmark execution times. Opinions go from the believe that benchmarks should finish in a relatively short period of time [12] to believes that longer execution times can provide more meaningful results.

What basically all benchmarks have in common is the fact that any computer system has a

lot of hardware components and different software layers which will be used when running applications. Each application uses all these parts and layers in a different manner and this usage needs to be represented in a benchmark. To create this `"example of sth which is used as a [...] point of reference for making comparisons"` [6] an examination of the expected applications and workloads needs to be undertaken and a benchmark has to be created which represents the usage and patterns of these components.
This includes amongst others the use the applications make of the CPU, network interface cards, random access memory and different available cache memories. Important also are factors which are not as obvious such as use of available switches in the network for example. If the applications the benchmark represents have communication patterns which try to send more data at one time as the switch can handle, the benchmark needs to represent this, too.
When looking at the I/O needs of applications, access patterns are one of the important factors which need to be represented but also amount of data being accessed should not be neglected. When different computer systems are compared to on another the existence of cache memory on the side of the storage devices can influence application behavior and can result in different data access times when the amount of data being accessed is changed.

All in all one can say that when running applications on computer clusters a long list of factors influences the execution behavior. Different configurations and setups can have more or less great impact on performance issues and what is good for one kind of application might be bad for another.

Problems with benchmarking

As said before benchmarks are written to find the hardware which is best for a special kind of application and in addition the software layers and their configurations which provide the best performance behavior.
While in a lot of cases benchmarks are able to aid in the search of bottlenecks and provide knowledge about which part of the system to adjust to remove those, there are also some limitations of benchmarks.

When a benchmark is used to compare computer clusters together with their given configuration it is not always possible to say that the results of the benchmark tell the user which setup would be able to provide better performance for this kind of application. It only shows which of the systems is better when a given setup and configuration is provided and cannot be adjusted.
The question is whether a benchmark is run on a computer system without tuning the system for the test and without optimizing the benchmark code to fit the resources provided or whether it is run after adjusting setup parameters to support the benchmark. While an application can perform better on one computer system than on another it might be possible that with a little bit of tuning the second system really is the one which could provide better execution times for the given application.

Furthermore nearly all existing benchmarks test how a given application performs when executed on different computer systems or how performance is influenced when configurations on one computer system are changed. All changes in execution times measured are results of changes of some part of the computer system outside the represented applications.
While this is very important, changes outside the client applications are not the only possibility to achieve better execution times. Also different ways of implementing a program and the use

of different concepts of communication and I/O can improve application behavior.
Nearly none of the available benchmarks provide possibilities to compare different ways of writing programs when trying to minimize execution times. This thesis will discuss the use of non-blocking I/O operations as one way to reduce execution times. It will propose a benchmark which aims at measuring the performance gains one can achieve when changing from blocking I/O operations to non-blocking I/O operations and will present the results of a series of benchmark test runs. Finally the results will be compared to expectations and will show if computer system setup and configuration is the only way to improve application performance or if the programmer can also aid in reducing the time his program needs to terminate.

Before non-blocking I/O operations and their semantics are discussed, some available parallel benchmarks are presented in the following sections.

4.2 Overview over existing benchmarks

4.2.1 Calculation and Communication Benchmarks

Effective Communication and File-I/O Bandwidth Benchmarks

The effective communication banwidth benchmark `b_eff` is one of two benchmarks designed to measure communication and I/O performance on parallel computers. It is implemented using MPI functions.
Both benchmarks aim to provide one characteristic value by running several experiments and taking an average of the results.
The communication benchmark runs a large number of very short test, all of which are run using several communication methods. This allows to obtain results which do not depend on the question which implementation of the MPI functions is optimized.
Also the benchmark makes sure, that all processes send messages to other processes at the same time, resulting in parallel communication. The test runs are combinations of a series of different communication patterns, 21 message sizes and 3 communication methods. The communication patterns include 6 ring patterns, 30 random and 13 additional patterns. All tests are run 3 times, resulting in 9261 experiments, which in the end calculate the `effective bandwidth`.
Since the developers of this benchmark suite believe that the execution time of a benchmark should be limited, the length of each experiment loop is automatically controlled.

[12, 11]

4.2.2 I/O Benchmarks

Effective Communication and File-I/O Bandwidth Benchmarks

The I/O benchmark of the effective communication and file-i/o bandwidth benchmarks, `b_eff_io` is a benchmark implemented with MPI functions. Its aim is to characterize the effective I/O-bandwidth. This benchmark first analyses the applications needs and then tests

hardware and parallel file system on its performance in satisfying these and not vice versa. It tests a variety of different access patterns, running 315 different measurements. These are combinations of 5 different I/O patterns, 7 different chunk sizes, 3 numbers of parallel benchmark processes and the access forms initial write, rewrite and read.
The developers of b_eff_io had a fixed time in mind concerning how long it should take to complete the benchmark. They believe that 10 minutes should be enough to overrun any cache in the system.
One unique scenario of this benchmark is the comparison of performance behavior of wellformed and non-wellformed data sizes. Wellformed data sizes being powers of 2.

[12, 11]

PIO_BENCH

In his master thesis F. Shorter proposes a suite of access pattern benchmarks which are representative of the needs of many parallel I/O codes. He aims to reduce the degree of subjectiveness running benchmarks by defining one unique timing mechanism. This timing mechanism is proposed as part of a framework for all tests in order to allow easy adding of new access pattern tests. To provide a standard way of setups for the access pattern tests, each module needs to implement 7 predefined functions. The field of responsibility for each of these function is predefined in order to have comparable test results.
In theory the paper also proposes a standard framework for interpreting results which should automatically evaluate the data gathered by the tests.
All in all the PIO_BENCH benchmark has been proposed in order `"to reduce the degree of subjectiveness in choosing a parallel filesystem to meet the needs that parallel applications typically have"` [14].
A very interesting part of the thesis is the discussion and description of a lot of different access patterns, both spacial and temporal access patterns.

[14]

PRIOmark

PRIOmark is a benchmark which `"measures file system and disk I/O performance of modern computer systems"` [10]. Unlike the benchmarks described above, PRIOmark not only uses the MPI-IO interface but also takes advantage of POSIX I/O semantics. POSIX I/O semantics are even used in parallel tests run in distributed environments.
PRIOmark comes with a set of sub-benchmarks which include a `raw` benchmark measuring performance of raw disk access. The sub-benchmarks `common_file` and `strided` measure access performance of a lot of processes to a single file, each accessing one large consecutive block of data for `common_file` and several non contiguous blocks in the `strided` case respectively. The penultimate sub-benchmark is the `single_file` test in which each process accesses an individual file on a local disk. In distributed environments every process would have its own file on a local disk which would be accessed in a block-by-block manner.
Last but not least, unlike nearly all benchmarks available for parallel I/O, PRIOmark also has a sub-benchmark which is called `async`. As described later in this paper (see 5.2), MPI-IO introduces the idea of asynchronous I/O, which allows for calculation to continue while data is being accessed. While this paper wants to examine how much time can be saved using

non blocking I/O routines, PRIOmark focuses on the percentage of time lost in each one of the following cases: It calculates what the authors call `asynchronous calculation loss` and `asynchronous bandwidth loss`, looking at how execution times increase for I/O and calculation. PRIOmark does not analyse whether non blocking I/O could nevertheless aid in reducing overall application execution times.

[10]

5 Non blocking I/O operations and their semantics

[to] block [...] to prevent sb/sth from moving or making progress [6]

5.1 Computer resources and concurrency

While computers grew faster and bigger throughout the years, the use of more resources and the developement of faster hardware components were not the only factors increasing the performance of computer systems.

One thing that changed during the years was the operating system. First the programmers interacted directly with the computer hardware without any kind of operating system. Then computers where created where a first version of an operating system called the monitor started to organize the use of the system allowing for batch processing. While first one program was executed after another soon the first systems where created which were able to do batch processing with multiple programs in order to use idle processor time during the execution of one job by executing another. Already on those first systems the I/O system was one of the main reasons for idle time like this. A lot of time could be saved by using the free processor cycles for a second job while the first waited for results of an I/O operation.
The next step in the developement to the current operating systems were systems allowing for time sharing. This now allowed users to interact with the programs which was not possible in the previous batch systems.
With this came new tasks to be taken care of by the operating system including process management, memory management, making sure that applications of one user didn't interfere with those of another and overall resource management. Then operating systems were created which introduced the idea of threads allowing for one program to use multiple processors on a symmetric multiprocessor architecture. Now different parts of one program can make use of different resources at the same time, different processes can be executed concurrently and unused CPU cycles by one application are used by another.

[17]

While applications needed to wait on I/O operations or execution of software stacks for network communication and therefore left CPU cycles unused which could then be used by other applications, the operations they were waiting for still needed a lot of CPU cycles themselves in order to satisfy the applications needs. The processor was needed to move data from the main memory storage devices or to the network interface card in order to send it to a distant location. Therefore data access was limited by how fast the CPU could satisfy the needs of the used hardware and the CPU was not free to attend other waiting tasks.

Direct memory access modules were created and are able to free the CPU from tasks like moving data between hard disks and main memory or between main memory and network devices. While DMA modules and CPU still need to compete for use of the computers bus system, the CPU is free of the tasks concerning data movement and can continue execution of a different process. Important here can be existing cache memories which can contain data the CPU is operating on without the need to use the bus system.

[17]

Now different parts of processes are executed in parallel while special hardware like DMA modules or extra processors on network interface cards take care of copying data from one location to another. A lot of tasks are executed in parallel and thanks to special hardware and software solutions concurrency makes faster execution of applications possible.
These ideas also play an important role for reducing execution times of parallel programs by use of non-blocking I/O as presented in the following chapters.

5.2 The idea of non-blocking I/O

When running applications on a cluster, ways of doing I/O come in many different flavors. Individual processes could access data stored in individual files or could access one shared file concurrently. Data might be saved on local hard disks or distributed over storage devices somewhere in the cluster network. Depending on the application, shared or individual file pointers allow for different kinds of access to a shared file and different file system semantics allow for block-by-block access or more advanced data manipulation by use of file views and explicit offsets.
For each of these versions of doing I/O, different parts of the I/O system allow for different options of optimizations to enhance I/O performance in order to avoid I/O becoming a performance bottleneck. Most of the points mentioned are addressed in the MPI-2 paper and are represented in the MPI-IO standard [2] or are probable parts of parallel file systems.

The MPI implementation MPICH2 for example uses amongst others the following two kinds of optimizations in their I/O software layer ROMIO in order to optimize data access.[8]
Data sieving is used to optimize access to noncontiguous data. When reading data the idea here is to not only request the noncontiguous data from the file system, but to read one large contiguous block of data and to discard the unwanted bytes between the desired concontiguous blocks. Access time can be reduced if the time to read the unnecessary data is smaller than the time won by reducing the overhead of creating a lot of small I/O requests.
In case of writing data this version operates by doing a read-modify-write operation since the holes need to be written with the current valid data.
This optimization of course only works if the use of extra send or receive buffer is available and the holes between the wanted data blocks are small and relatively few.
Two phase I/O is used to optimize collective I/O operations. Here the additional information available by the fact that ROMIO knows that all processes are going to do I/O operations can be used to enhance I/O performance. The idea is that in one phase each process reads or writes a chunk of contiguous data and in the other phase the data is distributed or collected from the corresponding process depending on whether the processes are reading or writing data.
This option also needs more buffer space on the client side than used by the data so be ac-

cessed in order to make the data organization possible. Performance benefit comes from less I/O requests and the fact that every process now requests modification of contiguous blocks of data. [8]

Another way of saving time when doing I/O is using non-blocking I/O operations.
For most applications which iteratively calculate new data, different points during execution are defined where data needs to be written or read. This often is done in a blocking way, resulting in a period of time during execution in which calculation is paused until the data access is finished. This is due to the fact that blocking I/O operations return only after successfully terminating the I/O request, i.e. in the case of a write operation when the file system signals that the data has been written and is now available for future access.
As suggested in chapter 5.1 computer systems allow for a lot of different ways of using resources concurrently. This fact soon leads to the idea of having the calculation part of an application go on while the I/O operation is being executed concurrently.
The idea of how this can be done using non-blocking I/O operations and what this means for the semantic of these operations is presented in the following paragraphs.

In order to look at non-blocking I/O operations and the question if performance gains can be achieved in comparison to the use of blocking I/O operations, one first needs to look at the blocking I/O operations.
For the following discussions an application is suggested, which iteratively does: calculations, I/O, calculations, I/O and so forth, as shown in Figure 5.1.

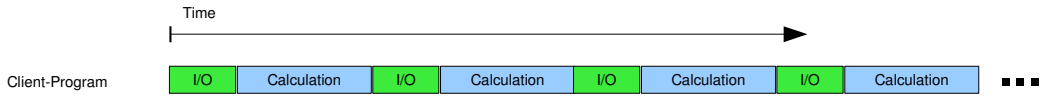

Figure 5.1: Example program using blocking I/O functions

While calculation will not be able to start again until the I/O operation has returned, it is not necessarily the case that the processor used by the executing process is used to full capacity during the time spent in the I/O operation.
In case of accessing data on local hard disks, the write or read operation could be done using direct memory access (DMA) unburdening the processor which then is not responsible for the actual movement of data between main memory and the storage device. [17, Chapter 1.7.3] The same idea can be applied when sending data over the network to a storage server somewhere in the system also leaving the CPU available for other tasks.
In case of blocking I/O operations these free CPU cycles might not be used by the client application. The application nevertheless has to wait for the operation to terminate, wasting valuable CPU cycles. Only when completion of the data transfer is signaled can the function return and therefore allow for the following calculation to commence.
Another scenario would be an application, which for whatever reason is not able to use more than one processor in a SMP environment and therefore could take advantage of a second processor to press ahead with calculations while the first processor is taking care of the I/O operations. One possible setup would be an application which needs all of the available main memory for two sets of data being manipulated by the process one after the other. While the calculation part does its work on one of the two parts, another part of the application could for example take care of saving the other part to the I/O system. While this is a possible setup, the first scenario is the more likely one.

Important is that this variant of I/O assures that after termination of the I/O function the client program can directly make sure that the operation completed successfully and could also access the data again right away.

In order to use the free CPU cycles which accrue sometime during the I/O operation, the idea of non-blocking I/O is to start the I/O operation and, in a different thread of the process, go on with calculation. If the operating system supports kernel threads and allows for one thread to go on working while another is suspended, this is possible.

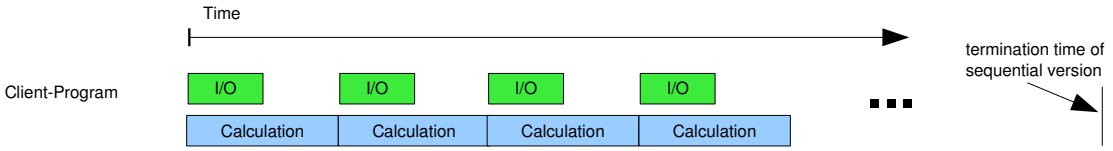

Figure 5.2: Example program using non-blocking I/O functions

Figure 5.2 demonstrates the theory behind this and indicates where the sequential version of the program in Figure 5.1 ended.
This of course assumes that individual times for I/O and calculation do not change when being executed in parallel and that there is no overhead involved when running non-blocking I/O functions.
As already indicated in [10] this is not the case and some kind of drawback for the individual times can be expected when running the non-blocking version of such a program.

Before looking at the theory in more detail in the following chapters, there are a few more points worth mentioning.
First of all one has to keep in mind that, when using non-blocking I/O, it is not ensured that data is written or read any time the application needs to do calculations. It is very important for the implementation of the non-blocking I/O functions to provide possibilities to and for the implementation of the application using the non-blocking I/O functions to use options to verify that data access has finished before trying to modify the buffer used to save that data.
Also, as mentioned before, the use of non-blocking I/O most likely needs additional main memory if the data to be accessed is also needed to continue with the calculations. In a scenario where more than one set of data already exists and calculation can be done on a set of data distinct from the data used in the I/O operation, this overhead can be saved.

5.3 Discussion of scenarios

> **scenario** [...] an imagined sequence of future events [6]

5.3.1 Introduction to different scenarios and possible performance gains

While non-blocking I/O can be done for write and read operations, this thesis only looks at write operations. The same ideas and tests can easily be applied to read operations and at least the theory isn't really different from the write operations.

In order to find out if performance boosts really are achievable write operations only are tested here. This is also done due to the fact that one possible area of application of non-blocking I/O in real applications is the time when check points are being written in order to be able to restart the application there and not from the beginning in cases of execution failures.

For the following scenarios and the rest of the proposed benchmark of this thesis the assumption is made that it is run on a computer cluster which makes use of a parallel file system allowing for all processes to access one file concurrently. Also used is the idea of a parallel file system which allows for data to be cached locally in the main memory of the I/O nodes of the cluster hosting the I/O server. This means that the regular I/O operation returns when the file system signals that all data has been received successfully, but does not mean that anything has physically been written to disc already. This is the case for the parallel file system PVFS2 used for the tests presented in chapters 6.2, 7.2 and 8. For more information on PVFS2 see chapter 3.3.2 and [7, 15]

The following chapters describe different possible scenarios of parallel programs using blocking I/O operations, their counterparts using non-blocking I/O operations and the expected performance boosts.
In contrast to the most likely expected approach of looking at the version using blocking I/O first, the scenarios are set up to consider the non-blocking variant first. This is done because it helps find the application using blocking I/O which in theory can result in the maximum speedup possible. For more information on this scenario look at chapter 5.3.4.
Furthermore the scenarios all fit into a fixed pattern, i.e. all scenarios look at a setup where a program repeatedly writes data followed by some calculation. This is done a variable number of times, but always assumes that every time I/O is done, it needs the same amount of time because everytime the same amount of data is being written. The same goes for the calculation. Small differences in each iteration are to be expected but are not taken into consideration when looking at the theory. This is because the influence of aspects as external processor use by for example the operating system, etc. are something that in theory should be negligible small. A computer cluster configured for high performance computing should only have very few tasks running on the compute and I/O nodes in order to provide the maximum amount of compute power to the applications run on the system.

All scenarios will be presented showing operations done on the client side and the expected behavior of the I/O server used to save the data. Most cases are presented as using 1 client and 1 I/O server even if that is not a scenario for parallel applications but single process programs. It nevertheless explaines the idea behind the scenario and is arranged a lot more clearly. This also helps to sum up theoretical expectations of performance behavior.

Performance boosts will be presented as a single number which compares the time for execution of the application using non-blocking I/O to the version using blocking I/O by calculating the ratio of `non-blocking version / blocking version`.
This number is to be interpreted as follows:

- A number **bigger than 1** means **performance loss**. This would show that the non-blocking version of the scenario would overall take more time than the blocking version.
- A number **equal to 1** means **no change in performance**. This would show that both version need exactly the same time to terminate.

- A number **smaller than 1** means **performance gain**. This would show that an actual performance gain has been achieved. The smaller the number, the faster the non-blocking version terminated in comparison to the version using blocking I/O routines.

While there are a lot of different aspects which could be used to create different scenarios there is one aspect which seems to be the adequate to arrange different setups.
While one could take amount of data to be written, block size of contiguous data to be written or number of processors used for the application as the primary criteria, it is not those that seem to make the most sense. When looking at execution times and the wish to reduce these, the primary criteria chosen here is the ratio between the time it takes to do the desired I/O operations and the time necessary to do the desired calculation. This choice is reinforced by the theory of the possible speedup one can get by using non-blocking I/O operations.
If for theory discussions the assumption is made that the individual times of I/O and calcultation do not change when executed in parallel, the maximum amount of time which can be saved using non-blocking I/O is the lesser of the times needed to do the desired I/O or the desired calculation.
If the I/O time is always a lot smaller than the calculation for the blocking version and the times do not change for the non-blocking version, the I/O will in each iteration be able to finish before the corresponding iteration of calculations. The result is that the time needed for overall application execution now is the sum of the individual calculation times. This can already be seen in Figure 5.2.
A more thorough discussion of this shall be presented in the following three sections discussing three scenarios. :

- The time needed to finish the desired I/O operation is always **smaller than** the time needed to finish the desired calculations.
- The time needed to finish the desired I/O operation is always **larger than** the time needed to finish the desired calculations
- Both times are always **identical** when executed in parallel.

In contrast to the examples presented already the following parts try to take into account the differences in times for I/O and calcuation in the two versions. Because of this the times refered to in the list above are the times for I/O and calculation in the version using **non-blocking I/O**.
Changes in times between non-blocking versions and versions using blocking I/O routines might be presented exaggeratedly which is done because it makes them more obvious and better visible in the graphics.

5.3.2 I/O < Calculation

As an example this scenario represents an application which is very CPU intensiv and needs to save a small amount of data periodically. With the assumption that I/O times and calculation times do not vary when executed in parallel, the time saved by using non-blocking I/O routines would be the time needed to finish all I/O operations. Figure 5.1 and 5.2 demonstrate this.
When looking at the fact that there is some overhead included when running two tasks in parallel, such as additional operations to initialize threads needed, times needed to switch between threads and new concurrent usage of resources, some changes in execution times are

to be expected.
The new concurrent usage of resources for example can include competitive use of the nodes bus system. In case direct memory access is used in order to write a block of data from main memory to hard disk or to send the data over the network interface to an external I/O server, the bus system is still needed to transfer this data. Due to the fact that calculation goes on while this is done, additional usage of the bus system arises in order to load and write back data needed for the calculation to continue. In case of communication between the processes on different nodes the use of the underlying network also has to be shared and coordinated for both communication and I/O in case there is no seperate I/O network available.

The worst case for the I/O operation wanting to save data to an I/O server somewhere in the network is one where the I/O server has no free memory available to cache the data to be written. This means that the I/O server can only receive data as fast as it takes to free cache by writing data physically to the available storage devices.
In times where no write request is submitted to the I/O server, such as in times the application only is doing calculations, the I/O server has the opportunity to free cache memory by physically writing data to disk. This is called write behind and allows for future write operations to finish faster. This happens when the available bandwidth to transfer data to the main memory of the node hosting the I/O server is faster than bandwidth available to write data to disk.
Since I/O operations can be acknowledged as finished whenever all data to be written is received by the I/O server, I/O operations terminate faster if the deciding time is the transfer time only and not times needed to to write behind.

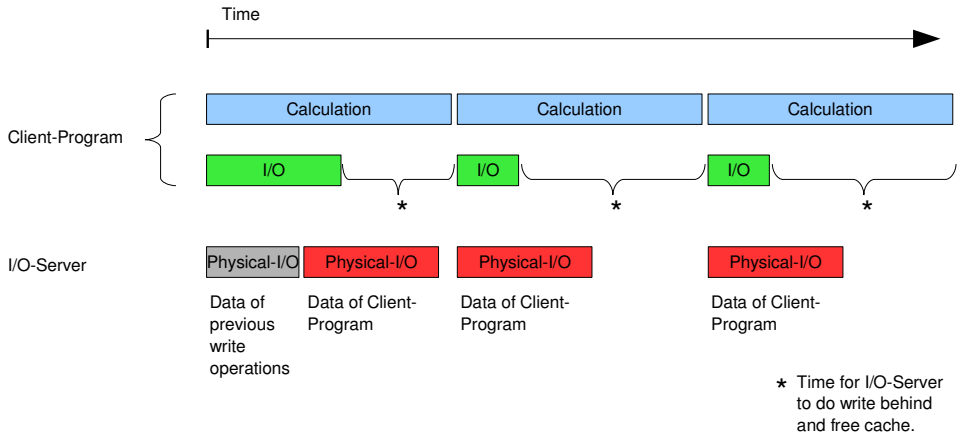

Figure 5.3: Non-blocking I/O for small I/O times compared to calculation times

Figure 5.3 shows one possible version of this scenario using non-blocking I/O routines.
Important here is that the bars representing the operations do not necessarily implicate CPU usage or usage of any kind of resource. Of course resources will be used during this time, but temporary discontinuances in execution are not being represented. The bars represent the operation from the time they are initialized to the time the I/O operation and the iteration of calculations respectively can be seen as finished. In the case of blocking I/O the end would signal the point in execution where the I/O operation can return and in case of non-blocking I/O this would signal the point in execution where the client applications would get a positive acknowledgement when checking whether the I/O operation has been finished.

On the server side of the scenario times of physical I/O are shown. These are examples and show how in order to free memory for new data in the beginning, old cached data is being written to hard disk. Because here a scenario is shown where the I/O server has no free memory in the beginning, the first I/O operation takes as long as it takes to save the same amount of data to hard disk. The other I/O operations can terminate faster since during the longer calculation the I/O server had the time to further write data to storage devices and therefore free memory which can be used to receive data of future I/O operations.

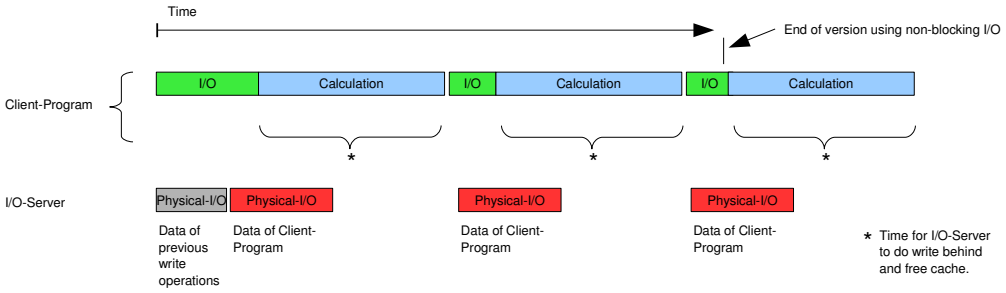

Figure 5.4: Non-blocking I/O for small I/O times compared to calculation times

Taking all this into consideration the version using blocking I/O operations would be as shown in Figure 5.4. In this scenario this is the straight forward execution of the I/O and calculation operations one after the other. Due to the fact that in both versions the I/O server has enough time to free sufficient memory to cache the data to be written in the second and third iteration, the I/O times are expected to be nearly the same in both versions. Time differences for each in between the two versions are neglected in the graphic. In theory the times for doing calculations can be expected to be shorter in this version due to the absence of CPU usage by the I/O operation.

A more interesting influence of the possibility of the I/O server to use the time during calculations to free memory is visible in the following scenarios.

When looking at the ratio between the versions using non-blocking I/O and blocking I/O routines the following can be expected.
For a theoretical discussion of the ratio let's assume what was assumed above already and say that the times for I/O and calculations respectively do not change depending on the version. Now as an example let's assume that the I/O time takes 2 time intervals compared to 8 intervals needed for calculations resulting in a total time of 10 intervals for each iteration. As seen the time needed for the version using non-blocking I/O would come down to 8 times the amount of iterations done. In the other case 10 times the amount of iterations is needed. Therefore the ratio of `non-blocking version / blocking version` equals `0.8`. This would be the best speedup possible by use of non-blocking I/O for this scenario.
Therefore for an application in which needs to do little I/O in comparison to calculations, the best possible speedup would be:

$$\frac{non-blocking\ version}{blocking\ version} = \frac{calculation\ time}{calculation\ time + I/O\ time}$$

5.3.3 IO > Calculation

This scenario represents I/O intensiv applications. A lot of data has to be written every time a few calculations have been done. As already shown above the maximum amount of time one could save would be the time to do calculations.
If taken into consideration that the I/O takes a long time and the I/O server has no chance to free cache memory in order to save data sent by a write request, it might even be possible, that the unused CPU cycles on the client side while the I/O operation is waiting is enough for the calculations to be done. This would be equal to the assumption that the times of I/O and calculation respectively do not change when non-blocking I/O routines are used in comparison to a blocking version.

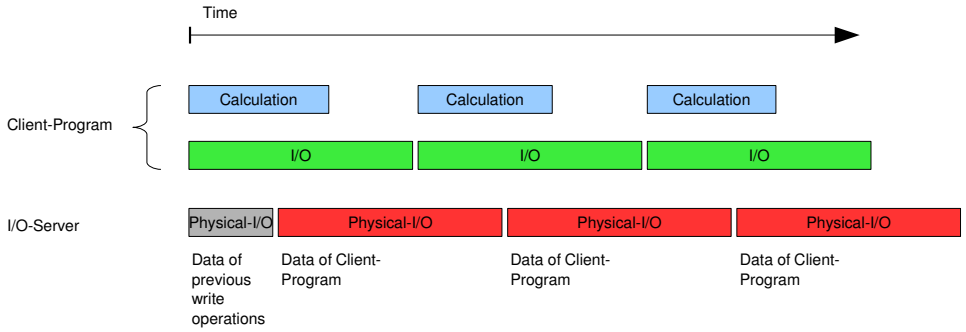

Figure 5.5: Non-blocking I/O for large I/O times compared to calculation times

Figure 5.5 shows the version of this scenario which makes use of non-blocking I/O routines. As done before a situation is taken as an example where the I/O server in the beginning has no free memory available to cache data. Therefore all I/O operations have to wait for the time needed to physically write the data to disk which they want to save.
Due to the fact that there are no times in which the I/O server has the chance to free memory by writing data to disk without having to receive data, all I/O operations take that amount of time and cannot profit by use of free main memory on the node hosting the I/O server.

Looking at the version using blocking I/O operations no difference should be seen for the first I/O operation. It still has to wait for the I/O server to free enough memory on the node it is running on in order to receive the data to be written.
Other than in the non-blocking version of this scenario the I/O server does have time to do write behind and free memory which can be used to cache data of the next write operation. This can be done during the times the calculations take place on the client side of the application. When looking at Figure 5.6 one can also see that it is likely that calculation terminates faster than it would in the version using non-blocking I/O. This can be expected since the calculation does not have to wait for free CPU cycles in the I/O process but can use all CPU cycles directly.
The second and third I/O operation of the example can terminate faster than the first one due to the freed memory on the I/O server side. As suggested in this example, the third I/O operation could even benefit more from this than the second by additional free memory accumulated during the second phase of calculations.

While the behavior described is to be expected, the extent of changes in times and the amount of memory possibly freed during a write behind period might not be as extreme as suggested

in the Figures 5.3 and 5.6. The extent of changes rather depends amongst others on factors like actual times needed to do the desired calculations, amount of data to be written and the amount of CPU cycles needed on the client side to transfer this data to the I/O server.

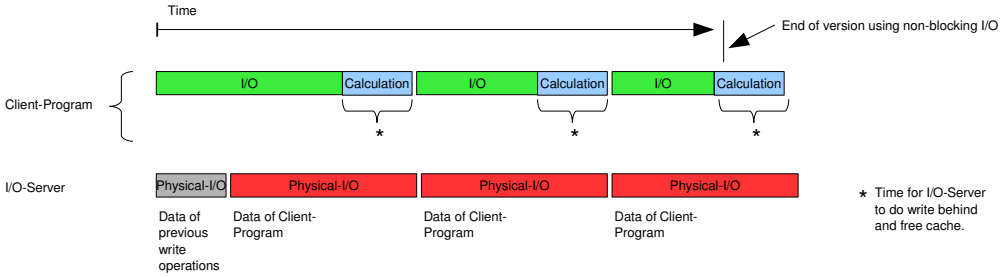

Figure 5.6: Blocking I/O for large I/O times compared to calculation times

Again when looking at the ideal situation where the times of I/O operations and calculation periods respectively do not change when changing from non-blocking I/O calls to their blocking counterparts, the ratio of execution times would be:

$$\frac{non-blocking\ version}{blocking\ version} = \frac{I/O\ time}{calculation\ time + I/O\ time}$$

So at best one can save the time needed to do the total amount of desired calcuations in this I/O intensiv scenario. While this should be possible something less than this is probable due to the described reasons. Again, these include change of execution times for both parts of the application when switching between the I/O routine variants and the question of how much data the I/O server can write to disk during calculation in order to free main memory it has available.

5.3.4 I/O == Calculation

This scenario probably is the most interesting in theory. This is due to the fact that this scenario suggests the maximum possible speedup which could promise a ration of `non-blocking version / blocking version` of `0.5`.

To get this performance gain it is best to again first take a look at an example where it makes no difference for I/O and calculation times respectively whether being executed in parallel or one after the other. The two versions using non-blocking I/O and blocking I/O routines are presented in Figure 5.7.

While this is a comparison of two scenarios which includes a lot of abstractions, it nevertheless is quite useful in the discussions of theory.
First of all the fact that a bigger workload on a computer system will not allow for faster completion shows that the shown example is the best possible. In case the times for I/O routines and calculation periods respectively change when switching from blocking I/O routines to non-blocking ones, they will change to become larger. Therefore no better speedup than

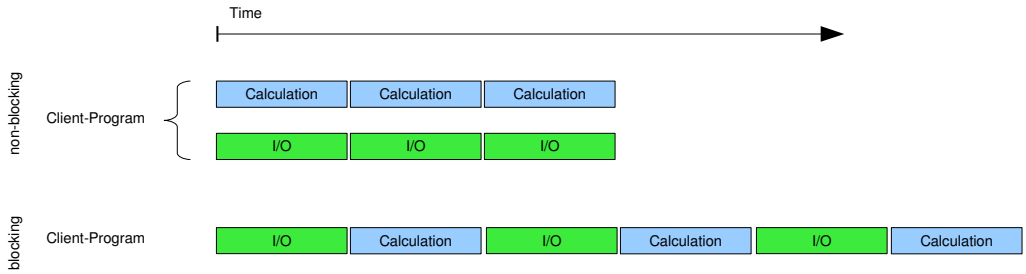

Figure 5.7: Theoratical comparison of scenarios with equal times for I/O and calculation

this situation is theoretically possible. Therefore the best value for the ratio of `non-blocking version / blocking version` is:

$$\frac{\textbf{non - blocking version}}{\textbf{blocking version}} = \textbf{0.5}$$

Now when looking at a scenario with the fact that times might differ between the two versions and also including the I/O server side, the version using non-blocking I/O would look as shown in Figure 5.8.

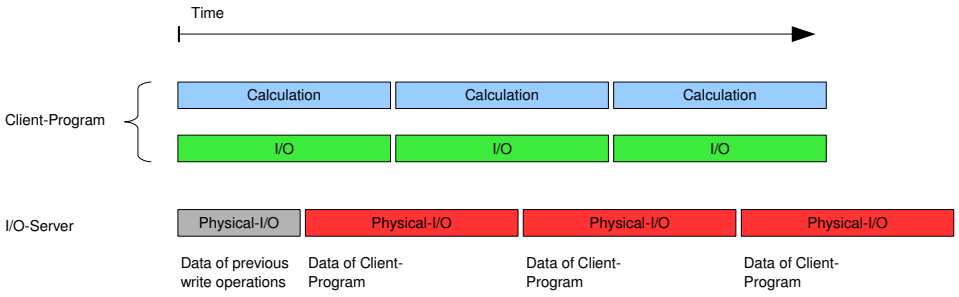

Figure 5.8: Non-blocking I/O used resulting in equal times for I/O and calculation

This scenario is important because it does not leave CPU cycle unused on the one side and does not let time pass by with calculations which could be used to transfer data to an I/O server using methods like direct memory access.
Both operations start and end at the same time which results in the maximum amount of data able to be written during a calculation period.
In case the application is run using one process per node on a cluster offering nodes with 2 CPUs each, this is especially obvious. While one CPU is busy doing calculations, the other is used to write as much data to the underlying I/O system as is possible during this time. In case the application is run on nodes with 1 CPU per node, this also is the other extreme in comparison to the scenarios presented in sections 5.3.2 and 5.3.3. In case the I/O operations would always turn out to take less time than the calculation periods, their time would be the maximum time possibly saved. In case I/O operations would always take longer than the desired calculations, the calculation times would limit the maximum possible time saved. The closer the time which can be saved is to the time of the other operation, the more time can be

saved. Therefore the biggest performance boost can be expected when both times result to be equal in the non-blocking version.

In theory this scenario would look as presented in Figure 5.7, showing the perfect behavior. In real execution environments the behavior should be different when switching from using non-blocking I/O routinges in Figure 5.8 to using blocking I/O routines.
For demonstration purposes two possible versions of the application using blocking I/O operations are presented here.

In the first example shown in Figure 5.9 the first I/O operation takes as long as it did in the version using non-blocking I/O due to the fact that the I/O server can only receive data as fast as physical I/O can be done.
Here the first calculation period is assumed to be long enough for the I/O server to write as much data to the storage devices as is needed to have memory available for all of the data of the next I/O request. Therefore the second and third write operations only take the time needed to transfer the data over the network to the I/O server memory used for caching it. After the successful transfer has been signaled the I/O server can use the times during calculations in order to further write data to disk.

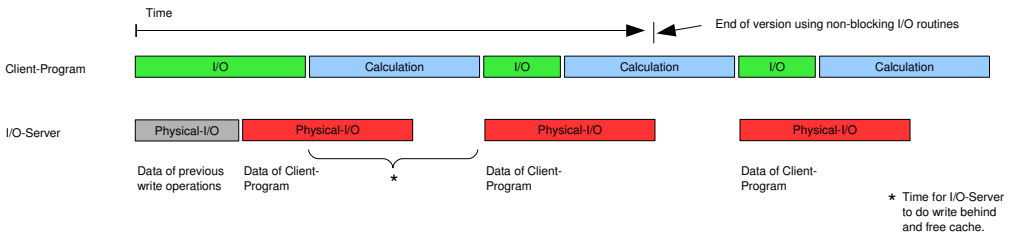

Figure 5.9: Example 1: Blocking I/O operations used for scenario in Figure 5.8

Another possible scenario would be one where the I/O server does not have enough time to save all data sent by the second I/O request as shown in Figure 5.10. This means that the second I/O request can send as much data as the I/O server has freed doing write behind for which the transfer bandwidth is the limiting factor. For the rest of the data which cannot be cached on the I/O server side the client has to wait again for physical I/O to take place. This results in a second I/O operation which is faster than the first one, but is not as fast as the case has been in the example of Figure 5.9.
Only the third write operation would take only the time needed to transfer the data in this example.

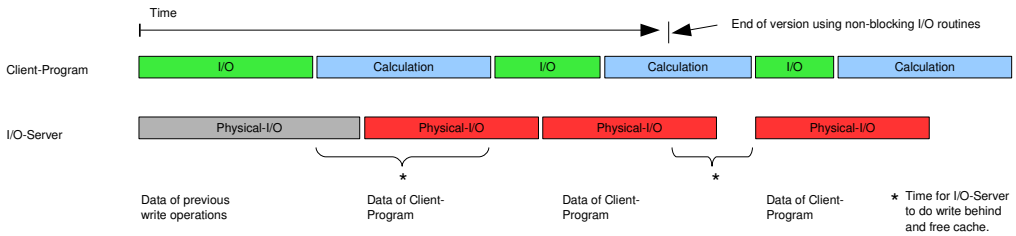

Figure 5.10: Example 2: Blocking I/O operations used for scenario in Figure 5.8

As seen looking at these examples it is not easy to find the version using blocking I/O which would save the most time when being changed to use non-blocking I/O. While the best maximum speedup possible is 0.5 for the said ratio, it is not easy to find an application which really would show this performance boost. A lot of factors play an important role when comparing the two versions and indicate that an application behaving as shown in Figure 5.7 is no likely to be found. Therefore the question is how much speedup can acutally be achieved by the possibilities suggested here.

5.4 MPICH2 and the wish to use possibilities suggested in theory

When writing applications which are to be executed on a parallel computer such as a computer cluster, a lot of programmers use the MPICH2 implementation of the MPI-1 and MPI-2 standards in order to handle the tasks of process creation, communication and file access.
A lot of times the program deveolpers do not have unlimited access to compute resources and need to make sure not to exceed their resource access time limits when doing so. Also it is desirable to use the available time in an optimized way in order to get the maximum amount of results possible. When the application needs to do I/O operations during execution the theory presented in the previous chapters probably start to sound very interesting when implementing the application.

The MPI-IO part of the MPI-2 standard does define non-blocking I/O routines [2]. In the chapter on I/O in the paper, the function `MPI_FILE_IWRITE(...)` is defined as "a nonblocking version of the `MPI_FILE_WRITE` interface" [2]. The standard also provides the routines `MPI_TEST` and `MPI_WAIT` in order to complete the non-blocking operations and at the same time to have an option to check whether it has completed successfully.
The problems start when trying to use MPICH2 to implement the application. While the non-blocking I/O functions are provided their use will not get the desired and expected results. The problem becomes obvious when looking at the source code of the MPICH2 implementation. Here the code in the MPI_File_iwrite(...) function calls the blocking version MPI_File_write(...). The semantics for the non-blocking function are exactly as defined in the MPI-IO definitions and the implementation is not controversial to the MPI-2 standard, but would not lead to the expected behavior. Here the non-blocking I/O routine would basically result in the same behavior as the blocking counterpart with the small difference that the successful completion has to be checked by a call to `MPI_Test(...)` or `MPI_Wait(...)`.

5.5 Summary of expectations and plans for the benchmark

As demonstrated in this chapter the use of non-blocking I/O routines has a lot of potential.
For applications having to do I/O operations periodically the I/O part of the application can soon become a bottleneck in performance. In order to address this problem one possiblilty is to have the calculation part of the application go on while the desired I/O operation is being taken care of.
In order to do this the MPI-2 standard defines non-blocking I/O operations [2]. The discussion

of different scenarios in which non-blocking I/O routines can be used led to the following results:

- In case the I/O operations are relatively short in comparison to the necessary calculations in between them, the time which can be saved using non-blocking I/O routines is the time needed to do the I/O when using the blocking I/O counterparts.
- In case the I/O operations are relatively large in comparison to the necessary calculations in between them, the time which can be saved using non-blocking I/O routines is the time needed to do the desired calculations in the version using blocking I/O calls.
- The biggest performance boost can in theory be achieved when both times take the same amount of time in both versions. Here execution time could be halved.

All in all this leads to the following ratio between the two versions of doing I/O indicating the speedup obtained when switching from blocking I/O routines to their non-blocking counterparts:

$$\frac{\textbf{non - blocking version}}{\textbf{blocking version}}$$

For more information on this ratio and the possible values consult the previous chapters.

Due to the fact that not all implementations of the MPI standards implement non-blocking I/O functions in a way which allows for real concurrency in execution, not much experience exists in its use. In order to find out if the performance gains suggested in theory acutally can be achieved in case implementations of the MPI-IO standard would allow for real non-blocking I/O, a benchmark is proposed in the following chapters. The benchmark aims at implementing a version of non-blocking I/O calls and testing the theory described above.
This is done not only to prove or disprove theory but also to see if the work needed to expand implementations of the MPI-IO standard would actually be worth the trouble and result in better execution times for the client applications.

6 A non-blocking I/O benchmark

6.1 The program

The idea of the benchmark proposed here is to measure possible performance gains by switching from using blocking I/O operations to non-blocking I/O operations. The theory of this possible performance boost is discussed in detail in chapter 5.
The idea of the implementation is to create functions representing the non-blocking I/O semantics defined by the MPI-IO standard in [2] and functionality to run different test scenarios using non-blocking I/O and blocking I/O operations. The times measured of the two versions shall be compared and the ratio `non-blocking version / blocking version` discussed in chapters 5.3.1 will be calculated. The following section will present the MPI definitions, talk about their usage and present how this functionality will be emulated in the program. It will also talk about problems with the implementation indicating limitations of possible tests and their reasons.

The programming language used will be C and the benchmark will be using the MPI interface where possible. The benchmark will test non collective write operations only. The test environment will be presented in section 6.2.

As seen in the previous discussion on different scenarios the most important scenario is the one where I/O operation and calculations in an iteration take the same amount of time. Therefore this is the first test implemented for this benchmark. The test is presented in section 6.1.3.

6.1.1 MPI-IO definitions and implemented emulations

In the paper "MPI-2: Extensions to the Message-Passing Interface"[2] published by the Message Passing Interface Forum in 1997 the chapter on I/O defines functions for non-blocking write operations. Used for this benchmark is the function `MPI_FILE_IWRITE` which allows the processes of the parallel program to write to a shared file using individual file pointers in a non collective manner. Figure 6.1 shows the definition of the MPI-IO section of [2] and the C interface definition of the same.

When a non-blocking write is started, the call to `MPI_FILE_IWRITE(...)` returns directly after initializing the operation. In order to finish the operation, i.e. to make sure that the access to the used buffer has been finished, one of the two operations `MPI_WAIT` or `MPI_TEST` needs to be called. While `MPI_TEST` returns immediately indicating the status of the operation, `MPI_WAIT` only returns after the I/O operation has been terminated. For this benchmark `MPI_WAIT` is needed and therefore is the only one of the two operations considered here. As can be seen in Figure 6.1 a handle to a `MPI_REQUEST` object is returned in order to identify the operation.

MPI_FILE_IWRITE(fh, buf, count, datatype, request)

INOUT	`fh`	file handle (handle)
IN	`buf`	initial address of buffer (choice)
IN	`count`	number of elements in buffer (integer)
IN	`datatype`	datatype of each buffer element (handle)
OUT	`request`	request object (handle)

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype, MPI_Request *request)
```

Figure 6.1: Definition MPI_FILE_IWRITE of MPI-2 standard and C interface

This handle is needed when calling `MPI_WAIT` to identify the operation one wants to wait for. As shown in Figure 6.2 the `MPI_WAIT` call then returns a `MPI_STATUS` object which can be used to see if any kind of error occured during the non-blocking write operation.

MPI_WAIT(request, status)

INOUT	request	request (handle)
OUT	status	status object (Status)

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Figure 6.2: Definition MPI_WAIT of MPI-1 standard and C interface

As said before the MPI implementation MPICH2 uses the blocking version of the I/O operation internally, calling `MPI_File_write(...)`. This is done without creating a new thread resulting in a blocking write operation which has then to be terminated by a call to `MPI_WAIT`.
In order to emulate the functionality of `MPI_File_iwrite(...)` and `MPI_Wait(...)` two functions have been implemented. Their interface is shown in Figures 6.3 and 6.4.

E_MPI_FILE_IWRITE(E_fh, E_buffer, E_count, E_datatype, E_request)

INOUT	`E_fh`	file handle (handle)
IN	`E_buffer`	initial address of buffer (choice)
IN	`E_count`	number of elements in buffer (integer)
IN	`E_datatype`	datatype of each buffer element (handle)
OUT	`E_request`	request object (handle)

```
int E_MPI_File_iwrite(MPI_File E_fh, void *E_buffer, int E_count,
               MPI_Datatype E_datatype, E_MPIO_Request *E_request)
```

Figure 6.3: Definition E_MPI_FILE_IWRITE and C interface

The emulated non-blocking I/O function `E_MPI_File_iwrite(...)` creates and starts a POSIX thread. It also collects all necessary information for the I/O operation and passes it to the started thread. A function has been implemented which then uses this information to run the blocking MPI call `MPI_File_write(...)`. The function creating the thread can return directly after the thread has been created returning an `E_MPIO_Request` object, which includes amongst

others information about the ID of the created thread. This information is then passed to the `E_MPI_Wait` function which makes sure the thread is joined with the main thread when both threads reach that point in execution. Therefore the two functions can basically be used as the definitions of MPI-2 suggest. The only thing is that due to the MPICH2 implementation no two MPI functions can be called by one process due to a giant mutex regulating this access. Therefore if for example communication functions are called in between the calls to `E_MPI_File_iwrite(...)` and `E_MPI_Wait(...)` the `MPI_File_write(...)` call and the communication call compete for the mutex and will be executed one after the other. Due to this and the fact that communication has not been part of the discussions in the previous chapter on non-blocking I/O, no communication will be done during the I/O and calculation operations.

E_MPI_WAIT(E_request, E_status)

INOUT	E_request	request (handle)
OUT	E_status	status object (Status)

```
int E_MPI_Wait( E_MPIO_Request *E_request,
                MPI_Status *E_status)
```

Figure 6.4: Definition E_MPI_WAIT and C interface

As mentioned above and indicated in the interfaces shown in Figures 6.3 and 6.4 a new object has been created: `E_MPIO_Request`. The exact definition of this object can be seen in Figure 6.5.

```
typedef struct{
  void*       E_handle;
  MPI_Status* E_Status;
  double      E_io_time;
} E_MPIO_Request;
```

Figure 6.5: Definition of E_MPI_Request

The value `E_handle` is used to pass along the thread handle in order to call the operation to join the thread with the main program in the `E_MPI_Wait(...)` function. `E_status` is used to save the status returned by the `MPI_File_write(...)` operation and is only accessible when the `E_MPI_Wait(...)` function returns.
The value `E_io_time` is used to save the time the actual call to the blocking I/o operation took in the extra thread and can also can only be accessed when completion of the joining of the threads has been verified.

The emulated functions have been tested and it has been verified that I/O and operations done between `E_MPI_File_iwrite(...)` and `E_MPI_Wait(...)` are actually executed concurrently if the underlying operating system allows for POSIX threads to do run concurrently.

6.1.2 Auxiliary functions

In addition to the emulated I/O functions presented in the previous chapter and the actual test presented in the following chapter, different auxiliary functions have been implemented. These functions are used in the test and are provided to be available in future enhancements and additions to the benchmark.

As indicated in the discussion of different scenarios in Chapter 5 the tests will be executed simulating a filled I/O buffer on the I/O server side at the beginning of the tests. In order to be able to provide this a function `write_fill_buffer(MPI_File fill_fh, int write_times)` is provided. This function writes data to the file referenced by the handle `fill_fh`. This is done `write_times` times using a buffer which can be defined using command line parameters when starting the benchmark. In order to fill the main memory of the cluster nodes hosting the I/O servers and to force the I/O servers to do physical I/O, the parameters have to be chosen accordingly depending on the execution environment.
The command line options provided for the benchmark are presented in section 6.1.4.

After deciding which amount of data is to be written in each iteration of the scenario presented in 5.3.4 it has to be figured out how many calculation operations are needed. The time to do the calculations needs to be equal to the time it takes to write the desired data to the file system when both parts are being executed concurrently.
Therefore a function `int get_workload(MPI_File fh, int workload_iterations)` has been implemented. It needs to be provided with a handle to a file used for the I/O operation and with an initial value of workload iterations to be done. The workload itself is a short series of operations which can be executed as many times as desired. The function runs the I/O operation, which is using a buffer also defined using command line options, and the calculations concurrently using the emulated functions described above and adjusts the value of `workload_iterations` until the time difference between the two operations is either smaller than a defined margin or until a maximum number of tries to achieve this is reached. A minimum number can also be set before compile time.
In order to do this for a given set of processes an average of the desired workload iterations is taken every 10 tries using the MPI communication function `MPI_Allreduce(...)`.
In the testruns desribed in the following section this function worked well and provided numbers for `workload_iterations` which turned out to have the desired effect. The write operation and calculations in each iteration of I/O and calculation differed very little.

6.1.3 Test0: I/O == calculation

The first test implemented for the benchmark is the one testing the scenario of section 5.3.4
The idea is to define an amount of data to be written in each iteration, decide how much calculation needs to be done in order to terminate both operations at the same time when executed in parallel and then run the test. The test will run the version using the emulated non-blocking I/O functions first and then run the test again with the same amount of data to be written and the same amount of calculations to be done one after the other. This is done a number of times which can be set when starting the benchmark.
In order to be able to judge how good the test is, i.e. if the test run represents the scenario in a good way, the average time differences between the non-blocking I/O operations and

the corresponding calculations is measured for each iteration. The overall average of these differences for all processes and all iterations is one of the output parameters of the benchmark. The important values measured are the total time needed to execute all iterations for the version using non-blocking I/O and the total time needed to execute all iterations for the version using the blocking I/O routine `MPI_File_write(...)`. Both times and the ratio of `non-blocking version / blocking version` are measured or calculated respectively and are also part of the program output.

In order to have comparable situations the files used for filling the I/O buffer and for the actual measured write operations are created and opened before every phase of the test. Due to this the file view is defined each time and each phase begins writing at the beginning of the file. At the end of each phase the files are closed and, if set to do so, deleted.

The four phases of the test are:

- **Preparation** of the values. In this phase the auxiliary function `int get_workload(...)` is used to figure out the amount of calculations needed to be done for the following phases. The test file is created before this and closed and, if set to do so, deleted afterwards.
- **Non-blocking test.** This phase is the testrun using the emulated write function presented in 6.1.1 is executed. This phase creates and opens the needed files and runs the test a user-defined number of times. The total execution time, the average difference between the I/O operation and the calculation of the individual iterations and average I/O and calculation times are measured.
- **Blocking test.** This phase is the testrun using the original blocking write operation `MPI_File_write(...)` provided by the MPICH2 implementation. It executes the write operation and the calculations one after the other the same amount of times as done in the phase using non-blocking I/O. The same times are measured in this phase as done in the previous one although the average time difference here is not important for the judgement of the testrun. But it can be used to see if the times of the two operations diverge in comparison to the non-blocking version.
  This phase also opens and, if set to do so, deletes the test file used.
- **Results.** This phase summarizes all times and values measured in the previous phases and creates the following output values. These are written to a file called `results.txt` by the process with the process id `0` if possible and to the standard output if not. The data is formated in a tab seperated format, providing the following information from left to right. If written to file the data is added as a new line at the end of the file in order to allow automatic execution of a series of tests which save the results to one file.
  - **reduce_p_time:** The average time needed for the version using non-blocking I/O. In the source code of the benchmark this is referred to as the parallel version with reference to concurrent execution of I/O and calculation.
  - **reduce_s_time:** The average time needed for the version using blocking I/O. In the source code of the benchmark this is referred to as the sequential version with reference to the execution of I/O and calculation one after the other.
  - **reduce_ps_time:** The average of the time differences between the non-blocking version and the blocking version of the test.
  - **reduce_p_time / reduce_s_time:** The ratio defined and presented in chapter 5.3.1:

$$\frac{\textbf{non - blocking version}}{\textbf{blocking version}}$$

- **p_reduce_average_calc_time:** The average time needed for calculations in the version using the emulated write operation.
- **s_reduce_average_calc_time:** The average time needed for calculations in the version using the blocking I/O operation `MPI_File_write(...)`.
- **p_reduce_average_io_time:** The average time needed for one execution of the write operation inside the emulated I/O operation.
- **s_reduce_average_io_time:** The average time needed for one execution of the write operation in the version using the blocking I/O routine.
- **p_reduce_average_time_difference:** The average difference of the time needed to write using the emulated function `E_MPI_File_iwrite(...)` and the time needed to exectue the corresponding block of calculations. These are times for concurrent execution of the two parts.
- **s_reduce_average_time_difference:** The average difference of the time needed to write using the blocking I/O routine and the time needed to calculate the corresponding block of calculations. These are times measured for operations exectued one after the other.
- **size:** The number of processes used in the testrun. All processes write to the same file and execute the same amount of calculations.
- **elements:** The number of elements in the buffer used for the write operations. Here each element is one `MPI_BYTE`. Therefore the number of elements is the number of bytes which are written by each process in each iteration.
- **iterations:** The number of iterations calculations and I/O are executed in the phases **Non-blocking test** and **blocking test**. Therefore the number of bytes written in total in the test to the file used for these phases can be calculated by:

$$\textbf{Bytes written to file} = \textbf{size} * \textbf{elements} * \textbf{iterations}$$

- **fill_elements:** The number of elements in the buffer used for the write operations in the auxiliary function `write_fill_buffer(...)`.
- **fill_iterations:** The amount of times each process writes to the file used to fill the I/O buffer on the I/O server side. The amount of data written to the file system each time the buffer is filled can be calculated by:

$$\textbf{Bytes written to fill file} = \textbf{size} * \textbf{fill\_elements} * \textbf{fill\_iterations}$$

The names of the values here are identical to the names of the variables in the source code used to save these values. A variable with `reduce` as part of its name always means that it is an average of the value from all processes. A `p` as part of the name implicates the value has been measured during the phase using non-blocking I/O and a `s` in the name means the value is part of the results of the phase using the blocking write operation.

6.1.4 Benchmark usage

use [...] a way in which sth can be used [6]

When running the benchmark there are a series of options which can be provided in order to influence benchmark behavior.
Also some values can be set before the source code is compiled. Furthermore different scenarios of the benchmark can be created by external influences such as the choice of number of processes used when starting the benchmark and the kind of parallel file system used as well as the number of I/O servers used in case file system like `PVFS2` is used.

<name of binary> [-d] [-f filename] [-F fill_filename] [-h] [-H hints] [-i iterations] [-s elements] [-I fill_iterations] [-e fill_elements] [-m margin]

-d	Toggle delete files on close. (Default: false)
-f	The name of the file used for the test. (Default: pvfs2:///pvfs2/test_file)
-F	The name of the file used to fill I/O server buffer. (Default: pvfs2:///pvfs2/fill_file)
-h	Display help.
-H	Hints provided to the MPI-IO calls in the form key=value.
-i	Number of times the test repeats I/O and calculation blocks. (Default: 10)
-s	Number of elements of buffer used to write to test file. (Default: 10485760)
-I	Number of times each process writes to I/O server buffer. (Default: 10)
-e	Number of elements of buffer used to fill I/O server buffer. (Default: 104857600)
-m	Margin acceptable for difference in I/O and calculation times in seconds. (Default: 0.02)

Figure 6.6: Command line options of benchmark

The two values needed to be set before compiling the source code are the values `MAX_TRIES` and `MIN_TRIES` which are used in the auxiliary function `int get_workload(...)` presented in chapter 6.1.2.
When figuring out how many iterations of the calculation part are needed to last as long as the concurrently executed I/O operation, the function iteratively adjusts the original value. This is done at least `MIN_TRIES` times and until the average time difference on each process has dropped under the defined margin (see below) but at most `MAX_TRIES` times.

A lot of different parts of the benchmark can be influenced by use of command line arguments. These include buffer sizes, number times the test repeats I/O and calculation blocks and the amount of data written to fill the buffer of the I/O system. Furthermore the margin used for the function calculating the workload can be set. Figure 6.6 explains the possible parameters in more detail. The defaults here can be changed in the source code before it is compiled.

6.2 The test environment

The test environment for this paper will be computer cluster located at the `Ruprecht Karls University of Heidelberg, Germany`. It is the research cluster of the research group `Parallel and Distributed Systems`.

The cluster consists of 10 nodes. One node acts as the master node, hosts the home directories of the cluster users and is the access point for users to the cluster. The nodes, named node1 to node9, can be split into compute nodes (node6 to node9) and especially equipped I/O nodes (node1 to node5). The cluster runs Debian Sarge (Linux 2.6.19-5-pvs) as the operating system. The details of the setup are as follows:

Common components

- Two Intel Xeon 2GHz CPUs
- Intel Server Board SE7500CW2
- 1 GB DDR-RAM
- 80GB IDE HDD
- CD-ROM Drive
- Floppy Disk Drive
- Two 100-MBit/s-Ethernet-Interfaces. These interfaces are not used)
- Two 1-GBit/s-Ethernet-Interfaces. One of these interfaces is used for the cluster network.
- 450 Watt Single Power Supply

Special hardware of master node

- 80 GB IDE HDD used to host the users home directories. The home directories are accessible on all nodes by use of the network file system NFS.
- 2nd 1-GBit/s-Ethernet-Interface used for external network connection. This connection is the only access point for users to the cluster.

Special hardware of I/O nodes

- RAID-Controller Promise FastTrack TX
- RAID0 (Striping): Two 160 GB S-ATA HDDs. This raid system is used for the I/O system of the test runs in this paper. The pvfs2-servers can be run on node1 to node5 in order to provide disk space on these hardware I/O devices.

Special hardware used in the cluster network

- D-Link DGS-1016: The cluster nodes are connected using the 1-GBit/s-Ethernet-Interface connected through a D-Link DGS-1016 switch. The switch supports 10,100 and 1000 Mbit connections and has 16 ports available.

No additional hardware is used for the compute nodes (node6 to node9).

7 Testing

> **test** [...] a trial or an experiment intended to show whether sth works or works well [6]

7.1 Planning the tests

> **plan** [...] to consider sth in detail in advance [6]

The previous chapters of this paper gave a short introduction to high performance cluster computing and presented the idea of benchmarking. Then the concept of non-blocking I/O operations was presented and different scenarios have been discussed. All these scenarios examined situations in applications where the use of non-blocking I/O in theory promises a performance gain in comparison to the same application using blocking I/O routines. A benchmark has been implemented and described which is supposed to be used as a test to show whether this method works in order to reduce overall execution times. Interesting of course is not only the question if the execution time of an application can be reduced but also if the potential suggested in theory can be exploited well.
The presentation of the available parameters one can use to tune the benchmark behavior together with the presented scenarios indicates that a lot of different tests can be planned and executed. In order to find out if the use of non-blocking I/O really can provide the benefits suggested in theory and if so how much really is possible, a series of tests is planned and presented in the following chapters.
Since these are the first tests to be done here, they are all based on the scenario presented in chapter 5.3.4 and aim at verifying the theory, the fact that the program can actually represent the needs of applications of this scenario and at showing what can actually be achieved.

7.1.1 The client side

When planning the test runs and looking at the client side of the application, a lot of decisions have to or can be taken. These include amongst others how much data should be written, how often and, more important for the first tests, how many processes should take part in the benchmark.
When looking at the available test environment described in section 6.2 one can see that the nodes used for computation have 2 CPUs each and that 4 nodes are available. While each node can also be configured to use only one CPU this setup is good for the first round of tests. The second set of tests will be run on nodes configured to be single processor nodes and will be representing the scenarios where the performance boost comes by making use of unused CPU cycles during the I/O operation. Both versions will be discussed in a little more detail in the following paragraphs.

1 Client per 2-CPU node

Running one client process on a seperate node providing 2 CPUs each will help to prove if a performance boost can actually be achieved and will represent applications which for reasons of memory usage can only use 1 process per node even when 2 CPUs per node are provided.

For this set of tests 3 different numbers of processes have been selected. **1 process** will show if a performance boost can be achieved without having more than one process access the file used to save the data. **2 and 4 processes** will show if the total execution time can be reduced when writing to one file using more than one process.

Verifying the program

This set of tests is above all used to verify theory and the question if the implemented benchmark can actually represent it.
The discussions on the different scenarios in chapter 5 often assumed that when switching from blocking I/O to non-blocking I/O, the times for the write operation and the blocks of calculation respectively do not change. This is very unlikely if not impossible when running the tests on a node where both threads have to use 1 CPU. But in case 2 CPUs are provided and are available for the one process only, both threads can use all CPU cycles of one of them in both phases of the benchmark. This setup therefore best represents the idea of the theory using this assumption.

Due to the fact that less changes in these times are expected in this setup, it can also be used to see how much of the promised reduction in execution time can really be obtained with this benchmark. The hope is for some testrun to provide a ratio of `non-blocking version / blocking version` of `0.5` to make sure that it is not the fault of the benchmark in case this is not achieved when running the other tests.

1 Client per 1-CPU node

For scenarios which are more CPU intensiv and which allow only for one CPU per process the times of each operation will most likely change between the two phases of the benchmark. Here both threads executing in parallel have to make use of one CPU and compete for the available CPU cycles. For this set of tests the same numbers of processes will be used as in the previous setup.
This setup probably is the more realistic one representing the majority of the applications being run on computer clusters and having to do I/O regularly. The hope here is that the difference between performance gains here and performance gains in the previous setups and in theory are small.

7.1.2 The I/O-server side

When looking at the I/O server side of the benchmark the configuration of the parallel file system used has be decided upon.
In this case here the parallel file system PVFS2 is used and the test environment provides for 5 nodes which can be used as I/O nodes in this case. Since for the client side the number of processes used are 1, 2 and 4 it has been decided to use the same numbers for the number of nodes providing the storage devices and running the pvfs2-servers.
Important for the testruns is that the amount of data written to fill the memory available on the nodes hosting the I/O servers is enough to fill the memory all pvfs2-server nodes used.

7.1.3 Summary of planned tests

Taking all this into consideration the number of tests planned is a combination of 3 numbers of client processes, 2 setups of CPUs per node and 3 numbers of nodes hosting I/O servers.

Number of tests = 3 * 2 * 3 = 18

In addition to these tests one more setup on the client side will be considered and tested.
Using 2 CPUs per node 8 processes will be used. For this setup only 2 and 4 I/O nodes will be tested.

For each scenario a series of tests will be run and in order to judge the quality of the single test runs, i.e. how exact the individual test run finds a workload matching the I/O in the non-blocking phase of the test, only test runs are judged which have a ratio of `average time difference / average I/O time in blocking version` of less than `0.11`.
This is done because the test runs with a bigger ratio here do not represent the scenario well enough.

7.2 Test runs and their results

This chapter will present the results of all the test runs made. The tests have been prepared in the previous chapter and their results will be shown here individually. The following chapter 8 will compare the results with the expectations discussed in the chapter on non-blocking I/O operations and their semantics (for more information on this see 5). It will also be responsible for a more detailed overview of the results and will attempt to present a ranking of the tests. Here the focus is more on individual tests and their results.

In order to get a first idea of the results Figures 7.1 and 7.2 show a first summary of the obtained results. Figure 7.1 presents the results for the tests run with 1 process on a node providing 2 CPUs and figure 7.2 presents the results for the tests where each process is executed on a cluster node running with 1 CPU.
Both figures show the ratio of `non-blocking version / blocking version` for each results.

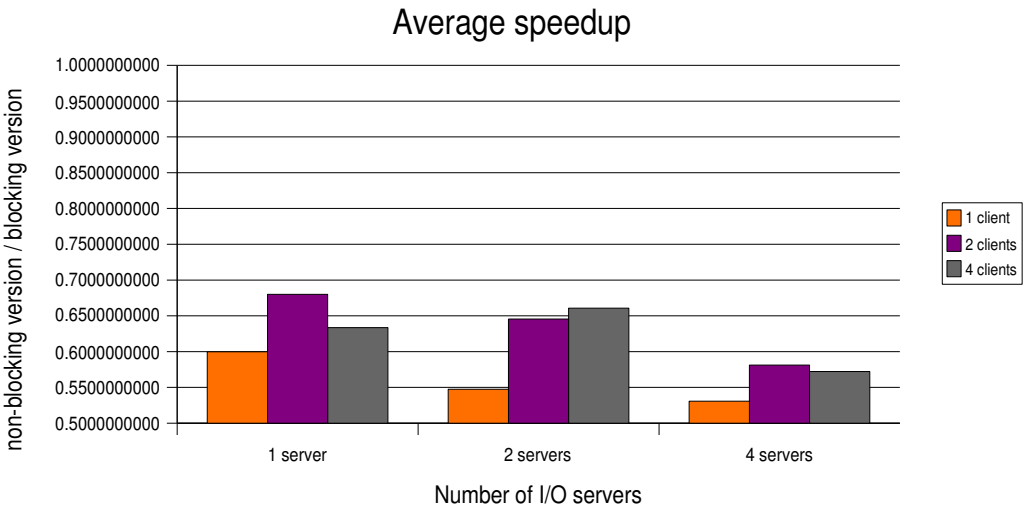

Figure 7.1: Overview over test results for test with 2 CPUs per process

Important here is that a **smaller number** means a **greater performance boost** has been obtained.

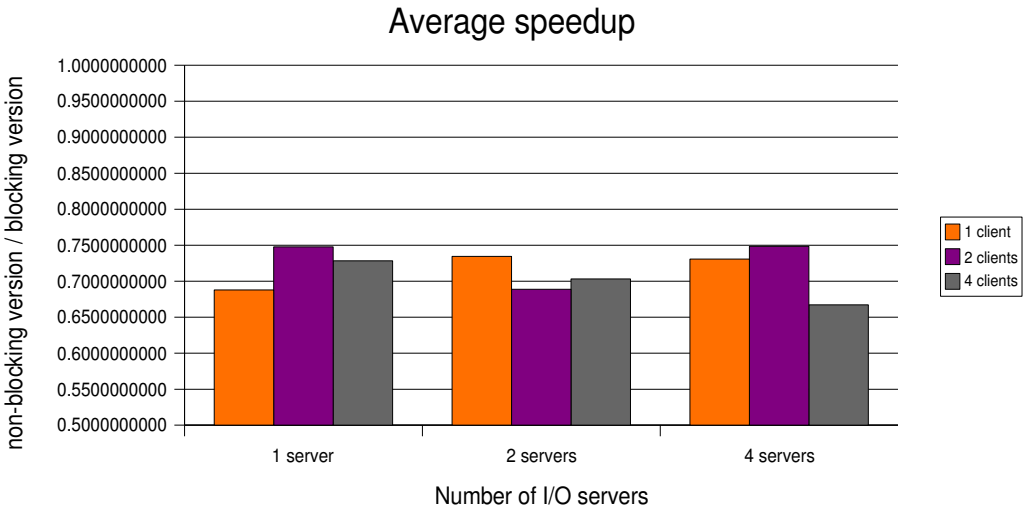

Figure 7.2: Overview over test results for test with 1 CPUs per process

As can be seen already the results show that performance gains are really achievable and the benchmark provided can get close to the optimum as shown in figure 7.1 for the test using 1 client and 4 I/O servers.

For a view of the tests different amounts of data to be written have been chosen and have been tested. Since the results of different buffer sizes to be written didn't seem to provide great differences in performance gains they have not been distinguished in the presentation of the results. The results of these runs mixed with each other when looking at one test executed with different amounts of data to be written.

Presented will be the following information for the series of tests made for each version:

- **Average ratio** `non-blocking version / blocking version`**:** This value is the indicator for how much execution time has been saved for the test scenario when changing from blocking I/O to non-blocking I/O . This average is the average over all times the test has been executed. For more information on this number see chapter 5.3.1.
- **Maximum ratio** `non-blocking version / blocking version`**:** This value is the result of the test execution run which showed the worst test result. The **maximum ratio** is identical to the **least performance boost**.
- **Minimum ratio** `non-blocking version / blocking version`**:** This value is the result of the test execution run which showed the best test result. The **minimum ratio** is identical to the **best performance boost** seen during repetition of the individual test.
- **Average ratio** `average time difference / average I/O time in blocking version`**:** This value indicates the quality of the test runs. It is the ration between `average time difference between I/O and calculation in the non-blocking phase of the benchmark` and `average I/O time in the non-blocking phase of the benchmark`. Therefore it shows how many percent of the average I/O time in the non-blocking test the I/O time and calculation time differed in average. The average is taken over all tests run for the corresponding setup.
- **Maximum ratio** `average time difference / average I/O time in blocking version`**:** This value indicates the quality of the worst test run for the corresponding setup.
- **Minimum ratio** `average time difference / average I/O time in blocking version`**:** This value indicates the quality of the best test run for the corresponding setup.
- **Ratio** `calculation time non-blocking version / calculation time blocking version`**:** This value indicates the change of execution time for the calculation part of the benchmark. It is an average over all executions of one test. The value is to be interpreted as follows:
  - A number **greater than 1** indicates that the calculation takes longer in the non-blocking version. The greater the value the greater the time difference.
  - A number **equal to 1** indicates that the calculation takes an equal amount of times in both test phases.
  - A number **smaller than 1** indicates that the calculation can terminate faster when non-blocking I/O is used.
- **Ratio** `I/O time non-blocking version / I/O time blocking version`**:** This value indicates the change of execution time for the I/O part of the benchmark in the two phases using non-blocking I/O and blocking I/O routines. It is an average over all executions of one test. The value is to be interpreted the same way as the previous value for the difference in calculation times.

The maximum and minimum values are provided in order to see in between which range the values fluctuated.

7.2.1 1 Client per 2-CPU node

This section will present the results of the test runs for the scenarios using 2 CPUs on each node and running 1 process per node.
The performance gain ratio is between `0.5307` for 4 I/O servers and 1 client and `0.6800` for 1 I/O server and 2 clients.

1 I/O server - 1 client process

Average ratio	**0.5996**
Minimum speedup	0.6680
Maximum speedup	0.55137
Average quality	0.001039
Minimum quality	0.002709
Maximum quality	0.000006344
Ratio of calculation times	1.007591
Ratio of io times	1.03158

1 I/O server - 2 client processes

Average ratio	**0.6800**
Minimum speedup	0.7781
Maximum speedup	0.5962
Average quality	0.01858
Minimum quality	0.1080
Maximum quality	0.00009491
Ratio of calculation times	1.007822
Ratio of io times	1.8051

1 I/O server - 4 client processes

Average ratio	**0.6335**
Minimum speedup	0.7229
Maximum speedup	0.5740
Average quality	0.01870
Minimum quality	0.09406
Maximum quality	0.000004928
Ratio of calculation times	0.9983
Ratio of io times	1.4639

2 I/O servers - 1 client process

Average ratio	**0.5474**
Minimum speedup	0.5508
Maximum speedup	0.5455
Average quality	0.06381
Minimum quality	0.07299
Maximum quality	0.04719
Ratio of calculation times	1.0972
Ratio of io times	1.009457

2 I/O servers - 2 client processes

Average ratio	**0.6456**
Minimum speedup	0.7447
Maximum speedup	0.6064
Average quality	0.05619
Minimum quality	0.0875
Maximum quality	0.007106
Ratio of calculation times	1.04926
Ratio of io times	1.2079

2 I/O servers - 4 client processes

Average ratio	**0.6607**
Minimum speedup	0.6888
Maximum speedup	0.6394
Average quality	0.03496
Minimum quality	0.1027
Maximum quality	0.005776
Ratio of calculation times	1.03191
Ratio of io times	1.3512

4 I/O server - 1 client process

Average ratio	**0.5307**
Minimum speedup	0.5329
Maximum speedup	0.5295
Average quality	0.002536
Minimum quality	0.003890
Maximum quality	0.0008469
Ratio of calculation times	1.1143
Ratio of io times	0.9987

4 I/O server - 2 client processes

Average ratio	**0.5812**
Minimum speedup	0.5924
Maximum speedup	0.5720
Average quality	0.0682
Minimum quality	0.1091
Maximum quality	0.009083
Ratio of calculation times	1.1073
Ratio of io times	1.05680

4 I/O server - 4 client processes

Average ratio	**0.5722**
Minimum speedup	0.6092
Maximum speedup	0.5447
Average quality	0.05869
Minimum quality	0.1016
Maximum quality	0.0008741
Ratio of calculation times	1.07032
Ratio of io times	1.0714

7.2.2 1 Client per 1-CPU node

This section will present the results of the test runs for the scenarios using 1 CPU on each node and running 1 process per node.
The performance gain ratio is between `0.6671` for 4 I/O servers and 4 clients and `0.7484` for 4 I/O servers and 2 clients.

1 I/O server - 1 client process

Average ratio	**0.6878**
Minimum speedup	0.7022
Maximum speedup	0.6779
Average quality	0.02621
Minimum quality	0.06697
Maximum quality	0.004474
Ratio of calculation times	1.08635
Ratio of io times	1.5089

1 I/O server - 2 client processes

Average ratio	**0.7477**
Minimum speedup	0.7579
Maximum speedup	0.7304
Average quality	0.04449
Minimum quality	0.06699
Maximum quality	0.03350
Ratio of calculation times	1.05761
Ratio of io times	1.9682

1 I/O server - 4 client processes

Average ratio	**0.7282**
Minimum speedup	0.7557
Maximum speedup	0.6951
Average quality	0.01933
Minimum quality	0.02690
Maximum quality	0.01492
Ratio of calculation times	1.04296
Ratio of io times	1.9720

2 I/O servers - 1 client process

Average ratio	**0.7345**
Minimum speedup	0.7394
Maximum speedup	0.7314
Average quality	0.08720
Minimum quality	0.0985
Maximum quality	0.07429
Ratio of calculation times	1.2925
Ratio of io times	1.5230

2 I/O servers - 2 client processes

Average ratio	**0.6887**
Minimum speedup	0.7124
Maximum speedup	0.6646
Average quality	0.02330
Minimum quality	0.03195
Maximum quality	0.01874
Ratio of calculation times	1.1861
Ratio of io times	1.3360

2 I/O servers - 4 client processes

Average ratio	**0.7031**
Minimum speedup	0.7873
Maximum speedup	0.6205
Average quality	0.04264
Minimum quality	0.08455
Maximum quality	0.009949
Ratio of calculation times	1.1695
Ratio of io times	1.4517

4 I/O servers - 1 client process

Average ratio	**0.7308**
Minimum speedup	0.7329
Maximum speedup	0.7290
Average quality	0.06618
Minimum quality	0.07484
Maximum quality	0.05927
Ratio of calculation times	1.4495
Ratio of io times	1.3642

4 I/O servers - 2 client processes

Average ratio	**0.7484**
Minimum speedup	0.7529
Maximum speedup	0.7432
Average quality	0.08203
Minimum quality	0.09922
Maximum quality	0.06636
Ratio of calculation times	1.3288
Ratio of io times	1.5259

4 I/O servers - 4 client processes

Average ratio	**0.6671**
Minimum speedup	0.6748
Maximum speedup	0.6619
Average quality	0.07209
Minimum quality	0.07950
Maximum quality	0.06463
Ratio of calculation times	1.2501
Ratio of io times	1.2962

7.2.3 Extra: 8 clients on 4 2-CPU nodes

This section will present the results of the test runs for the scenarios using 2 CPU on each node and running 2 processes per node.
The performance gain ratio is between `0.7246` and `0.7360`. All tests here are using 8 clients on 4 compute nodes.

2 I/O servers - 8 client processes

Average ratio	**0.7246**
Minimum speedup	0.7576
Maximum speedup	0.7021
Average quality	0.03180
Minimum quality	0.06695
Maximum quality	0.009745
Ratio of calculation times	1.0580
Ratio of io times	1.7273

4 I/O servers - 8 client processes

Average ratio	**0.7360**
Minimum speedup	0.7823
Maximum speedup	0.6735
Average quality	0.05417
Minimum quality	0.08923
Maximum quality	0.03100
Ratio of calculation times	1.1415
Ratio of io times	1.6815

8 Comparing expectations with results

Possible perfomance gains by switching from blocking I/O routines to non-blocking I/O routines have been discussed. A benchmark has been implemented and run in order to see how actual results would be in comparison to the values expected after looking at the theory. The individual result of the test runs have been presented and each scenario has shown at least some kind of performance gain as presented in chapter 7.2. In order to get a better overview of the results presented, this chapter will sum up results and discuss a few different numbers. It will sum up what as been seen and will compare the results to the expectations presented in chapter 5.

All in all the expectations for all scenarios tested suggested that the possible performance boost can be described by the ratio of `non-blocking version / blocking version` and that the expectations of all scenarios are expected to be in between the values `0.5` and `1.0` for this number. This means that performance gains were expected for all scenarios which has been proven to be correct by the tests presented in the previous chapter. All average ratios have been inside this interval.

$$\mathbf{0.5} < \frac{\mathbf{non-blocking\ version}}{\mathbf{blocking\ version}} < \mathbf{1.0}$$

Figure 8.1: Expected interval for results

In the following section will discuss the results which will be split up into the two goups made up by number of CPUs used for one benchmark process. It will try to analyze trends in the result and talk about changes in times for I/O and calculation respectively when switching between the two I/O versions.
The final section of this chapter will present a list of all results sorted by the average ratio of `non-blocking version / blocking version`.

8.1 Comparison of individual results

8.1.1 1 Client per 2-CPU node

This series of tests was presented in chapter 5.3.4 and made the assumption that there are no times for either I/O or calculation parts when switching from one version of I/O to the other. It also represents the applications which have 2 CPUs available for each application process. Due to this fact the results were expected to be close to the maximum speedup possible.
The test results of these tests also were expected to show that if the scenario allows for maximum speedup, the benchmark can actually achieve that.

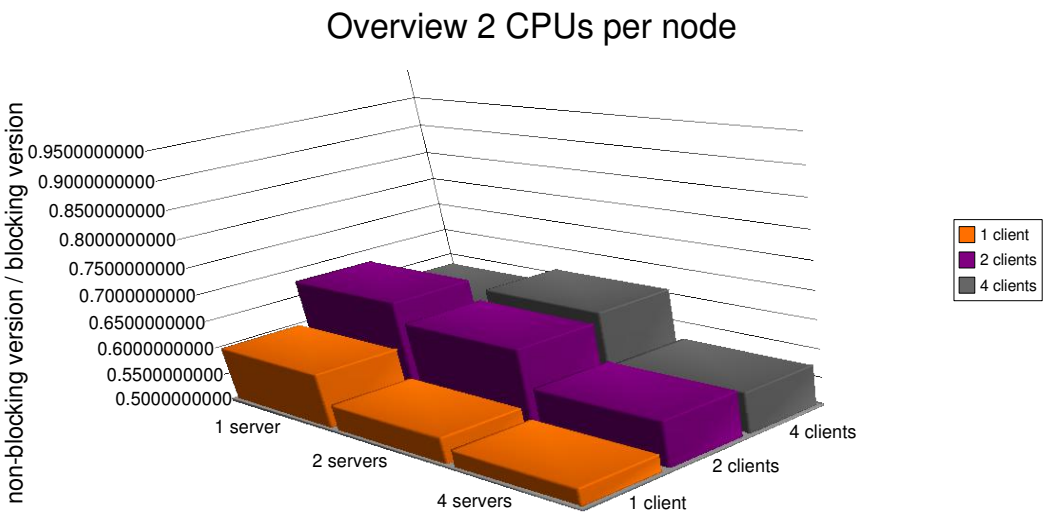

Figure 8.2: Results for tests run on nodes providing 2 CPUs for 1 process

The latter has been shown with the results of the tests using 4 I/O servers and 1 client process which itself was executed on a node providing 2 CPUs. The best result here was a ratio of `0.5295` and the average speedup turned out to be not much worse with a ratio of `0.5307`. Also the other test runs using 2 CPUs per node and more than one client process were not too far of the possible maximum.

Figure 8.2 shows all results for these tests.

All results were better than a ratio of `0.6800` and over half of the results proved to be lower than `0.6`.

Interesting are the numbers representing the changes in I/O times and calculation times respectively when switching from blocking I/O to non-blocking I/O. While most of the numbers suggest an increase in times it is very small in most cases. A few test results even implicate that the execution of one part has been faster when using non-blocking I/O. This of course is not possible in theory but those really small time diffences can be the result to external influences as CPU use by the operating system or monitoring tasks on the cluster. This explains how in individual cases a lower time can be seen for those cases.

Using trace tools provided with the MPICH2 implementation of the MPI standards and the work of the resarch group `Parallel and Distributed Systems` at the `Ruprecht Karls University, Heidelberg`, who added a lot of functionality concerning traces on the I/O side of an application, the following figures 8.3 and 8.4 have been created.
Figure 8.3 shows a few iterations of the blocking phase of a test run using 1 I/O server and 2 client processes. As this group of test runs implicates, 2 CPUs have been available on each node running one of the two processes.
Figure 8.4 shows a few iterations of the non-blocking phase of the same test run.

The upper two lines represent the two client processes, the lower line the behavior of the I/O server.
The orange phases on the client side represent the time in which calculation takes place. The

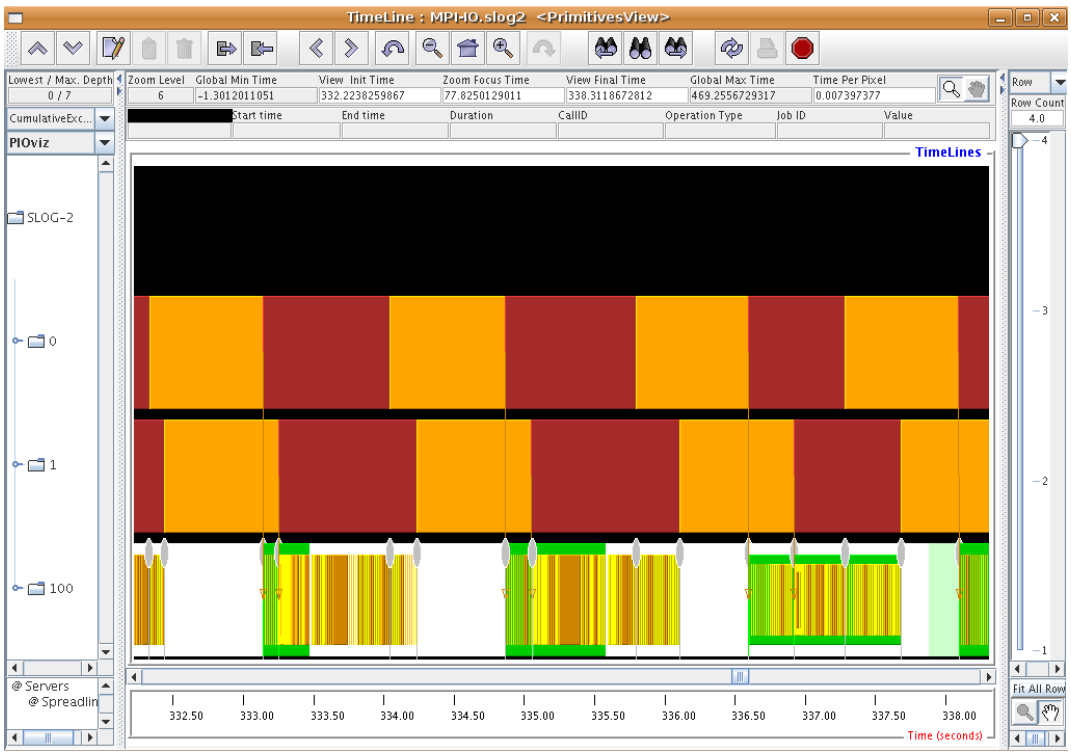

Figure 8.3: 2 CPUs per node - blocking I/O phase of a test using 1 I/O server and 2 client processes

dark red phase represents the time the client process spends inside the call to `MPI_File_write(...)`. While this call is done in the usual way in the blocking version, `MPI_File_write(...)` is called inside the `E_MPI_File_iwrite(...)` function implemented for the presented benchmark.
Due to the fact that concurrent events in one client process are presented one behind the other, the calculation part in the non-blocking version is hidden mostly behind the I/O part. Nevertheless one can see the calculation part in between some of the I/O iterations, showing that both times are in fact more or less equal.

One can see nicely that the I/O phases take about the same amount of time in each version. The same goes for the calculation blocks. This proves nicely the theory of no time change in between the two versions when 2 CPUs per process are provided.

All in all these numbers show that when using 2 CPUs for one process of the benchmark the theoretical behavior of no time change is represented and the average ratios show that results close to the expected optimum can be achieved.

8.1.2 1 Client per 1-CPU node

The group of tests executed in a test environment which provides 1 CPU per node used for computation and running 1 benchmark process on each compute node is the more interesting of the two test groups. Most applications are expected to not have the luxury to use 2 CPUs on one node for each process executed on that node.
These tests were presented in chapter 5.3.4 and did not make the assumption that the times

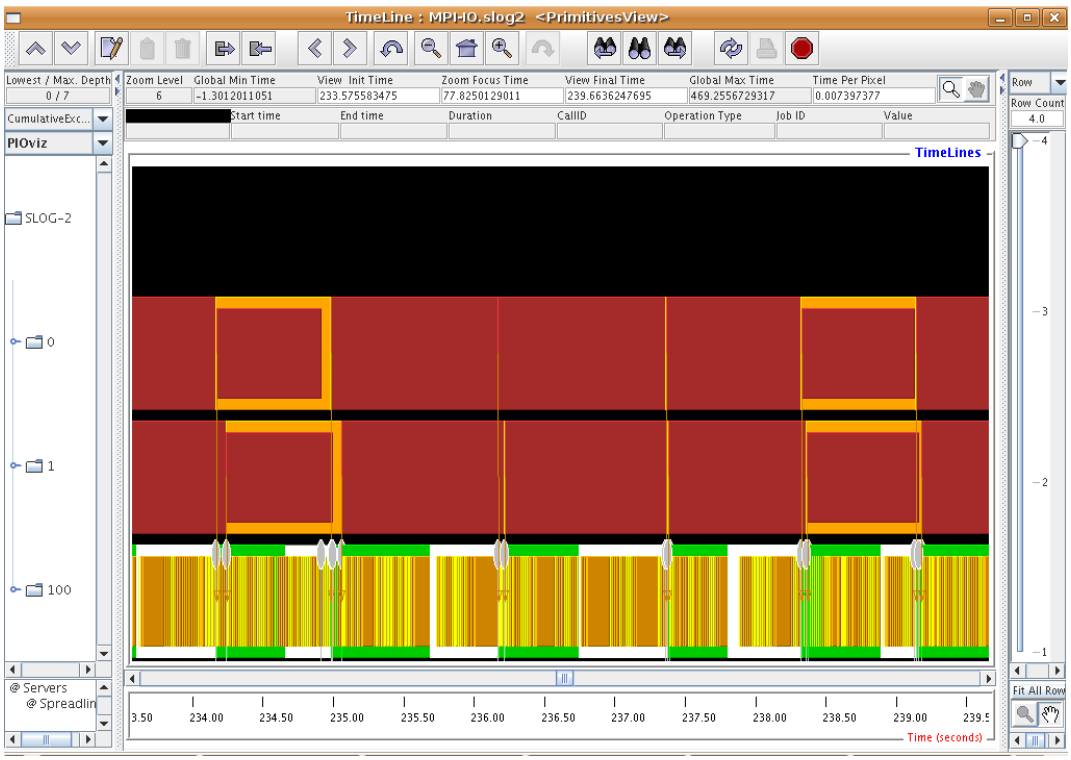

Figure 8.4: 2 CPUs per node - non-blocking I/O phase of a test using 1 I/O server and 2 client processes

needed to complete the two parts of the benchmark stay the same when being executed in parallel. Here the hope was for unused CPU cycles during the I/O operation which could be used by the calcualtion part. These free CPU cycles arise from use of techniques like direct memory acces and would be waisted waiting when using blocking I/O operations.

Due to times for the I/O server to free memory for caching data to be written during times when the client processes do calculations in the blocking version of the benchmark, the I/O operations are expected to finish faster in this version than when non-blocking routines are used. Also overhead arising from the need to switch between the threads during execution on one processor was expected to influence execution time differences.
Nevertheless the theory still allowed for performance gains to be achieved by making use of the non-blocking write operation, even when these gains were expected to be smaller than in the previously presented test group.

Figure 8.5 gives an overview of the results of this test group.

As already has been shown when looking at the individual scenario results, all tests have provided some kind of speedup. The least speedup still presenting an average ratio of `non-blocking version / blocking version` lower than `0.75`. While the best result with at ratio of `0.6671` is not as good as the best results shown in the previous test group, it is still better than the worst result there.

Interesting again is the change in time needed for I/O and calculations respectively when changing the I/O version. While in the previous test group the changes have been small to

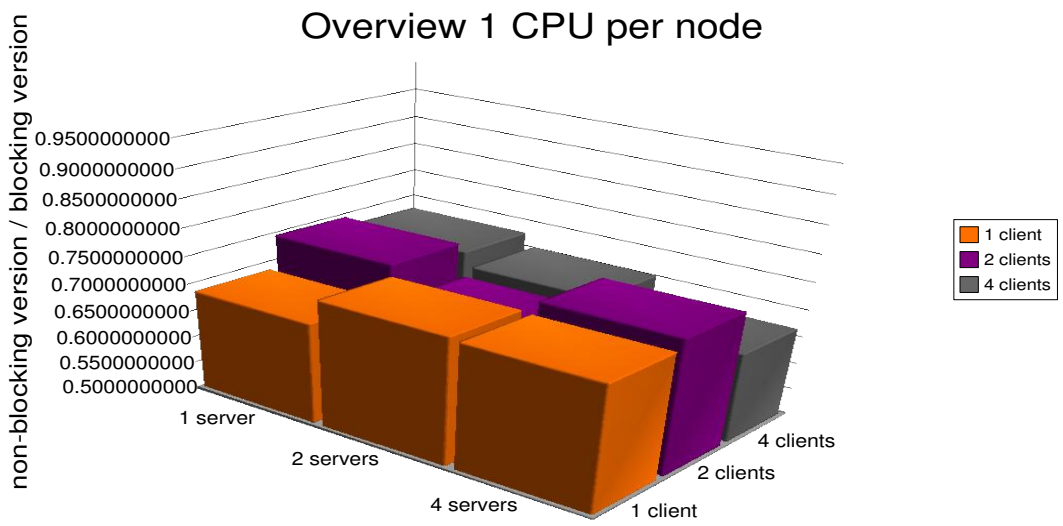

Figure 8.5: Results for tests run on nodes providing 1 CPU for 1 process

none, this test group shows, as expected, bigger differences.
Especially the I/O times increased when changing from blocking I/O to the non-blocking write operation. In most cases the I/O time ratio was even bigger or equal to `1.5`.
While calculation time changes show less alterations, they are still changing more than in the tests using 2 CPUs for each benchmark process.

Again examples of the execution of a test of this group of tests is provided in figures 8.6 and 8.7. Here each process was, as implicated by this group of tests, executed on a node providing 1 CPU. The test run used 4 I/O servers and 4 client processes.
The first 4 lines labeled 0, 1, 2 and 3 represent the behavior on the client side. One can see again the calculation part represented by the orange blocks and the write operation represented by the dark red blocks.
The last 4 lines labeled 100, 101, 102 and 103 represent the behavior of the 4 I/O servers.

As the numbers in the results show one can see that the I/O times are larger in the non-blocking version. Here the time differences show an increase ratio of `non-blocking I/O time / blocking I/O time` of about `1.25`. The exact numbers of the test run shown here are presented in the following table.

Non-blocking version / blocking version	**0.6494**
Quality ratio:	0.08234
Ratio of calculation times	1.2394
Ratio of io times	1.2408

But while the indivdual times do increase, in some cases more than just a little, does the use of non-blocking I/O still provide a performance gain when looking at the overall execution times.

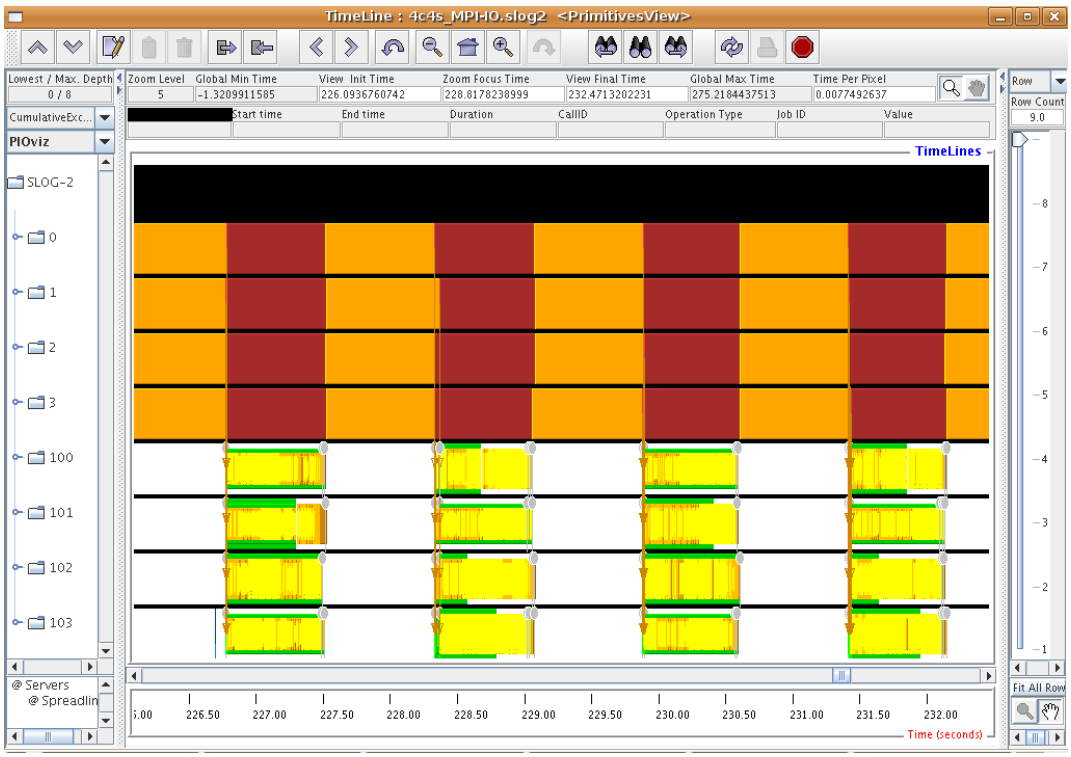

Figure 8.6: 1 CPU per node - blocking I/O phase of a test using 4 I/O servers and 4 client processes

8.1.3 Extra: 8 clients on 4 2-CPU nodes

As a mixture of the two previous test groups a few tests have been run which use compute nodes providing 2 CPUs each. These tests ran 2 application processes on each node using 4 compute nodes in total. The 8 benchmark processes used 4 I/O servers in one test group and 2 I/O servers in the other.
The overview of the test results of the two test series was given in chapter 7.2.3.

Due to the fact that each benchmark process has the same amount of CPU resources, i.e. 1 CPU per process, as in the test group providing 1 CPU per node the expectations were that the performance gains would be somewhere in the same area as there.
But while the same amount of CPU resources per process was provided, one cannot neglect the fact that another important cluster resouce changed in this scenario. Here two processes had to use the same network interface card and therefore share the network resources which in the other tests were available for each process. This could influence execution behavior since here more data has to be sent to the I/O server throught the same connection as before.

The numbers presented in the previous chapter show that the results of this group of tests really is in the scope of the test runs using 1 CPU per node. While the calculation times only change a little bit, the I/O times grow quite a bit when switching from blocking I/O to the non-blocking write operation.

All in all one can say that for this still small number of benchmark processes and the relatively large number of I/O servers provided for those the network connection provided was still able

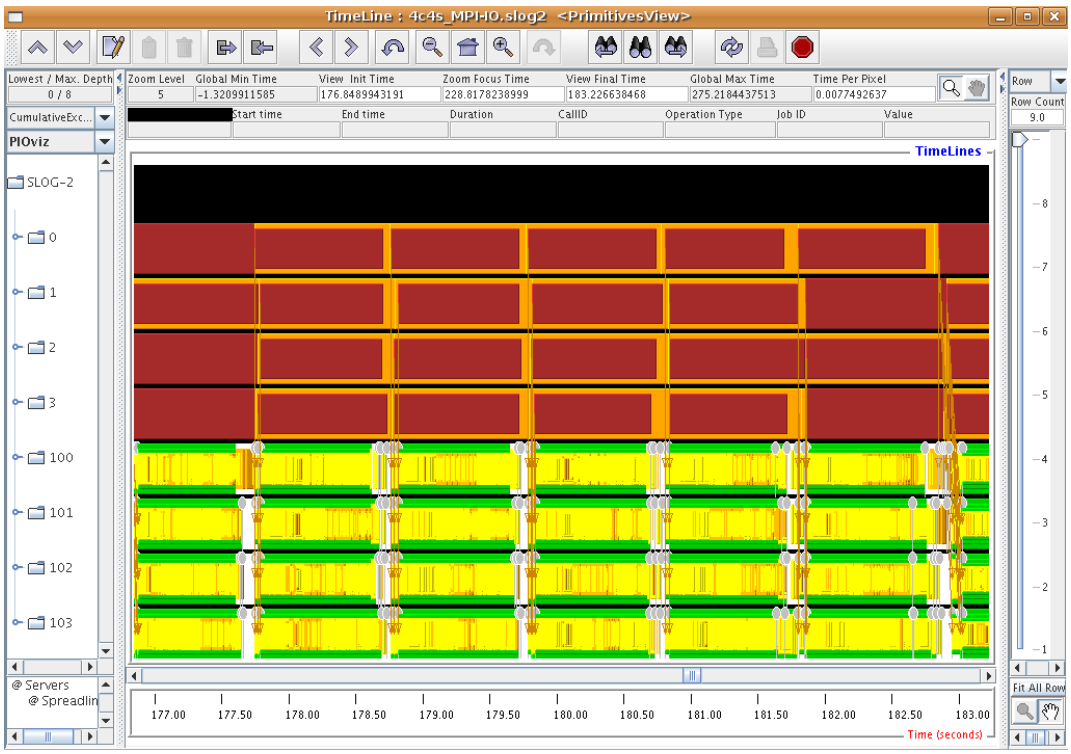

Figure 8.7: 1 CPU per node - non-blocking I/O phase of a test using 4 I/O servers and 4 client processes

to satisfy the needs of the benchmark in a way which allows for non-blocking I/O operations to speed up overall execution times in comparison to their blocking counterparts. As suggested in chapter 10 the use of more processes and if available the use of an symmetric multiprocessor environment providing more than 2 processors per node can be an interesting scenario for future examinations.

8.2 Ranking of results

The following list presents all test scenarios ordered by the average result all executions of one test scenario provided.

Number of CPUs per node	Number of I/O servers	Number of client processes	Average results
2	4	1	0.5307490000
2	2	1	0.5474842500
2	4	4	0.5722641250
2	4	2	0.5812520000
2	1	1	0.5996916842
2	1	4	0.6335783243
2	2	2	0.6456498333
2	2	4	0.6607702500
1	4	4	0.6671947500
2	1	2	0.6800488333
1	1	1	0.6878722000
1	2	2	0.6887297500
1	2	4	0.7031473333
2	2	8	0.7246107500
1	1	4	0.7282810000
1	4	1	0.7308408000
1	2	1	0.7345192000
2	4	8	0.7360940000
1	1	2	0.7477625000
1	4	2	0.7484068000

9 Non-blocking I/O on "Poor people's clusters"

9.1 Theory and expectations

While all previous discussions of non-blocking I/O operations and all previous tests have been done assuming that the cluster used has a dedicated part for hosting the I/O system, this is not always the case.
Small research groups, different departments of businesses or private people might not have the resources to by and configure a big cluster and to equip part of it with special I/O devices. These groups might just connect some of the shelf computers and install open source software to create a small personal cluster without special hardware for e.g. a RAID system. The applications run on small clusters might need all available processors and especially need to make use of the hard discs provided in the cluster nodes. Even if the cluster nodes might be equipped with a RAID system it might be necessary to use all nodes to run all parts necessary: The I/O servers and the client processes.

In order to find out if setups like this can also make use of non-blocking I/O operations, the benchmark has been run on a subpart of the available cluster using only the I/O nodes to host both the pvfs2-servers and the benchmark processes.
First these nodes were used providing 2 CPUs each and then some tests have been run with all parts of the benchmark using one CPU on each node only. In the non-blocking phase this now means that I/O server, calculation thread and I/O thread need to compete for the single CPU resource.
The results of the tests will be presented in the following sections.

Before the tests are run and results are looked at some points about expectations for these tests should be mentioned.
The maximum speedup possible of course does not change in theory and it might be possible that the speedup matches the ratio of `0.5`. The results presented this paper so far suggest that this of course might not be the case and that there are reasons to expect a behavior worse than the best case.
The previous tests showed that the calculation part of the benchmark can use time during the I/O part when using non-blocking I/O operations in order to provide a better total execution time. This can be done even when the two threads compete for one available CPU. The question in the scenario where I/O server and client processes are executed on the same nodes is whether the additional need for CPU by the I/O server still enables this or if blocking I/O would be the better choice in this situation. But while the additional CPU usage by the I/O server influences the non-blocking phase of the benchmark, it also influences the part using blocking I/O operations. The overall time needed to run an application on a cluster with less resources such as only 1 CPU per node and the fact that the I/O server also competes for those, will

increase. Therefore it might be possible that while overall execution time increases, the use of non-blocking I/O still can help in keeping it from increasing to far.
The following tests will show if the execution of all parts of the system on the same nodes will still bring perfomance gains and if so, how much.

9.2 Results

9.2.1 2-CPU nodes with 1 I/O-server and 1 client process

The following tables show the results for the test runs using 2 CPUs per node and executing benchmark and I/O server on the same nodes of the cluster. The results show that this scenario also allows for reduction in execution times by switching from blocking I/O to the non-blocking write operation.

2 I/O servers - 2 client processes

Average ratio	**0.7556**
Minimum speedup	0.7760
Maximum speedup	0.7355
Average quality	0.03411
Minimum quality	0.05782
Maximum quality	0.009799
Ratio of calculation times	1.4314
Ratio of io times	1.1735

4 I/O servers - 4 client processes

Average ratio	**0.7899**
Minimum speedup	0.8300
Maximum speedup	0.7637
Average quality	0.07003
Minimum quality	0.08789
Maximum quality	0.03590
Ratio of calculation times	1.5052
Ratio of io times	1.3172

9.2.2 1-CPU nodes with 1 I/O-server and 1 client process

The following tables show the results for the test runs using 1 CPU per node and executing benchmark and I/O server on the same nodes of the cluster.
The results show that again a reduction in application execution time can be achieved. Very

interesting is that the results for 2 nodes shows that a lot of time has been saved when switching from blocking I/O write to the non-blocking counterpart. While the absolute execution time of this scenario was a lot longer than all the other execution times of the different scenarios presented so far, it shows that if one has to work with a setup which runs I/O server and client application on the same nodes which provide only 1 CPU per node, the use of non-blocking I/O can nearly cut in half the execution time. Of course in this case it might be better to change the execution environment in order to reduce absolute I/O time but concerning the potential of non-blocking I/O operations it shows that its use can really help in both cases: In case a pretty good execution environment can be provided, but also in cases where this cannot be done.

Average ratio	**0.5783**
Minimum speedup	0.5936
Maximum speedup	0.5716
Average quality	0.05133
Minimum quality	0.08288
Maximum quality	0.03573
Ratio of calculation times	1.08561
Ratio of io times	1.1208

Average ratio	**0.7418**
Minimum speedup	0.7615
Maximum speedup	0.7211
Average quality	0.009700
Minimum quality	0.01604
Maximum quality	0.0003914
Ratio of calculation times	1.1294
Ratio of io times	1.9594

10 Future work

While the provided thoughts on non-blocking I/O operations and the implemented benchmark showed that non-blocking I/O operations not only could bring a performance boost but also that is actually does, this is only a first step in the examination of this version of doing I/O. A lot different aspects can be distinguished and more detailed examinations can be done. Some of these are the following.

- In the chapter on non-blocking I/O operations and their semantics different scenarios have been presented. While the scenario where I/O and calculation take the same amount of time when executed concurrently is the one which promised the best performance boosts, it is not the only one presented. Not all applications need to do more or less the same amount of data access and calculations but rather are either I/O or calculation intensive. As discussed in chapter 5 the latter scenarios could also save some time by using non-blocking I/O operations.
  This should be tested and verified. Also it could be analyzed at what ratio of I/O to calculation it does not help any more to use non-blocking operations.
- In order to discuss non-blocking I/O and to analyze performance gains made possible by it, only the non-collective write operation has been used. Other forms of doing I/O such as collective operations can be analyzed. The collective I/O operations defined in the MPI-IO standard are called split collective operations [2]. Also the read operations are a candidate for analyzation and could also promise performance gains when used in the non-blocking versions.
- Furthermore more detailed tests can be done using different kinds of access patterns. Also the different influences of amount of data to be accessed in each I/O operation on the performance gains can provide a series of interesting tests.
- Last but not least all scenarios and tests should be run on a bigger computer cluster using more nodes. While the presented tests do show that non-blocking I/O has the potential to save overall execution times it coud be very interesting how the benchmark behavior changes when scaled to greater numbers.

The benchmark code provided has been written in a way which should allow for easy adjustment and adding of new tests.
The command line parameters influence variables and buffers which can be interesting for a lot of different future tests. The actual test implemented has been written in a way which allows for new tests to be added which allows for a non-blocking I/O benchmark suite to emerge.

11 Conclusion

While optimizations and modernizations in computer systems are very important in the struggle to minimize execution times for applications run on parallel computers, it has been shown that they are not the only way to do so.
Non-blocking I/O operations and their semantics have been presented and the theory of possible performance gains by making use of them have been discussed. The theory has been tested and proved by the proposed benchmark and the results have been compared to the expectations established when looking at the theory.

While the overall setup of the execution environment and optimizations e.g. in the used parallel file system are very important to minimize I/O times and help to make sure that data access does not slow down applications by becoming a bottleneck, it is not always possible to switch to a better computer system. In order to still reduce the time an application needs to terminate, the programmer can make use of non-blocking I/O operations like the emulated `E_MPI_File_iwrite(...)` function proposed in this paper.
All scenarios tested showed a ratio of `non-blocking version / blocking version` lower than `0.75` and some scenarios even allowed for results close to the theoretical optimum. While for applications with small execution times and very fast I/O systems this might not be much time when looking at absolute numbers, it can be quite a bit for applications which take a long time and have to deal with an I/O system which does not provide a good data access bandwidth.

All in all one can say that not only administrators and developers of used software layers such as the parallel file system can help in reducing the application execution times but also the application programmers themselves can make better use of their available and valuable time on computer clusters by changing from blocking I/O operations to their non-blocking counterparts where possible.

Bibliography

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org`, May 1994.

[2] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org`, June 1997.

[3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[4] MPICH group of Math and ANL Computer Science Divison. MPICH2 home page. `http://www-unix.mcs.anl.gov/mpi/mpich/index.htm`.

[5] Hans Meuer (Univ. of Mannheim), Jack Dongarra (Univ. of Tennessee), Erich Strohmeier (NERSC/LBNL), and Horst Simon (NERSC/LBNL). TOP500.org. `http://top500.org`, April 2007.

[6] A. S. Hornby. *Oxford Advanced Learner's Dictionary of Current English*. Oxford University Press, 5 edition, 1995.

[7] Julian Martin Kunkel. Performance Analysis of the PVFS2 Persistency Layer. `http://pvs.informatik.uni-heidelberg.de/Theses/2006-kunkel-bsc.pdf`, 2006.

[8] Thomas Ludwig. Folien zur Vorlesung: Hochleistungs E/A Systeme. `http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0607/heas-0607.pdf`, 2006.

[9] Thomas Ludwig. Materialien zur Vorlesung: Cluster Computing. `http://pvs.informatik.uni-heidelberg.de/Teaching/CC-06/index.html`, 2006.

[10] Michael Krietemeyer, Daniel Versick, and Djamshid Tavangarian. THE PRIOmark PARALLEL I/O-BENCHMARK. In *International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, Feburary 2005.

[11] R. Rabenseifner. Invited Talk in the Lecture: "Hochleistungs-Eingabe/Ausgabe-Systeme, Effective File-I/O Bandwidth Benchmark (b_eff_io) and Other I/O Benchmarks. `http://www.hlrs.de/people/rabenseifner/publ/iwr_Jan2007_b_eff_io_slides.pdf`, 2007.

[12] R. Rabenseifner and Alice E. Koniges. Effective Communication and File-I/O Bandwidth Benchmarks. In *Proceedings, 8th European PVM/MPI Users' Group Meeting*, Santorini, Greece, September 2001.

[13] Rob Latham, Neill Miller, Robert Ross, and Phil Carns. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld Magazine*, pages 56–59.

[14] Frank Shorter. Design and Analysis of a Performance Evaluation Standard for Parallel File Systems. `ftp://ftp.parl.clemson.edu/pub/techreports/2003/PARL-2003-001.ps`, August 2003.

[15] PVFS2 Development Team. Parallel Virtual File System, Version 2. `http://www.pvfs.org/pvfs2-guide.html`, September 2003.

[16] William Gropp, Ewing Lusk, and Thomas Sterling. *Beowulf Cluster Computing with Linux.* The MIT Press, 2 edition, 2003.

[17] William Stallings. *Betriebssysteme.* Pearson Studium, 4 edition, 2003.