INAUGURAL - DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von
Diplom-Informatiker Hipolito Vásquez Lucas
aus: Ostuncalco, Guatemala, Mittelamerika

Tag der mündlichen Prüfung: 04.07.2011

Efficient Management of Huge Data Sets on Cluster Computers

Gutachter:
Prof. Dr. Thomas Ludwig
Prof. Dr. Ulrich Brüning

Abstract

In a cluster computer a parallel file system is encharged to spread one single parallel file on the different computer's I/O nodes using a determined distribution function. In file I/O intensive parallel scientific applications with *semi-random temporal parallel file I/O access patterns,* this file is accessed at different addresses at the same time by a number of processes that may vary between two consecutive iterations.
In this thesis a set of *semi-random temporal parallel file I/O access patterns* generated by a phylogenetical application is categorized. For these patterns a partitioning function is proposed that guarantees at any time during execution access to the parallel file.
This thesis shows the correlation existing between the type of I/O access patterns and the type and setting of two round robin based distribution functions so that the overall application's execution time can be reduced.

Auf einem Clusterrechner dient ein paralleles Dateisystem dazu, eine einzelne parallele Datei den verschiedenen E/A-Knoten des Rechners mittels einer bestimmten Verteilungsfunktion zuzuweisen.
In parallelen wissenschaftlichen Anwendungen mit intensiven *zeitlich semi-zufälligen parallelen E/A-Zugriffen* wird auf mehrere Addressen einer solchen parallelen Datei aus unterschiedlichen Prozessen gleichzeitig zugegriffen, wobei sich die Anzahl dieser Prozesse zwischen zwei nacheinander folgenden Iterationen ändern kann.
In dieser Dissertation wurde ein Satz von *zeitlich semi-zufälligen parallelen E/A-Zugriffen* kategorisiert, der von einer Stammbaumberechnungsanwendung erzeugt wird. Für diesen Satz von Zugriffen wurde eine Partitionierungsfunktion konzipiert, die der Anwendung jederzeit das Schreiben ihrer Daten in die parallele Datei unabhängig vom E/A-Zugriffstyp ermöglicht.
In dieser Dissertation wird die existierende Korrelation zwischen den E/A-Zugriffstypen und den Einstellungen zweier Round-Robin-basierter Verteilungsfunktionen gezeigt, unter denen die Anwendungsausführungszeit reduziert wird.

Dedication

I dedicate this PhD thesis to Susanne Löwe-Vasquez, the Vásquez Lucas family from San Juan Ostuncalco, Guatemala, Eileen Schwartz and her family from Portland, Oregon, Dr. Sandra Kurtinitis and her family from Washington D. C., and to all those who with their activities make a better world out of our planet.

Acknowledgement

I would like to thank every one who scientifically, technically, and motivationally helped me throughout the course of this PhD thesis.
Firstly, I would like to thank my advisor Prof. Dr. Thomas Ludwig for his scientifical and technical guidance at any point in time during the development of this thesis.
I thank Martin Julian Kunkel for the help provided concerning the cluster administration, programming debugging, and his valuable suggestions for this thesis.
I would also like to thank Sven Marnach for his help concerning cluster administration.
I am very thankful to Mrs. Catherine Proux-Wieland and Mrs. Karin Tenschert for the motivational talks throughout all years that I have been here at the Institut für Informatik from the University of Heidelberg. I also thank Mrs. Karin Tenschert for her help concerning this work's spelling corrections.
Last but not least, I am also thankful to Dr. Bertil Mächtle for the motivational talks after very long working days, weekends, and holidays, and for reading a version of this work.

Contents

1. Introduction

1.1 Motivation

Mainly due to the electromechanical dependent seek time and rotational latency of hard disk drives (HDDs), the performance of disks has evolved at a smaller rate than that corresponding to the performance of CPUs during the last years [MTHC$^+$08, PaCh98]. This difference of development at the hardware level is one of the main causes of the so-called I/O-bottleneck problem in disk-based computing systems. Several approaches have been undertaken to solve this problem. In the last years at the hardware level, the deployment of fast flash memory based solid state drives (SSDs) [PSSM$^+$10, TKRM08] presents a promising solution to this isssue.
Nevertheless, one single fast secondary storage device can also become the I/O bottleneck, if its throughput does not scale with the number of used CPUs. Such scenarios are typical for data intensive parallel scientific applications that frequently convey huge amount of data sets between primary and secondary storage on SMP- and multicore-based cluster computers[1]. Therefore, other methods to diminish the I/O-bottleneck drawbacks have been conceived. The parallel I/O concept is one of them and it consists in providing the computing system with more than one secondary storage device that can parallely be accessed from processes running on any of the used CPUs. The application of this concept does not only reduce access and delay times to one single value, but it also exploits the aggregated throughput and capacity of the used secondary storage devices set.
A cluster computer, in which at least two nodes have their own secondary storage device, inherently constitutes an appropriate hardware testbed for supporting parallel I/O. Nevertheless, in order for the file I/O intensive scientific applications to appropriately access this set of secondary storage devices, corresponding operations at the file system and middleware level must be provided. Furthermore, the applications' file accesses must be adapted to these operations.
At the file system level, a parallel file system is encharged of the physical distribution of files on top of the involved storage devices. It also manages the created files and their attributes. The middleware provides an appropriate interface and optimizations to use the parallel file system's functionalities.

[1]For this work I used an HDD-based testbed. Thus, except when explicitly stated, the descriptions are limited to this type of systems.

A parallel I/O intensive scientific application can access its data from many processes to one single parallel file. This file is spread across the involved secondary storage devices by the parallel file system through the usage of physical distribution functions. At different times throughout the whole execution time the application accesses the file at different addresses. Thus, generating temporal and spatial I/O patterns while accessing the file.
In this work I propose the thesis that one distribution function presents different degrees of efficiency when it is applied to service different applications' access patterns. Therefore, the execution time of an I/O intensive scientific application, accessing one single file with given access patterns, can be reduced by applying the adequate distribution functions or parameters to store the file across secondary storage devices.

1.2 Summary of Main Results

This section summarizes this work's contributions while using the approach that I describe in subsection 2.4.1 to answer the posed research question under section 2.3. These contributions support the thesis stated in section 1.1.

1. In this work I have identified and categorized in four types a set of *semi-random temporal parallel file I/O access patterns* that may appear during the execution of the p(MC)^3 algorithm on an abstract machine provided with a one dimensional array as parallel I/O interface. I have also proposed a mathematical expression to determine the total possible number of different patterns that might appear during the whole execution time based upon the number of I/O swaps that can be determined during the first iteration.

2. I have proposed the *mapping at first generation* partitioning function, in order to write into a parallel file data generated by any of the identified *semi-random temporal parallel file I/O access patterns.*
   The *mapping at first generation* partitioning function can be applied to applications in which the time parallel I/O access patterns randomly vary throughout the whole execution time, but whose data storage structure in a parallel file can be determined in a map during the first iteration. This map is a data structure containing all offsets and sizes of the parallel file. It is constructed based upon a predefined file format, one of each element sizes that the file will contain, and the number of entries for each one of these elements. At writing iteration each process with data to be written, using the map and the iteration number, computes the corresponding address(es) for its data in the parallel file. Finally, the write operation takes place.
   In this work I have implemented the *MFG* function for the p(MC)^3 algorithm. This is a write-only case that generates a set of *semi-random temporal parallel file I/O access patterns.* Since the *mapping at first generation* partitioning function provides each process with a complete mapping of the parallel file at any time, it is also suited for write-only applications with random parallel I/O access patterns.

3. The *mapping at first generation* partitioning function in combination with the simple_stripe distribution function supports each one of the categorized *semi-random temporal parallel file I/O access patterns* generated by an application based on the p(MC)^3 algorithm, whereas in combination with the varstrip distribution function it supports only the π_1 pattern (*see Sec. 7.4*). Independently

of the used distribution function to handle the parallel file, this pattern presents the highest speedup values among all recognized patterns on the I/O environment described in section 7.2. Furthermore, this pattern presents the highest of its speedup values while serving it with the varstrip distribution function. In order to further increase this speedup with this distribution function, I also proposed the *parallel line buffering* mechanism to control the number of computed results to be kept in main memory before writing them into the parallel file.

4. For π_1 patterns I identified a set of conditions to set the striping unit within the simple_stripe distribution function and to apply *parallel line buffering* to set the *varstrip chunk size* within the varstrip distribution function, in order to reduce execution time.

5. In this work I have shown that the speedup for a π_1 pattern varied in a semi-parabolic manner in terms of the MPI processes number. I called the number of processes with which the highest speedup was attained as the *process saturation number* (*see Sec. 7.6.2.1*). Except for the highest speedup value, this parabolic behavior means that the same speedup value was attained by choosing two different MPI process numbers to execute the serial and parallel I/O implementations of the program. Nevertheless, the same speedup translates into different execution times for the serial and parallel I/O implementations. A speedup value obtained with MPI process numbers smaller or equal than the *process saturation number* corresponded to the smaller of these two execution time values. Thus, I called it the *smallest-process-overhead speedup.*
   In this work I have experimentally determined the *process saturation number* to be equal to one MPI process per CPU. Furthermore, the speedup values also approached a maximum value towards the total used number of CPUs on an I/O cluster with 2 CPUs SMP architecture nodes.

6. The application's parallel I/O implementation that I propose in this work is suited for the computational phylogenetical analyses involving a big number of taxa and runs. While using the *mapping at first generation* in combination with the simple_stripe distribution function and its striping unit default value to compute an analysis with 256 runs an input matrix containing 2664 taxa, a speedup at *process saturation number* of 11.3 with an efficiency of 70% was measured. This speedup was computed on an I/O cluster of 8 nodes with 2 CPUs SMP architecture nodes and one secondary storage device per node[2].

1.3 Structure of the Thesis

This chapter states the thesis that I propose in this work. It also summarizes the results obtained through the course of this work that support the proposed thesis. This work involves basically two disciplines of human knowledge: parallel I/O and phylogenetical analysis. These are disciplines from computer sciences and biology, respectively. Chapter 2 provides the necessary terminology from these areas, in order to gain a precise understanding of this work's research question and the approach that I used to solve it. I thoroughly describe these under section 2.3 and 2.4, respectively.

[2]Since this is a π_1 pattern, this speedup can still be improved by applying the varstrip distribution function with the optimal value for the *varstrip chunk size.*

Chapter 3 describes two distribution functions implemented within the PVFS2 parallel file system. The first one is the varstrip distribution function, which we proposed to control I/O throughput and load balancing within the parallel file system. The second one is the simple_stripe distribution function. In order to show the performance of these distribution functions and their interactions with different parallel I/O patterns, in this work I have proposed a parallel I/O implementation for MrBayes, which is a program for the computation of phylogeny. Chapter 4 presents a description of the original program's serial file I/O activities. This is a very first approach to acquire some knowledge about the program's I/O requirements. As part of the parallel I/O implementation process, in chapter 5 I present a theoretical analysis of the p(MC)^3 algorithm (application's algorithm) in terms of its I/O requirements on an abstract computing machine under the assumption that this machine is provided with a one dimensional array to store its data. In section 5.2 I quantify and categorize the set of time parallel I/O access patterns that can appear in a semi-random manner during the algorithm's execution time.
In chapter 6 I propose the *mapping at first generation* partitioning function *(MFG)*. In a general case, the *mapping at first generation* partitioning function enables processes, involved in an algorithm with random time parallel I/O access patterns, to access their data into a parallel file at the corresponding addresses. In this particular case, I have implemented the *MFG* for the p(MC)^3, an algorithm with *semi-random temporal parallel file I/O access patterns* and write-only operations. Chapter 6 also presents the function's interactions with distribution functions within the parallel file system, especially how the *parallel line buffering* mechanism sets the varstrip distribution function. Chapter 7 describes the performance evaluation of the *MFG* function with different parameters at the application and system software level.
Finally, chapter 8 presents the conclusions and a list of possible future research topics that can be conducted based upon this thesis.

2. Background and Research Problem

The purpose of this chapter is to provide a detailed description of the research question (*see Sec. 2.3*) that I am addressing in this work. Before posing this question, the required vocabulary to understand its context is introduced. Due to the multi-disciplinarity of this work, it is necessary to define two types of terminologies: one in the area of parallel I/O (*see Sec. 2.1*) and the other in the area of phylogenetical analysis (*see Sec. 2.2*). Section 2.4 describes the approach that I used to tackle this research question and it also describes some related work.

2.1 The Parallel I/O Concept

A relatively abstract idea of a hardware computing system consists of a central processing unit (CPU) and a memory. Concerning the system's performance, the user should theoretically expect an infinite fast processing unit, an infinite space to store data, and an infinite fast access time for the CPU to access the corresponding data. Unfortunately, due to physical constraints actual computing systems do not provide these requirements. In the memory's case an actual computing system does not have only one type of memory, but rather it has a set of different types of memory that are ordered in a hierarchical manner according to their parameter ranges. The access time, the data troughput, and the storage capacity are three of these parameters. The access time across two of these types may vary to one or more orders of magnitude. In many computing systems hard drive disks constitute the slowest devices of their memory hierarchy. When a CPU requires to access data stored on such devices for a given computation, the overall system's performance to conduct this task suffers because the parameters' ranges of such devices are greater than those corresponding to registers, cache and main memory. This difference between CPU and I/O devices yields to the so-called I/O-bottleneck problem [HsSm04] in disk-based computing systems.
In past years the storage capacity of hard disks has increased, but their throughput has not correspondingly kept pace with this development [Leve10, MoTr03]. This increment of storage capacity makes disk-based systems attractive for applications that deal with huge amounts of data in the area of high performance parallel computing [FoCo94, CuSG99, DoSe98, Dong05]. Nevertheless, solutions are required to

overcome or ameliorate the corresponding I/O-bottleneck of such systems. One of these solutions is the concept of parallel I/O [May01, RCCK$^+$95, WoGr97]. The basic idea behind this concept is to provide the computing system with more than one secondary storage device and arrange them in such a manner that their throughput and storage capacity values arithmetically add up to one single aggregated throughput and one single storage capacity value.

This parallel I/O concept applies to distributed memory systems such as cluster computers and it has as central concept the so-called logical file. The logical file is an abstraction for representing the total data accessed by a computing program in a one dimensional array. On the one hand this data is divided among the so-called compute nodes in order to process their corresponding part of the problem. On the other hand, this one dimensional array must be divided among many I/O nodes, which store it in their corresponding secondary storage devices. Spreading this one dimensional logical file across different I/O nodes is called physical file layout or striping [Croc89, ChPa90], whereas the division among the compute nodes is called partitioning [Quin03]. The striping concept has been applied at the different levels within a computing system. In the case of RAID level 0 systems [LeKa93], this concept is applied at the hardware level in special disk controllers, whereas other applications are virtualizations, such as it is the case of software RAIDs [Cort99].

In a cluster computer the parallel I/O consists in conveying the global data structure that is partitioned among compute nodes to a file spread across I/O nodes. At the lowest level a parallel I/O operation can be viewed as a mapping between compute node memories and disk addresses [NiLo97].

The following sections describe the computing system environment that is required to implement the parallel I/O concept on a cluster computer. This includes the hardware, the file system, the middleware, and a parallel I/O application.

2.1.1 I/O Cluster

Even though the processor speed and the hard disk capacity storage of monoprocessor systems (*see Def. A.1*) continue to increase, there are certain computing problems for which these resources are insufficient. Usually the characteristic of such a computing problem is not its difficulty to be computed, but to be computed with a small execution time in order to keep up with the design cycle of a product being developed or to meet real time requirements [Volk99]. Furthermore, such a computing problem might generate an amount of data which exceeds the storage capacity provided by the monoprocessor system. This limitation might lead to a partial solution. In order to provide at some degree suited hardware for these computing problems, cluster computers are deployed. These are collections of monoprocessor systems that communicate and cooperate to quickly solve a large problem [AlGo89]. A cluster computer is a distributed memory computer in which information exchange occurs using messages. The strengths of a cluster computer is its memory scalability in terms of the number of nodes and the fast access of one node to its local memory. Nevertheless, a domain decomposition function and a load balancing function (if required) must be programmed to divide the computing task among the cluster's nodes. For the purposes of this work I consider an *I/O cluster computer as a special type of cluster computer, in which each node has exactly one physical secondary storage device.* In this work an I/O cluster provides the basic parallel I/O infrastructure that must be made available to the application through corresponding mechanisms at the parallel file system and I/O library level.

2.1.2 Parallel File Systems

Applying the parallel I/O concept to an I/O cluster offers applications the possibility to use the secondary storage devices as a single entity. In order for an application to exploit the strengths of such storage entity, a corresponding software architecture is required. This basically is a layered architecture, which consists of a parallel file system and a parallel I/O interface. This and the next subsection describe these two layers, respectively.
In the context of a parallel file system the physical nodes of a cluster computer are divided into three different sets of nodes: the compute, the I/O, and the metadata nodes. The compute nodes are deployed for the actual program's computation. The I/O nodes store the corresponding data. Thus, they must have at least one physical secondary storage device. The metadata nodes administer the file system's metadata set.
A parallel file system is the software encharged for the management of parallel files. This management consists of two parts. The first one consists in providing a logical view of a file to processes that parallely access it. The second one consists in storing or retrieving the file to or from more than one physical secondary storage device.
The basic mechanisms implemented in parallel file systems are: Buffering and distribution functions. The idea behind buffering is to minimize the access to secondary storage as much as possible by keeping data in buffers in main memory. Buffering requires the corresponding implementation of mechanisms to decide the appropriate time to access secondary storage. In applications in which accesses occur in small sizes some parallel file system keep them in buffers on the compute nodes, I/O nodes, or both and proccess them at once after certain criteria are met. In [IMOS$^+$04] the researchers describe the implementation of two such mechanisms, collective I/O and cooperative caching, in the clusterfile parallel file system. Keeping data on different buffers on different nodes requires the implementation of mechanisms such as locking or consistency protocols, in order to guarantee data consistency.
The distribution function is the mechanism used to parallelly store or retrieve one single logical file over many secondary storage devices. A simple distribution function can be implemented by using the round robin mechanism. This mechanism is also deployed for RAID 0 systems.
The provided aggregated bandwidth and the augmented maximum file size in parallel file systems are two characteristics that can be exploited by file I/O intensive scientific applications that generate huge amounts of data.
Some of the parallel file systems are: VESTA [CoFe96], PPFS [JERC$^+$01], GPFS [ScHa02], Clusterfile [IsTi03].

2.1.2.1 The PVFS2 Parallel File System

This section describes the PVFS2 parallel file system, the open source second version of the Parallel Virtual File System (PVFS) [LiRo01, IIRo99, LiRo96]. This is a small description of the implemented functionalities and mechanisms, architecture, provided interfaces, and configurations. It is based upon the documentation provided under `doc` and the source code of the pvfs-2.8.1. We provide in [KuLV04] a German description of the parallel file system that includes a set of diagrams, with which we thoroughly depict the parallel file system configuration, the server and client architecture in a layered model as well as in terms of state machines. This last set of diagrams is of special interest from the programming point of view.

PVFS2 has a client-server architecture. It has one type of client and two types of servers, the `io` and the `metadata` servers.
Client and servers have a layered architecture, from which the `Job manager`, `Flows`, and the `Buffered Message Interface (BMI)` are common to both. In the context of PVFS2 an operation consists of jobs. The `Job manager` handles the development of such jobs. The `Flows` layer is encharged of data movement among network, storage devices, or memory. The `BMI` abstracts the network. Additionaly to the above mentioned layers the client includes the `System Interface`. This interface provides the set of PVFS2 functionalities that can be accessed from a parallel I/O library such as ROMIO, the kernel driver, or the PVFS library. Many of the operations available at the system interface are implemented through corresponding state machines.
Besides the common layers the servers have also the `Trove`, `Op State Machine`, and the `Request Handler` layers. `Trove` abstracts the storage devices, whereas the other two layers take care of the operation machines and requests, correspondingly.
As in other parallel file systems, in the context of PVFS2 a cluster computer is divided in three different set of nodes: the compute, I/O, and the metadata nodes.

2.1.2.2 Distribution Functions

Distribution function is a method describing the mapping from logical files, logical one dimensional sequence of bytes, to a physical layout of bytes on PVFS2 I/O nodes. In PVFS2 several distribution functions have been implemented. If none of these is explicitly selected by the user, the parallel file system applies the simple_stripe distribution function with a default striping unit of 64KB.
The simple_stripe distribution function is a mechanism that divides the one dimensional logical file in a set of non-overlapping chunks of data, which are called strips. These strips are then stored in a round robin manner on datafiles on the I/O nodes [LMRC04].
In the PVFS2 terminology a strip is the amount of data written to a single server, whereas a stripe is the total number of strips written into the I/O servers in a round robin cycle. Chapter 3 describes in a more extended manner the distribution functions that I used in this work.

2.1.3 Interfaces

One way to use parallel file systems is directly from a parallel program. Such programs can be implemented by using different programming models and corresponding libraries such as PVM [GeSc02, Chapter 11], OpenMP [CDKM$^+$01] or the message passing interface (MPI) library.
In the case of MPI-based parallel programs the ROMIO library [ThGL99b], implemented according to the MPI-2 standard, is an alternative to access the parallel I/O subsytem. This subsection briefly describes the parallel I/O functions specified in the MPI-2 standard and the ROMIO library implementation.
The MPI-2 standard specifies functions for supporting asynchronous I/O, strided accesses, and control over the striping mechanisms. In the context of the MPI-2 standard an MPI File is considered as a set of ordered etypes (elementary types). These etypes can be MPI predefined or derived types. Writing or reading data occurs in etypes units. Figure 2.1 illustrates these elements. A view is defined as the data, which a process can access in an opened file. It is defined by the tuple `(displacement, etype, filetype)`. The offset is defined as a position in terms of etypes and within a given view. In the MPI-2 two types of file pointers are distinguished: shared and individuals. Shared file pointers have a shared value among

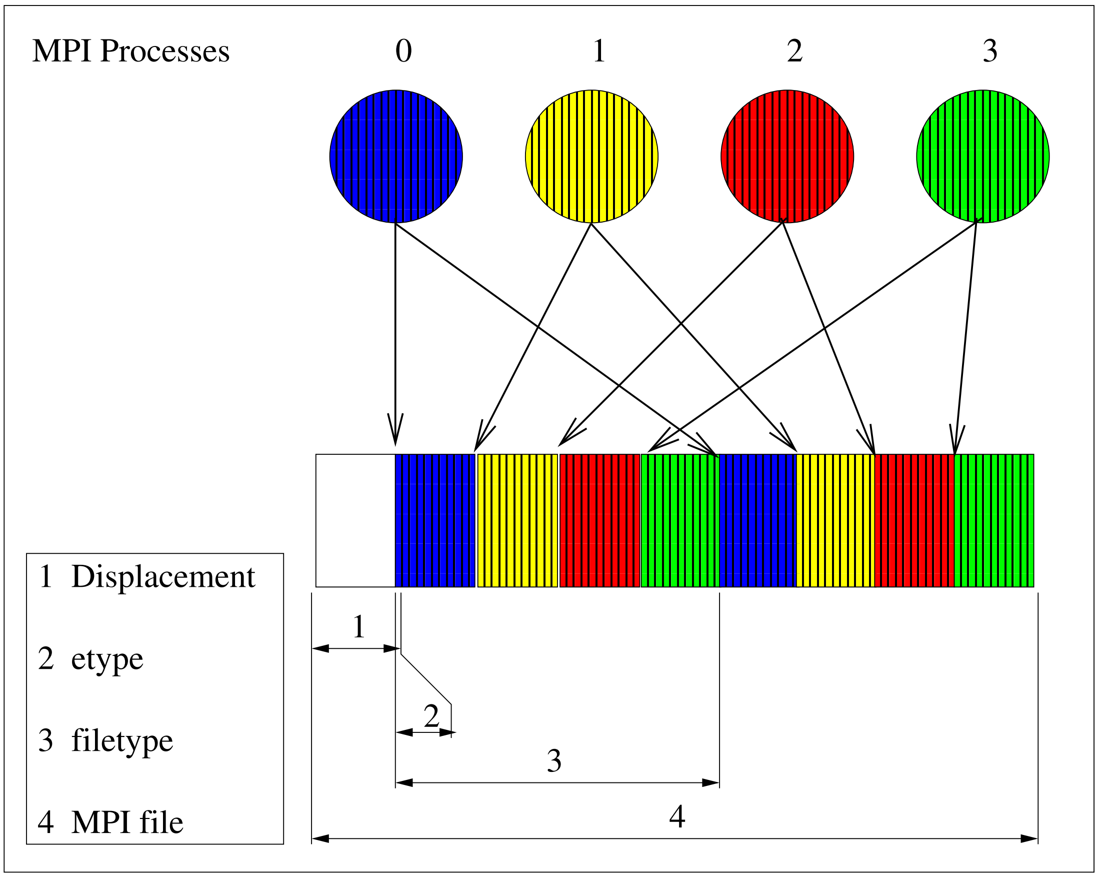

Figure 2.1: MPI file elements according to the MPI-2 standard (4 MPI processess)

the processes, which open a file. The value of individual file pointers are local to the processes.
Once an MPI file is open, a corresponding file handle is returned and this is used for the operations on the file. Besides operations for file manipulation such as `open, close, delete, resize, preallocation, query`, and `file info`, the MPI-2 standard specifies operations for manipulating file views, data accessing, file interoperability, consistency and semantics, and error handling. Detailed descriptions can be found in [GrLT99, MPI 08].
The following paragraphs to describe the ROMIO libray is strongly based on the description found in [WLRR03, Chapter 19]. ROMIO includes basically two optimizations: the two phase I/O [ThCh95] and data sieving [ThGL99a]. The two phase I/O or collective buffering is an optimization for collective I/O operations and it differs for write and read operations. For a write operation the first phase consists in distributing the data to a set of processes called the aggregators. In the second phase the aggregators write the data to the file system. In the case of read operations the aggregators acquire their regions from secondary storage and distribute them to the processes involved in the collective read operation. Six MPI hints from the type described in subsection 2.1.3.1 can be used to control the two-phase I/O: `cb_buffer_size, cb_nodes, romio_cb_read, romio_cb_write, romio_no_indep_rw, cb_config_list`. Data sieving consists in accessing a set of noncontiguous regions of a file by reading a single block, which includes all regions including the data between them. Finally, the client extracts from this single block its required data. The following hints can be used for data sieving in a stack consisting

of ROMIO and PVFS: `ind_rd_buffer_size, romio_ds_read`. The following hints control List I/O in PVFS: `romio_pvfs_listio_read`, `romio_pvfs_listio_write`.

2.1.3.1 MPI-Hints

In the context of the MPI-2 standard the `info` object [MPI 08, Chapter 9] is a tuple `(<key>, <value>)`. A key can have only one value and both elements are case sensitive strings. This object can be given as argument for MPI functions. The MPI-2 standard specifies a set of operations to `create, set, delete, query,` and `free` info objects.
In the context of MPI-2 standard the info object can be used to specify hints that provide information on optimizations. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. Hints are specified on a per-file basis in `MPI_FILE_OPEN`, `MPI_FILE_DELETE`, `MPI_FILE_SET_VIEW`, and `MPI_FILE_SET_INFO`. The reserved file hints mainly affect access patterns and the layout of data on parallel I/O devices [MPI 08, Chapter 10].

2.1.4 Applications

This subsection completes the description of the whole hardware and software stack in a parallel I/O environment. It provides a brief description of the file I/O intensive scientific applications' characteristics.

2.1.4.1 A File I/O Intensive Application

Since there are different areas of science and engineering, in which parallel file I/O intensive applications can be found, this section describes in a general manner the characteristics of such applications, instead of describing a particular one.
For the purposes of this description I simplify such an application as consisting of a number of iterations. In these iterations computing and access to the I/O subsystem can take place. In theory the application can access the I/O subsystem in each iteration for compulsory data, checkpointing, data staging [MiKa91], and other kinds of data or it can access the I/O subsystem only during the last iteration due to compulsory data. This means that the same scientific application represents different workload to the I/O subsystem. Based upon this fact, I propose in this work in expression 2.1 the definition of an *X percent file I/O intensive scientific application* .

Definition 2.1 *A scientific application is X percent file I/O intensive, if it computes or simulates phenomena in the areas of natural sciences or engineering and satisfies:*

1. *$X = \frac{T_{Max_{io}}}{T_{exec}} * 100$, T_{exec} = Total execution time, $T_{Max_{io}}$ = Maximum possible total I/O time. The time $T_{Max_{io}}$ refers to the time used in accessing the I/O subsystem.*

2. *Its I/O software capacity requirements at the file system level exceeds those provided by sequential file systems.*

2.2 Phylogenetical Analysis

In order to understand the application's context, this section introduces a small terminology in the area of phylogenetical analysis.

2.2.1 Phylogeny

A binary tree T_{bin} is a connected graph [CoLR90] without cycles, which can be expressed as $T_{bin}(N, E)$. N and E respectively are sets of nodes and edges. An $n \in N$ is called leaf or tip if it is connected by exactly one $e \in E$, else n is called internal node. A rooted binary tree T_{rooted} is a binary tree, in which except for the root node each internal node has three edges. The root node has two edges. An unrooted binary tree $T_{unrooted}$ has no root node.
A phylogenetic tree [DHJL$^+$97] or phylogeny T_{phyl} is a binary tree $T_{phyl}(N, E, D)$, in which D is a set of parameters such as distances or evolution parameters that are assigned to the edges $e \in E$. In the context of systematic biology phylogenetic trees show the course of evolution in a group of organisms [Fels83].

2.2.2 Computational Phylogenetical Analysis

The purpose of a phylogenetical analysis is to determine the relationship of familiarity among biological species, populations, individuals or genes from a given set of sequences of proteins, deoxyrebonucleic acids (DNA) or rebonucleic acids (RNA)[Lesk03][1].
The actual relationship of familiarity among a group of biological organisms is represented in a phylogeny $T_{phyl_{actual}}$. Its internal nodes represent the ancestors of each biological entity that are represented at the leaves. The edges represent the relationship of descendency or ancestry among the entities. A distance $d \in D$ between two nodes represents time or distance of mutational changes between the corresponding sequences. An unrooted phylogeny only provides information about the degree of familiarity among the organisms, whereas a rooted phylogenetic tree additionaly shows their common ancestors.
Ideally the ultimate goal of computational phylogenetical analysis is to determine $T_{phyl_{actual}}$ corresponding to a given set of sequences. If the number of sequences is small the determination of $T_{phyl_{actual}}$ from this set of different phylogenetical candidates $T_{phyl_{candidate_j}}, j \geq 2$, is a relatively easy task. As the number of sequences increases computing the value of $T_{phyl_{actual}}$ becomes complex and depending on the constraints, such as permitted execution time, used algorithm, and computing resources, an approximated phylogeny $T_{phyl_{final}}$ may be computed instead of the actual $T_{phyl_{actual}}$. Equation 2.1 and 2.2 show the total number of possible phylogenies $T_{phyl_{unrooted}}$ and $T_{phyl_{rooted}}$ as a function of n number of sequences to be compared as described in [HuBo01]. These equations correspond to the unrooted and rooted phylogenies, respectively.

$$T_{phyl_{unrooted}}(n) = \frac{(2n-5)!}{2^{n-3}(n-3)!} \tag{2.1}$$

$$T_{phyl_{rooted}}(n) = \frac{(2n-3)!}{2^{n-2}(n-2)!} \tag{2.2}$$

[1]Except when explicitly stated, the term sequence in this work refers to any of these three types.

Depending on the utilized method to determine $T_{phyl_{actual}}$ and the required accuracy a computer program may store some or each computed phylogeny in secondary storage. Equations 2.3 and 2.4 express the theoretically maximum values of the required secondary storage for phylogenies of γ bytes.

$$SS_{T_{unrooted}}(n) = \gamma * T_{unrooted}(n) \tag{2.3}$$

$$SS_{T_{rooted}}(n) = \gamma * T_{rooted}(n) \tag{2.4}$$

As an example, an analysis of 20 organisms where each computed phylogeny, rooted and unrooted, has a γ of 1 byte would maximally require the following space on secondary storage: $SS_{T_{unrooted}}(20) = 220$ Exabytes (10^{18}) and $SS_{T_{rooted}}(20) = 8,2$ Zettabytes (10^{21}).
A computer program determines the actual phylogeny $T_{phyl_{actual}}$ for a given set of sequences using a method of phylogeny's inference. This method works on the provided set of sequences producing as output $T_{phyl_{final}}$, the best approximation for $T_{phyl_{actual}}$. Some of the used methods provide only point estimates of the phylogeny, so assesing confidence methods such as bootstraping [HoLe03] must be applied to determine how strongly the data support each of the relationship presented in the phylogeny. The following section describes probabilistic methods for the inference of phylogenies.

2.2.3 Probabilistic Methods

The probabilistic methods work with a-priori or a-posteriori probabilities on the candidates trees $T_{phyl_{candidate_j}}, j \geq 2$, in order to determine the appropriate phylogeny $T_{phyl_{final}}$. These methods are based on the Bayes's theorem.
The Bayes's theorem can be mathematically expressed for two independent events M and T as indicated by equation 2.5. The term $P(T|M)$ is the probability of T to happen provided that the evidence M is known. This expression is also called the a-posteriori probability of T because it depends on M. $P(M|T)$ is known as the likelihood function. The other terms of equation 2.5 are a-priori probabilities of T and M.

$$P(T|M) = \frac{P(T)P(M|T)}{P(M)} \tag{2.5}$$

In the area of phylogenetical analysis equation 2.5 determines the probability of a tree T, with topology, and a set of distances and evolution parameters, given the known matrix of aligned DNA sequences M. Equation 2.6 presents a more detailed version of equation 2.5 that is used for phylogenetical inference.

2.2.3.1 The Maximum Likelihood Method

The Maximum Likelihood method identifies as the optimal phylogeny $T_{phyl_{final}}$ the one which presents the maximum value for the likelihood function $P(M|T)$ on the space of all $T_{phyl_{candidate_j}}, j \geq 2$ trees. A thorough description can be found in [DEKM98].

2.2.4 Bayesian Inference of Phylogeny

The determination of phylogeny $T_{phyl_{final}}$ through the Bayesian method basically consists of two parts. The first part is the computation of the posterior probability for each tree from the set of possible trees depending on the number of sequences as expressed by equations 2.1 and 2.2. The second part consists in summarizing the results from the computed posterior probability distribution of trees to determine phylogeny $T_{phyl_{final}}$. The Maximum a-posteriori probability or MAP method, for example, considers the tree having the highest posterior probability as $T_{phyl_{final}}$ [YaRa97]. The posterior probability of the ith phylogenetic tree τ_i can be calculated using expression 2.6 and 2.7 such as indicated in [HuBo01].

$$f(\tau_i|\mathbf{M}) = \frac{f(\mathbf{M}|\tau_i)f(\tau_i)}{\sum_{j=1}^{T(n)} f(\mathbf{M}|\tau_j)f(\tau_j)} \tag{2.6}$$

$$f(\mathbf{M}|\tau_i) = \int_\lambda \int_\theta f(M|\tau_i, \lambda, \theta)f(\lambda, \theta)d\lambda d\theta \tag{2.7}$$

In equation 2.6 and 2.7 $\mathbf{M}$ represents a matrix with the aligned DNA sequences. The total number of possible trees $T(n)$ can be either T_{rooted} or $T_{unrooted}$ as stated in equations 2.1 and 2.2. The branch lengths and substitution parameters are represented by λ and θ respectively. The a-priori probability of a phylogeny τ_i is usually $\frac{1}{T(n)}$ and the prior on branch lengths and substitution parameters is expressed as $f(\lambda, \theta)$. The likelihood function $f(M|\tau_i, \lambda, \theta)$ is typically calculated under the assumption that substitutions occur according to a time-homogeneous Poisson process [HuRo01].

2.2.5 The Markov Chain Monte Carlo, MCMC, Method

Because the integration and addition required to compute expression 2.6 are complex to perform analytically even for small phylogenetic problems [HoLe03], the Markov chain Monte Carlo (MCMC) method has been used [HRNB01, YaRa97], to approximate the posterior probability distribution. This approach consists in initiating a Markov chain with a tree T^c. The state space of the chain corresponds to the tree's characteristics such as topology, length, and evolution models. By stochastically perturbating the current tree T^c a new tree T^n is proposed. The acceptance probability R of this new tree is given by equation 2.8 as stated in [HLMR02].

$$R = min[1, \frac{f(M|T^n)}{f(M|T^c)} * \frac{f(T^n)}{f(T^c)} * \frac{f(T^c|T^n)}{f(T^n|T^c)}] \tag{2.8}$$

From left to right the three ratios of expression 2.8 are called likelihood, prior, and proposal ratio respectively. The terms of the proposal ratio are the probabilities of proposing the current or new chain. After the acceptance probability has been determined, a uniform random variable is drawn from interval $(0, 1)$. If this number is less than R then T^n becomes T^c. Otherwise the chain remains in the old state. This process of perturbing, accepting or rejecting new states is repeated many times.

2.2.6 The Metropolis coupled MCMC, (MC)^3, Method

This subsection presents the Metropolis coupled Markov Chain Monte Carlo (MC)^3 method. The next subsection describes its parallel version, the p(MC)^3 method.
Since the parallel I/O patterns that I categorize in this work are conditioned by the p(MC)^3 algorithm, the descriptions in these subsections are strongly based on the original description of these algorithms found in [ADHR04, HLMR02].
Instead of using one single Markov chain, the Metropolis coupled Markov chain Monte Carlo runs n chains. If for a chain the likelihood and prior ratio are raised

to $0 < \beta < 1$, the heat value, the chain is called heated, else it is a cold chain. Raising these terms to β increases the acceptance probability R. In the context of the so-called incremental heating the heat value of each chain $Ch_j, j = 1...n$ is given by equation 2.9.

$$\beta_j = \frac{1}{[1 + \Delta T * (j-1)]}, \Delta T > 1 \tag{2.9}$$

The term ΔT in equation 2.9 is called the temperature increment parameter and it determines the acceptance of the swaps. Equation 2.9 indicates that for n chains, the first one is a cold chain.
The iteration process in the (MC)^3 method works similarly to the one described for the MCMC method in subsection 2.2.5 except that after a given number of iterations two randomly chosen Ch_j and Ch_k are selected to exchange states. The swap has a probability of acceptance given in equation 2.10.

$$R_s = min[1, \frac{f(T_k|M)^{\beta_j} * f(T_j|M)^{\beta_k}}{f(T_j|M)^{\beta_j} * f(T_k|M)^{\beta_k})}] \tag{2.10}$$

As for the acceptance in equation 2.8 for each chain, R_s will be compared with a uniformly random number in the interval $(0, 1)$. If the random generated number is less than R_s then T_j and T_k exchange states. This process will be repeated many times and the frequency of states sampled by the cold chain will be taken as an approximation of the posterior distribution.

2.2.7 The Parallel (MC)^3 Algorithm

As in the case of the former subsection, this subsection is strongly based on the description found in [ADHR04]. The parallel (MC)^3 or p(MC)^3 distributes Markov chains among processes. This means that a swap may correspond to communication between processes. In order to minimize the amount of data being interchanged between two processes during a swap, the processes exchange their heat instead of their states. In order to guarantee that the result of the parallel implementation is equivalent to the serial implementation, the two chains which are involved in one swap must be in the same generation. This means that a synchronization must be made between two chains that are involved in a swap. In [ADHR04] two types of synchronization schemes have been proposed: the global exchange and the point-to-point swap. In the message passing implementaion of the global exchange all chains synchronize by calling the function `MPI_Barrier()` before sending and receiving the swap acceptance information and executing heat swaps between two randomly chosen chains Ch_j,Ch_k. The point-to-point variant in the message passing programming model is accomplished by using the function `MP_SendRecv()` between two chains involved in the swap and there is no need to synchronize with the rest of chains.
Figure 2.2 shows an example of the p(MC)^3 algorithm in the space of trees for a run with three heated chains and one cold. In this particular case two chains are chosen in a random manner to exchange states after five iterations.

2.2.8 The Open Source Program MrBayes

MrBayes [HuRo01, RoHu03] is an open source program for the computation of phylogeny that uses Bayesian inference principles. In the MPI version of the program the posterior probability distribution of a phylogenetical tree is approximated by using the parallel Metropolis coupled Markov chain Monte Carlo algorithm as described in subsections 2.2.5 - 2.2.7.

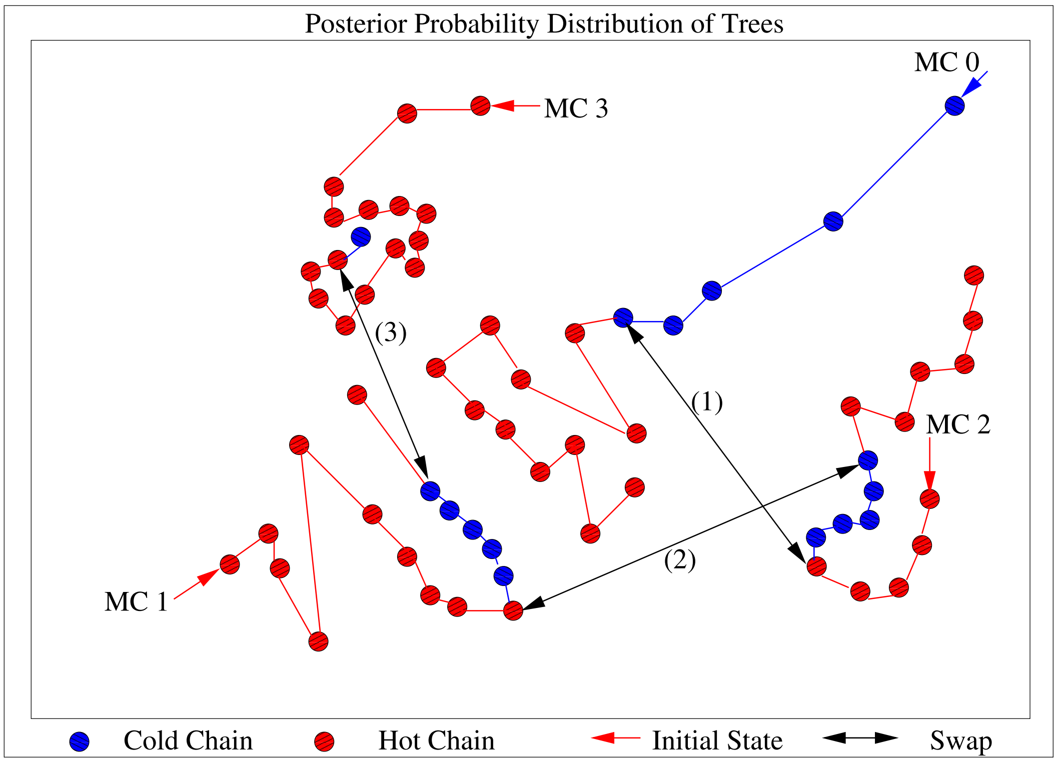

Figure 2.2: Example of possible paths of one cold chain with 3 corresponding heated chains in the posterior probability distribution of trees.

2.3 Research Question

The general research question of this work is how the usage of different physical distribution functions and corresponding parameters within a parallel file system can affect the execution time of different parallel I/O access patterns of a file I/O intensive scientific application running on an I/O cluster computer. In order to provide answers to this research question, in this work I implemented a parallel I/O version of MrBayes, a program to compute phylogeny that generates a set of *semi-random temporal parallel file I/O access patterns* during execution time. Using the PVFS2 parallel file system, this parallel I/O implementation writes its data into parallel files, whose physical distributions on the different I/O nodes can be chosen by the user through MPI hints at the application level.

2.4 Approach and Related Work

2.4.1 Approach

The approach that I used to answer the research question posed under section 2.3 consisted of the following steps:

1. Analysis of the I/O activities in the application's serial I/O implementation.
2. Qualitative and quantitative analyses of the spatial and temporal I/O access patterns generated by the MPI processes involved in the p(MC)^3 algorithm under the assumption that this algorithm accesses a parallel I/O interface. In this work I call them *semi-random temporal parallel file I/O access patterns.*

3. Proposition of the *mapping at first generation* partitioning function. This function supports the access to the one logical parallel file from the identified *semi-random temporal parallel file I/O access patterns* generated by the MPI processes involved in the p(MC)^3 algorithm. The function's name expresses the fact that in order to guarantee file access, each one of the involved process computes during the first iteration (generation, in the application's terminology) a map of the parallel file structure.

4. Interaction of the *MFG* function with the `simple_stripe` and `varstrip` physical distribution function.

The approach's first step was the characterization of the I/O properties used in the serial I/O application's implementation. This characterization consisted in determining the dataflow, the number and type of input and output files as well as their corresponding contents. During this stage I identified the *online files* as those files, in which the application writes while executing the p(MC)^3 algorithm. I experimentally determined the relationship among the amount of data written into the online files in each iteration and after the whole execution time. Analytically, I proposed a mathematical expression to compute the total amount of data written into the online files[2]. During the characterization stage I observed that the MPI process with `rank 0` is responsible for writing the data into n online files. The value of n was twice the number of runs. Thus, before the actual write operations into the files occur, one *many-to-one* communication pattern is generated. A many-to-one communication pattern appears when a group of MPI processes, the ones that have data to be written into the secondary storage, send their data to the MPI process with `rank 0`.
The second step consisted in a theoretical analysis of the *temporal parallel I/O access patterns* generated by the MPI processes involved in the p(MC)^3 algorithm under the assumption that they access a parallel I/O interface, a one dimensional array that can be accessed at the same time by more than one process. Under these conditions one write operation from the application's perspective corresponds to one temporal parallel I/O pattern. In this work I defined one temporal parallel I/O pattern as the assignment of cold chains to MPI processes and the processes' accesses to the I/O subsystem within a given iteration. Due to the random nature of the swaps between two Markov chains in the p(MC)^3 algorithm after a certain number of generations, the patterns generated during the first iteration (except pattern π_1) do not remain constant during the whole execution time. As part of this analysis I proposed a mathematical expression for the computation of the total possible different number of temporal parallel I/O access patterns that may appear during the p(MC)^3 algorithm's whole execution time. Furthermore, I also proposed a classification of these temporal parallel I/O patterns in four categories.
The next step was to propose the *mapping at first generation* partitioning function. As a partitioning function, the *MFG* is encharged to assign spaces of the parallel file to the processes that write into this file at the same time. One such assignment corresponds to one parallel I/O pattern. Thus, I conceived the *MFG* function to handle all possible patterns of the four above mentioned categories that may appear during the whole p(MC)^3 algorithm's execution.
In order to determine the influence of different distribution functions and their parameters' settings on the execution time, I determined the interaction of the *MFG*

[2]In this work I only took into consideration files with `.p` and `.t` extensions.

partitioning function with the simple_stripe distribution function as well as with the varstrip distribution function. In order to select an appropriate *varstrip chunk size* for this last distribution function, I also implemented the *parallel line buffering* mechanism.

2.4.2 Related Work

This thesis belongs to the category of research projects that have been conducted to gain a better understanding of the relationship existing between the scientific parallel applications' file input and output related characteristics and the mechanisms used to store these files in parallel file systems on multiprocessor systems.
One work on this research line was conducted in [Pura96]. In this work the researcher found that write operation into writes-only files were a dominant part of the used workload. The author proposed a write caching mechanism for write-only files based upon disk directed I/O and implemented it on a multiprocessor simulator.
In [Madh97] the researcher proposed two methods to detect qualitative file access patterns and correspondingly serve these access patterns by applying the appropriate caching and prefetching policies within the file system. In order to classify the access patterns the author proposed one method based on neural networks for short time scales and another on hidden Markov models for long time scales like file reuse.
In [Simi00] the researcher proposed a method based on fuzzy logic principles to set the striping unit provided the system's state. The researcher proposed to use the striping unit value based upon the request rate, the request size, the network, and disk speed. The parameters were expressed as fuzzy variables with corresponding fuzzy sets. These two projects were implemented as extensions to the Portable Parallel File System [JERC+01].
In [BCST+08] the researchers proposed a mechanism for the MPI-IO library to set prefetching based upon different so-called I/O signatures that they assign to access patterns, in order to classify them. These signatures are obtained from previous runs and are used by a thread to set prefetching during execution time.
In most of the projects mentioned above the researchers used a set of applications to test the performance of the proposed concepts within the parallel file system or the parallel I/O library.
Several projects, which take into consideration data set sizes, have been conducted to improve the performance of computational phylogenetical inference. In [FeCB06] the researchers proposed a parallel implementation based on the MCMC method to compute phylogeny. Their approach basically consisted in optimizing the likelihood evaluation and to propose a mapping of the input matrix and the number of chains among a matrix of CPUs for the computation. In [StAl10] the researchers proposed a compression algorithm that exploits the gap in the input original matrix. They proposed to use a reduced matrix for likelihood- or Bayesian-based methods, in order to compute phylogeny. For a minimum gapiness in the input matrix of 27% with an implementation for a likelihood method, half of the file I/O sizes measured and presented by the researchers were in the order of GB.
Both approaches have not taken into consideration parallel I/O neither for the input nor the output whatsoever. Therefore this work's contribution can be complementary to those approaches. Concerning solely a mean for this application to write data into a parallel file, we have proposed other approaches. One of them was implemented in [Zeer05]. This approach was based on the idea of defining one fixed set of I/O processes that was encharged for the application's write operations.
Instead of also using a set of applications as the other mentioned approaches, in this

work I concentrated on only one single application which generates a set of different semi-random parallel I/O access patterns during execution time. I proposed the *mapping at first generation* partitioning function as an application's extension to store its data into one single parallel file independently of the semi-random access pattern generated in the writing iteration. In order to further reduce the application's execution time, in this work I also proposed a set of conditions, with which the users can select the type of physical distribution function and parameters via MPI hints.

2.5 Summary

In this chapter I have described the work's main research questions as well as the corresponding approach to answer these questions. This description was possible due to the concepts in the area of parallel I/O and phylogenetical analysis which were introduced in the sections at the beginning of this chapter. At the end of this chapter I describe similar projects that have been conducted in order to tackle similar research questions.

3. Flexible Distribution Functions in PVFS2

The main purpose of this chapter is to describe the varstrip distribution function. Section 3.1 describes a set of spatial parallel file I/O access patterns at the application level that I surveyed of projects that have been conducted on multi-processor systems. One of these patterns, pattern 1 (*see Fig. 3.1*), served as reference for proposing the above mentioned distribution function. In subsection 3.2.2 I theoretically analyze the I/O performance of the simple_stripe distribution function to serve pattern 1. Furthermore, this subsection presents the mechanism behind the varstrip distribution function to serve this pattern. Subsection 3.2.2.2 describes the corresponding distribution's usage. In this subsection I also suggest a set of I/O parameters at the different levels of a parallel I/O environment that the users should take into consideration while choosing distribution functions.

3.1 Reference File I/O Access Patterns

The main objective of this subsection is to present a set of spatial parallel file I/O access patterns at the application level, which we have taken into consideration in order to propose the varstrip distribution function [LuLu05].
Parallel scientific applications store and retrieve to and from secondary storage the values of the variables that they compute. I represent such a spatial parallel I/O access at the application level as the mathematical relation 3.1.

$$P_{appl}(\mathbf{P}, \mathbf{A}, \mathbf{G}) \tag{3.1}$$

In expression 3.1 **P** represents a set of processes that require I/O accesses, **A** a set of addresses within the parallel logical file, and **G** a *set of global factors* . Each address $a_j \in \mathbf{A}$ has a corresponding offset Off_j and operation Op_j. For the purposes of this work, I assume a maximum possible cardinality for both **P** and **A** equal to the number of bytes within the parallel logical file. Expression 3.2 shows some elements of **G**.

$$\mathbf{G} = \{\tau, \alpha, \pi, \epsilon, \sigma, ...\} \tag{3.2}$$

A spatial parallel file I/O access pattern P_{appl} strongly depends on the time τ within the whole application's execution time. The total set of different spatial parallel file

I/O access patterns P_{appl} create the application's temporal parallel file I/O access pattern. Such a pattern might present a periodicity or it might be random. Chapter 5 presents a more detailed description of time patterns, especially of the random ones that are generated from the P(MC)^3 algorithm. As expression 3.2 shows, such a pattern is also conditioned by the application's nature α, the purpose of I/O π, the programming environment ϵ, and the system's I/O interface σ.
The application's nature α strongly defines the behavior of each computing process during execution time. A random nature, for example, might translate also into random I/O accesses. The user's purpose to access I/O π strongly influence spatial parallel file I/O access patterns. Basically, the user accesses the I/O subsystem for compulsory data, checkpointing, or data staging [MiKa91]. Compulsory data constitutes the data that must be processed as part of the computation, whereas checkpointing is done for recovery purposes. Data staging must be done when the data exceeds the size of main memory.
Some application patterns are conditioned by the used programming environment ϵ. Such as it is the case when using the `BLOCK, CYCLIC` combinations in High Performance Fortran [KLSS$^+$94]. The system's I/O interface σ influences spatial parallel file I/O access patterns basically through the view that they offer of the logical file and through the functions that they provide to manipulate this logical file. The logical file is presented to the applications as a one dimensional, such as in the case of MPI-IO, or as a one multidimensional array. For the purposes of this work I have summarized a set of spatial parallel file I/O access patterns and depicted it in figure 3.1. This summary is based upon the analyses of parallel I/O access patterns described in [MaRe97, ThGL98, RKPH04] that have been conducted on multi-processor systems. In pattern 0 each process accesses a corresponding file. In a strict manner this pattern does not belong to the parallel I/O set since there is no one parallel logical file. Nevertheless, it is important to take this pattern into consideration for comparison purposes. The rest of the patterns can be classified in terms of their degree of sequentiality as proposed in [MaRe04]. Pattern 1 is called global partitioned sequential. In this pattern the processes access different disjoint areas of the file in a sequential manner. Pattern 2 is a special case of pattern 1, in which only a portion of the entire file is accessed. This pattern appears at initialization time, for example. Pattern 3 as well as pattern 4 are global interleaved sequential. In pattern 3 each process has a sequential local view of the accessed data, but the data is strided in equal access regions within the file. Pattern 4 is an irregular global interleaved sequential pattern. In this pattern each process has a local sequential view of the data, but the data is strided in the file and each stride has different sizes. For the purposes of this work, I assume that each process does the same type of operation, read, write, sync or seek at a given point in time. This can be done through non-collective or collective I/O functions in the context of the MPI-IO interface.

3.2 Access Patterns and Distribution Functions

3.2.1 Variable Simple_stripe

The simple_stripe distribution function divides the logical file in chunks called strips and spreads these strips across the different I/O nodes involved in an I/O operation in a round robin manner. In the PVFS2 parallel file system if the users do not explicitly specify a distribution function and corresponding settings, this distribution function is used with a striping unit of 64 KB. This mechanism is shown in figure 3.2.

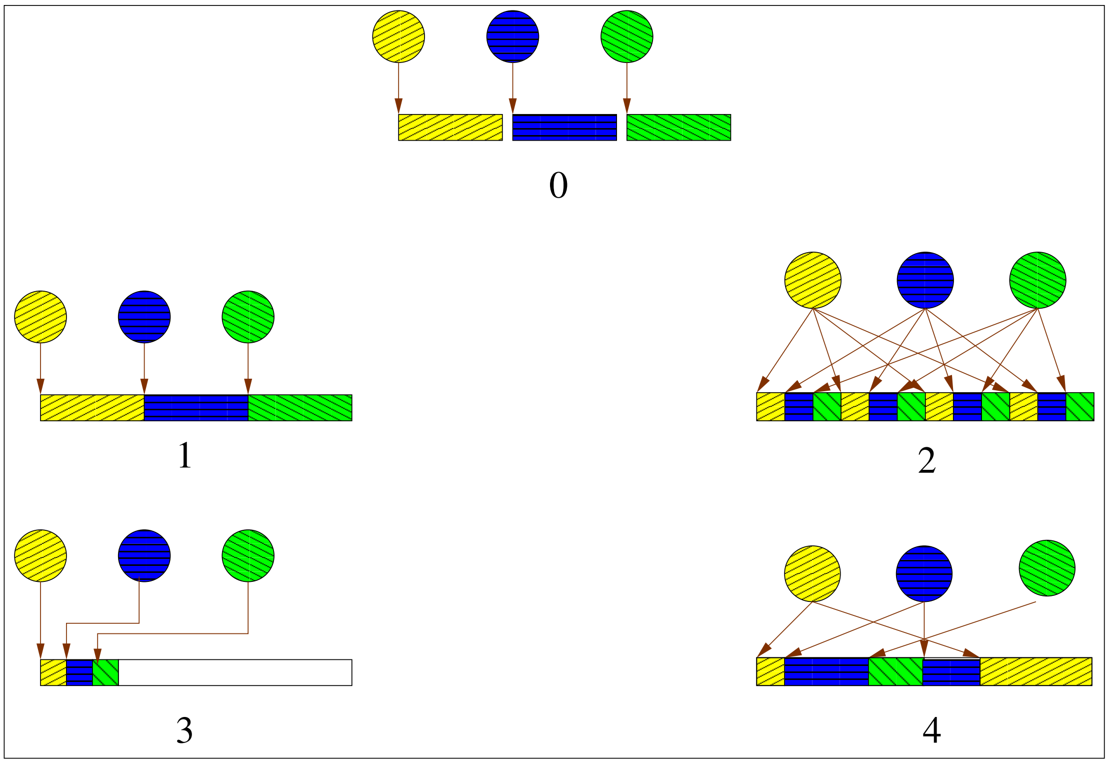

Figure 3.1: A set of spatial parallel I/O file access patterns at the application level

A flexible variant of this function offers the possibility to set the striping unit to a value different of the default one.

3.2.2 The Varstrip Distribution Function

Spreading the data on I/O nodes by applying only the simple_stripe distribution function results in throughput and load balancing performance drawbacks for some access patterns at the application level as I discuss in this subsection. An example of such patterns presents the pattern 1 generated on an I/O cluster. In figure 3.3 the simple_stripe and a configuration of the varstrip distribution functions are used to serve such a pattern[1]. From the algorithmical point of view there is no difference between the time complexities of these distribution functions to serve an I/O operation for an amount of data $data_{io}$ that equals one stripe in the context of the simple_stripe distribution function. Both have a time complexity of $O(Data_{io}/SF)$ with SF representing the number of chosen I/O nodes[2]. Nevertheless, in the case that this pattern runs on an I/O cluster in which each node has been configured as compute and I/O node at the same time and the application accesses an amount of data $data_{io}$ greater than one stripe, applying the simple_stripe distribution function induces network overhead by unnecessarily sending strips over the network to serve this I/O operation. Figure 3.3 depicts a hypothetical case of a pattern 1 running on the above mentioned configuration and writing into a parallel file that is divided in five strips by the simple_stripe distribution function. In such a scenario three strips use the network stack and reduce the total throughput performance in comparison to the varstrip function, in which no strips are sent over the network. From the load

[1]For this discussion I assume that the simple_stripe distribution function stripes the data always initiating on the same I/O node.

[2]In the ROMIO terminology SF is known as the striping factor.

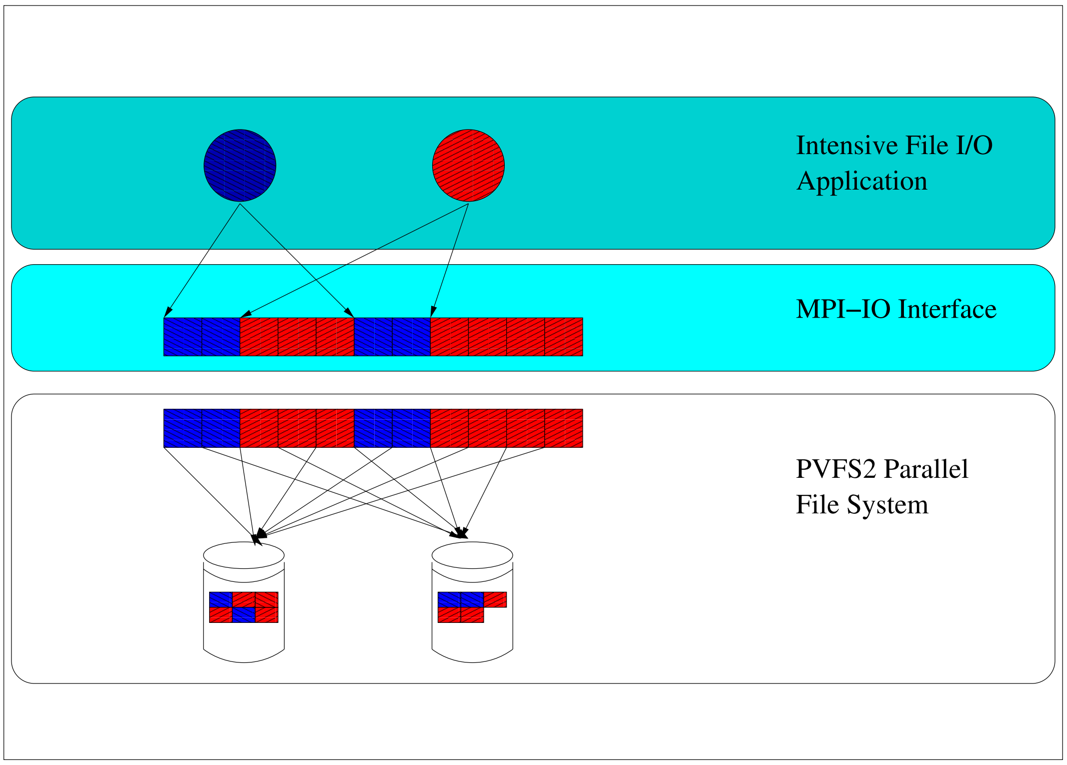

Figure 3.2: The Simple_stripe distribution function in PVFS2

balance perspective these strips are purposelessly sent over the network, since the load imbalanced partition translates also into a load imbalanced physical distribution of the data on the I/O nodes. Furthermore, in the case of a load imbalanced application, the load imbalance on the I/O nodes remains across different executions of the same application. In the case depicted in figure 3.3 each time the application is executed, the leftmost I/O node diminishes its available free storage capacity by one strip compared to the rightmost node. Thus, only the leftmost node's free available storage capacity limits the amount of data to be stored on the I/O nodes.

3.2.2.1 Objective

In order to overcome the drawbacks discussed under subsection 3.2.2 we proposed the varstrip distribution function [LuLu05]. The main idea behind this function is to provide, via MPI hints at the application level, a mean for *data placement control* on the I/O nodes. This data placement control in a general sense means that MPI processes storing data into one parallel file can choose data sizes as well as the I/O nodes, in which these data sizes must be stored. The strength of the *data placement control* is to tune the data storage in the parallel file for high throughput or for load balancing purposes. For the case depicted in figure 3.3 each process stores its data size, *varstrip chunk size* in this work, on its corresponding local I/O node. In this case the I/O nodes have *varstrip chunk size* values of 2 and 3 storing units. Since no data is sent over the network the I/O throughput increments compared to the throughput obtained while applying the simple_stripe distribution function.
As in the case of the simple_stripe distribution function, executing this application n times induces a load imbalance of n storing units. Nevertheless, due to the *data placement control* provided by the varstrip distribution function this load imbalance can be reduced to a value of zero or one storing unit for even or odd values of n,

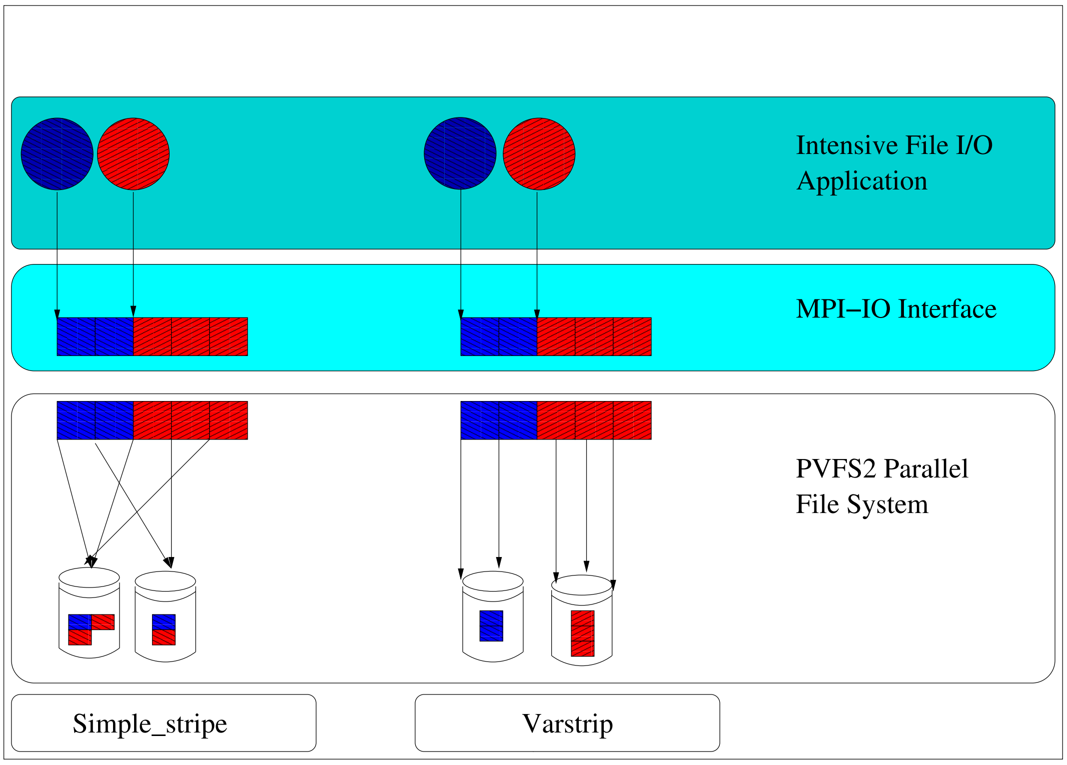

Figure 3.3: Pattern 1, the Simple_stripe and the Varstrip distribution functions in PVFS2 on an I/O cluster.

respectively. This is possible by executing the application $n/2$ times with the shown configuration and the rest by exchanging the I/O nodes between the MPI processes. In this last configuration the storage of each storing unit involves the network stack and the throughput decreases. This trade off between throughput and load balance must be considered by the user before applying the varstrip distribution function. In this work I concentrate only on throughput issues.

3.2.2.2 Usage

The *data placement control* provided by the varstrip distribution function enables users to select the *varstrip chunk size* values to be stored on the different used I/O nodes. In order to provide such functionality to an MPI based parallel program at the application level, distribution hints[LuLu05] can be deployed. These are MPI-IO hints for the selection of distribution functions and their corresponding parameters. The following snippet illustrates how to set the varstrip distribution function by a program with two MPI processes. Process 0 writes 200KB on I/O node 1 and process 1 writes 4MB on node 2[3]:

```
MPI_Info_set(fileInfo, "varstrip_dist:string:strips","0:200000;1:4000000");
```

The combination of the *data placement control* provided by the varstrip distribution function and the configuration of compute nodes as I/O nodes at the same time can translate into throughput or load balance performance gain.
Besides the relationship between compute and I/O nodes, in table 3.1 I propose a

[3]For simplicity I am considering 1MB = 1000000 bytes.

set of factors that users should take into consideration when choosing a given distribution function and corresponding parameter values. I selected part of these factors while conducting this work's evaluation. These factors are presented in appendix A.2 through an entity relationship diagram model and corresponding fields' explanations in table A.7. In [EGVL05] we present a set of measurements that show the

Applications
Type of access pattern
Partition's load balance
Blocking or non-blocking operations
Compute Nodes
Processes per compute node
PIOlibrary optimization
I/O Nodes
Interconnect between compute nodes and I/O nodes: network, no network.
Physical Storages
Existing load imbalance
Storage capacity heterogeneity

Table 3.1: Factors to take into consideration while choosing distribution functions.

performance of both distribution functions to serve a set of synthetic I/O accesses. These measurements experimentally support the argumentation in subsections 3.2.2 and 3.2.2.1.

3.3 Distribution Function Metrics

In order to be able to compare performances of distribution functions, a set of metrics is required. This section describes those that I use in the rest of this work. I use the term *physical degree of I/O parallelism* and define it in equation 3.3, where $N_{io_{acc}}$ represents the number of I/O nodes accessed during one I/O operation and N_{io_t} the total number of available I/O nodes.

$$\delta_{\pi_{iox}} = \frac{N_{io_{acc}}}{N_{io_t}} * 100 \tag{3.3}$$

I also use the term *depth of I/O parallelism* and represent it by $\delta_{\pi_{ioz}}$ to define the number of stripes required for one I/O operation. In round robin based distribution functions the value of these metrics depend on the data provided to the distribution function and the selected striping unit to store it. A striping unit bigger than the data to be written translates into access to one single I/O node and presents the minimum value of the *physical degree of I/O parallelism* for such a configuration.

3.4 Summary

In this chapter I have described *data placement control* and how the varstrip distribution function applies it to control data throughput and load balancing, to trade off these parameters against each other, and to decouple an imbalanced partition from an imbalanced data physical distribution. A variant of the simple_stripe distribution function, in which it is possible to choose the striping size was also described.
In this chapter I have presented a set of file I/O characteristics at the application level, which served as reference for proposing the varstrip distribution function. Furthermore, I listed a set of I/O parameters that users should take into consideration while selecting distribution functions.

4. Application's Characterization

Before proposing parallel I/O alternatives for the application, which would minimize the execution time of the p(MC)^3 algorithm, it was important to understand the I/O accesses in the application's original serial I/O version. This chapter presents the experimental and analytical steps that I conducted to characterize the application's serial I/O requirements.
Section 4.1 describes the different data flows with corresponding files and contents which occur before, during, and after the algorithm's execution. In section 4.2.1 I analytically and experimentally describe the file I/O activities during execution time. This description includes the amount of data, I/O patterns, and the application's percentage of file I/O intensity according to definition 2.1.

4.1 Analyzing the Data Flow

The first step of the application's I/O characterization consisted in determining the data flow generated during an `mcmc` analysis. Figure 4.1 shows a summary of this data flow with a granularity at the file level. In order to determine this data flow, I used an input matrix with 12 taxa and 898 characters (detailed information on the matrix can be found under appendix A.3.2) and ran an `mcmc` analysis with four chains 1000 number of generations [1]. The computed information in each generation was written into secondary storage.
Figure 4.1 shows a set of Petri-Netze with four elements: states, transitions, tokens, and edges. The states correspond to files or main memory. Files are represented in the figure through their suffixes. The transitions represent commands. The data and its direction flow are represented through tokens and edges, respectively. For sake of clarity, I depicted the edges joining commands and input states to represent the fact that a command can be called any number of times.
During this application's characterization, I identified three stages for a phylogenetical analysis. For the purposes of this work I simply called them *I*, *II*, and *III*. Figure 4.1 shows the data flow in each one of these stages. During stage *I* the input file containing the matrix to be analyzed was read into main memory. In stage *II*

[1] In the MrBayes terminology an iteration is called a generation. Thus, in this work I indistinctly used both terms.

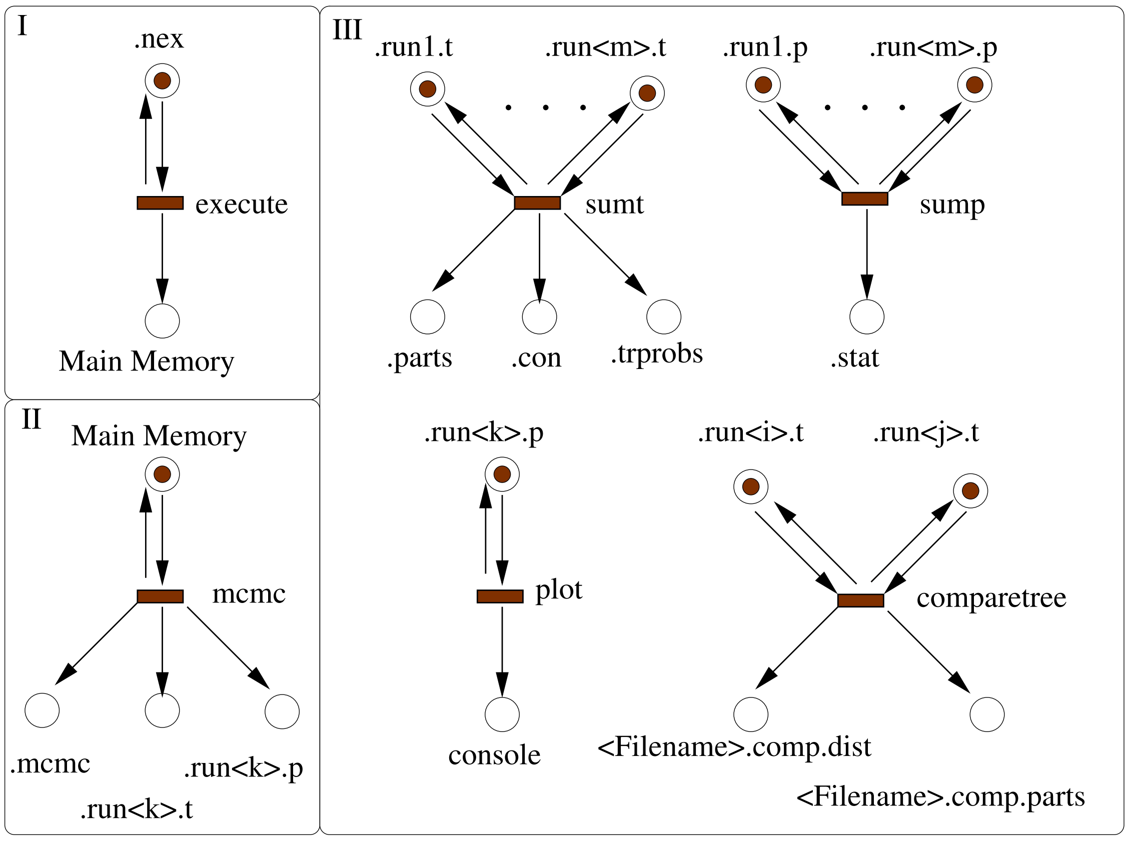

Figure 4.1: Data flow in MrBayes 3.1. Representation with a granularity at the file level.

the p(MC)[3] algorithm was executed and three types of files were generated: `<Filename>.mcmc`, `<Filename>.p`, and `<Filename>.t`, which in this work I call *mcmc*, *p*, and *t* file, respectively. Since these files were accessed during the algorithm's execution, I also called them *online files*. The total number of *p* and *t* files were defined by the chosen number of runs. Each run was stored in one *p* and one *t* file. Stage *III* corresponded to the postprocessing. As figure 4.1 shows, different types of files, which I called the *postprocessing files*, were generated during this stage using only the content of *p* and *t* files as input.

4.2 *The Online Files*

According to the above analysis only the manner in which the *online files* are accessed may alter the amount data generated by the application and its execution time. Thus, I concentrated on further analysis of these files. I especially concentrated on the analysis of the *p* and *t* files, which were needed by the postprocessing commands and contained the actual information concernig the phylogenies.

4.2.1 Amount of Data

Subsection 4.2.1.1 describes a set of experiments to determine the relative amount of data generated and written into the *online files* by selecting different iteration numbers. Furthermore, in subsection 4.2.1.2 I propose an expression to determine the total amount of data generated during one `mcmc` analysis.

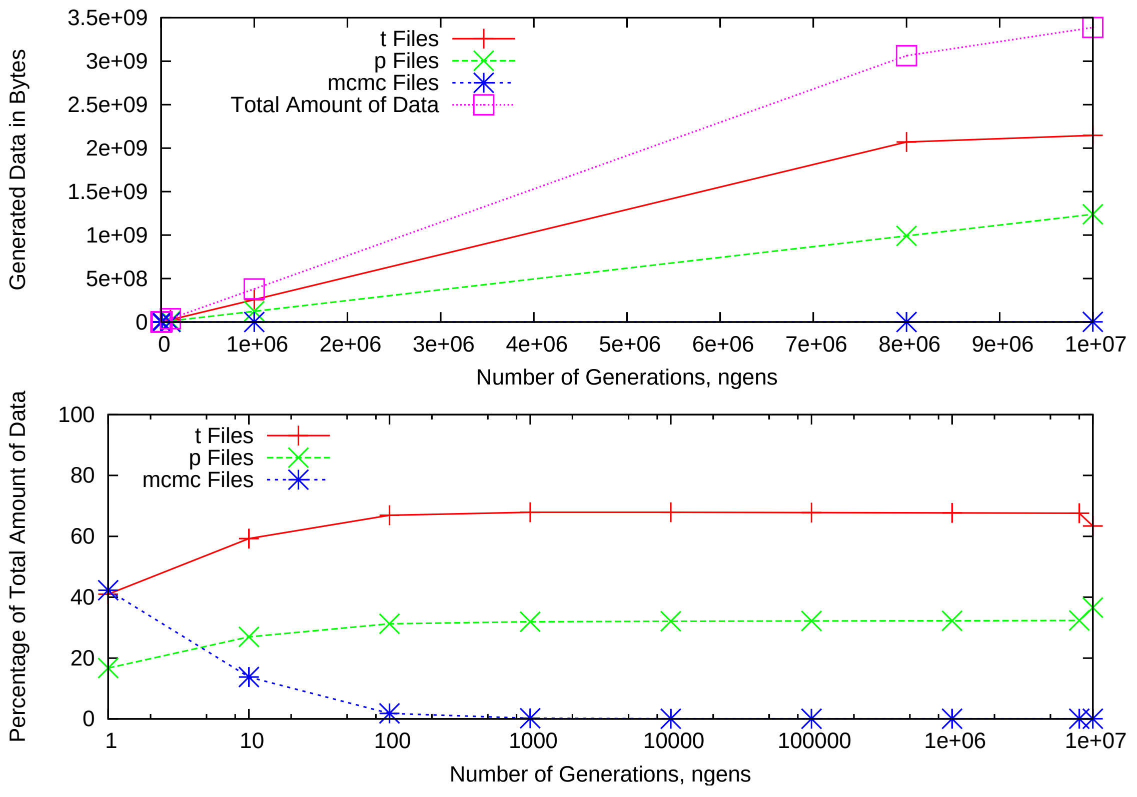

Figure 4.2: Amount of data generated during an `mcmc` analysis and its distribution among *online files*. These values correspond to an input matrix with 12 taxa of 898 characters.

4.2.1.1 Experimental

In order to experimentally determine the amount of data generated during `mcmc` analyses[2] and in which proportion this data was distributed among the three *online files*, I ran a set of `mcmc` analyses using the input matrix with 12 taxa and 898 characters.
I ran each analysis with four MPI processes. The difference among the analyses was the value of `ngen`, which I varied from 1 to 10 millions in steps of one order of magnitude. Except for the number of generations that I changed by each `mcmc` analysis, I explicitly set and used the values of the following `mcmc` parameters:

```
nruns=1             allchains=Yes         burninfrac=0.25   stoprule=No
ngen=<1,10,...>     temp=0.2              allcomps=Yes      Savebrlens=Yes
samplefreq=1        startingtree=Random   mcmcdiagn=Yes     Printall=No
nchains=4           relburnin=Yes         diagnfreq=1000    Printfreq=100
```

I present the measured values in figure 4.2. These measurements show that the maximum contribution of the *mcmc* file to the total amount of written data was 42% with `ngen`$= 1$ and dropped to values under 1.8% for `ngen`> 100. Under these conditions most of the data was written into the *p* and *t* files. In analyses with `ngen` > 100 the amount of data written into the *t* file was twice as much as the data written into the *p* file. In the case of the analysis with one million generations

[2]The MrBayes command `mcmc` starts the execution of the phylogenetical analysis.

the total amount of data was 0.35 GB. When I ran the `mcmc` analysis with ten million generations diagnostic information and the statistical parameter concerning the last generation were written into the *mcmc* and *p* files respectively, but the *t* file only contained the representation of 8295731 trees. With these number of tree representations the size of the *t* file was 2147483647 bytes or 2 GB, which corresponded to the maximum size set in the used `ext3` filesystem. This fact is clearly depicted in the upper half of figure 4.2. The fact that the last tree representations were not written could have altered the execution time after generation 8295731, therefore I also conducted an `mcmc` analysis with 8 million generations. After 8 million generations the size of the *t* file was of 1.92 GB and the total amount of generated data of 2,85 GB. Using the measured values I calculated the total amount of data for the 10 million generations. This calculated value was 3.5 GB.

4.2.1.2 Analytical

Before proposing a parallel I/O implementation for the *p* and *t* files, it is important to analytically determine the amount of data written at the end of each iteration, the total amount of data generated during an experiment, and the parameters on which these quantities depend.
In order to propose an expression for the total amount of generated data after an experiment, I abstract the contents of the *t* and *p* files as follows: I consider a *t* file as a set of *t lines*, $tt_i, i > 0$, with a file header tt_h. The file header provides information on the file's format, the identity of the analysis, from which the lines tt_i are generated, and the taxa. Each of the tt_i is a topological representation of a phylogenetic tree that is computed during the `mcmc` analysis.
I similarly consider a *p* file as a set of *p lines*, $tp_i, i > 0$, with a file header tp_h. Each tp_i contains parameter related information on the corresponding tt_i line of the *t* file. The header provides information on the semantical meaning of the components of the lines tp_i.
The complete information on a phylogenetical tree and its parameter values after the completion of a sampled iteration i is represented by tt_i and tp_i in the *t* and *p* file, respectively. Thus, I propose equation 4.1 to express the total amount of written data by `nruns` runs into the *t* and *p* files after `ngen` iterations with sample frequency `samplefreq`.

$$D_{total} = nruns(t_h + p_h) + \sum_{j=1}^{nruns} (\sum_{i=1}^{FileSize(ngen,samplefreq)} (tt + tp)_i)_j \tag{4.1}$$

Equation 4.1 shows the different parameters which influence the amount of generated data. The variable $FileSize$ (depending on `ngen` and `samplefreq`) represents the file size of one single file, either *t* or *p* file, expressed in number of lines. The size of each tp_i varies depending on the chosen parameters of the model of DNA substitution. The size of tt_i depends mainly on the number of taxa being studied. The execution of `nruns` > 1 increases the precision of the results computed within a given time frame. These results can be used, for example, to determine the degree of convergence such as conducted in [HiHJ05]. The total number of tt_i and tp_i within a given run to be written into secondary storage can be varied according to the chosen `ngen` and how often this are sampled, `samplefreq`. For a given number of iterations `ngen` and runs, the maximum amount of data is generated with `samplefreq` $= 1$ and the least amount with `samplefreq` $=$ `ngen`.
The generated data in each writing iteration can be computed by using equation 4.1 and the total number of these iterations.

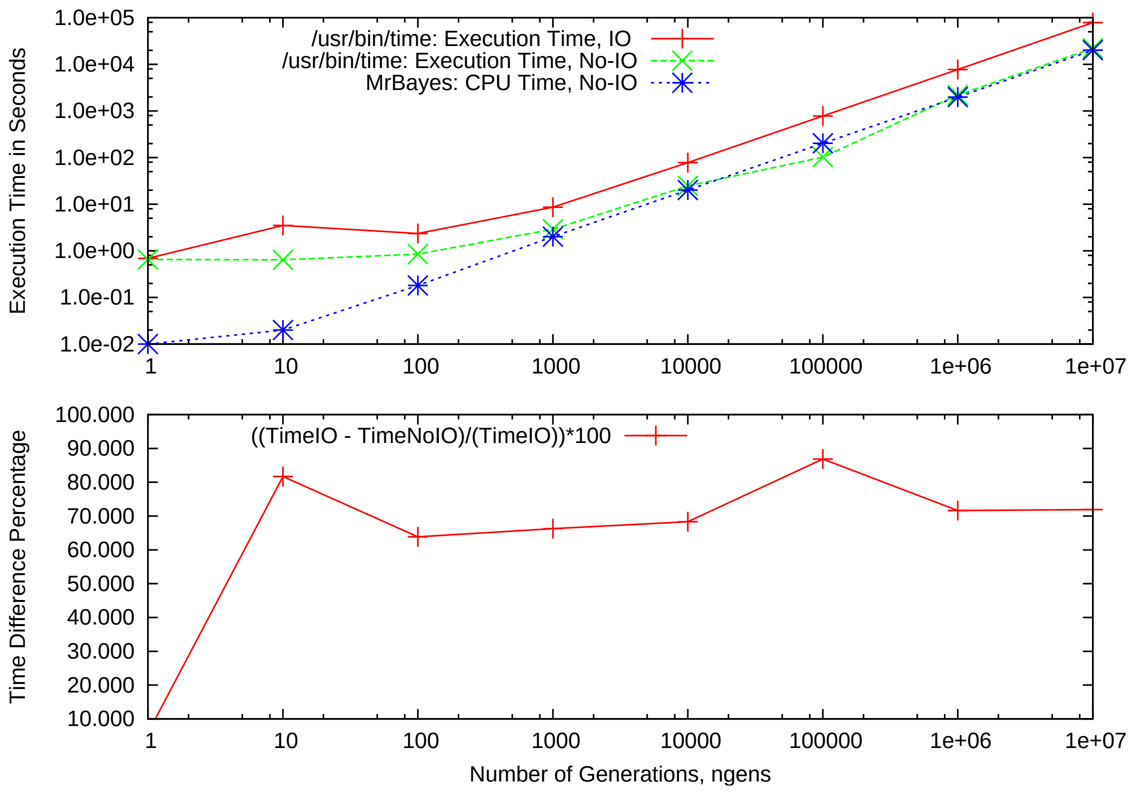

Figure 4.3: Execution Time with I/O, no I/O, and corresponding time difference percentage

4.2.2 Percentage of File I/O Intensity

This section describes experiments that I conducted to determine the different types of time frames which compose an iteration. These are basically the CPU, communication, and file I/O time. In order to have a preliminar idea of the application's (in)dependency on file I/O activities, the performance of the *No-I/O* code is also presented. This is a code variant that I modified so that no data is written into the *online files* during `mcmc` analyses.

The application's percentage of file I/O intensity, according to definition 2.1, is strongly conditioned by the parameters of equation 4.1 such as the number of runs and the number of iterations to be written into the *online files*. Furthermore, it is also conditioned by the input matrix size, the number of MPI processes such as thoroughly described in chapter 7. Thus, the objective of the experiment described in this section was to determine whether or not there was at least one application's setup that would benefit from a parallel I/O implementation of the *online files*.

I conducted these experiments using 4 MPI processes under the same conditions as in subsection 4.2.1. Figure 4.3 and 4.4 depict the measured values. The total execution time and the required CPU time were measured with `/usr/bin/time` and the built-in MrBayes timing. Figure 4.4 shows a correlation between the measurements made with `/usr/bin/time` and those provided by MrBayes. The total execution times were 17.5 and 21.9 hours for 8 and 10 million generations, respectively. The maximum percentage of CPU time was reached between these two intervals and was of 26%. This means that 74% of the time was used for communication and file I/O operations.

Figure 4.3 shows in the upper half the execution times measured for both codes vari-

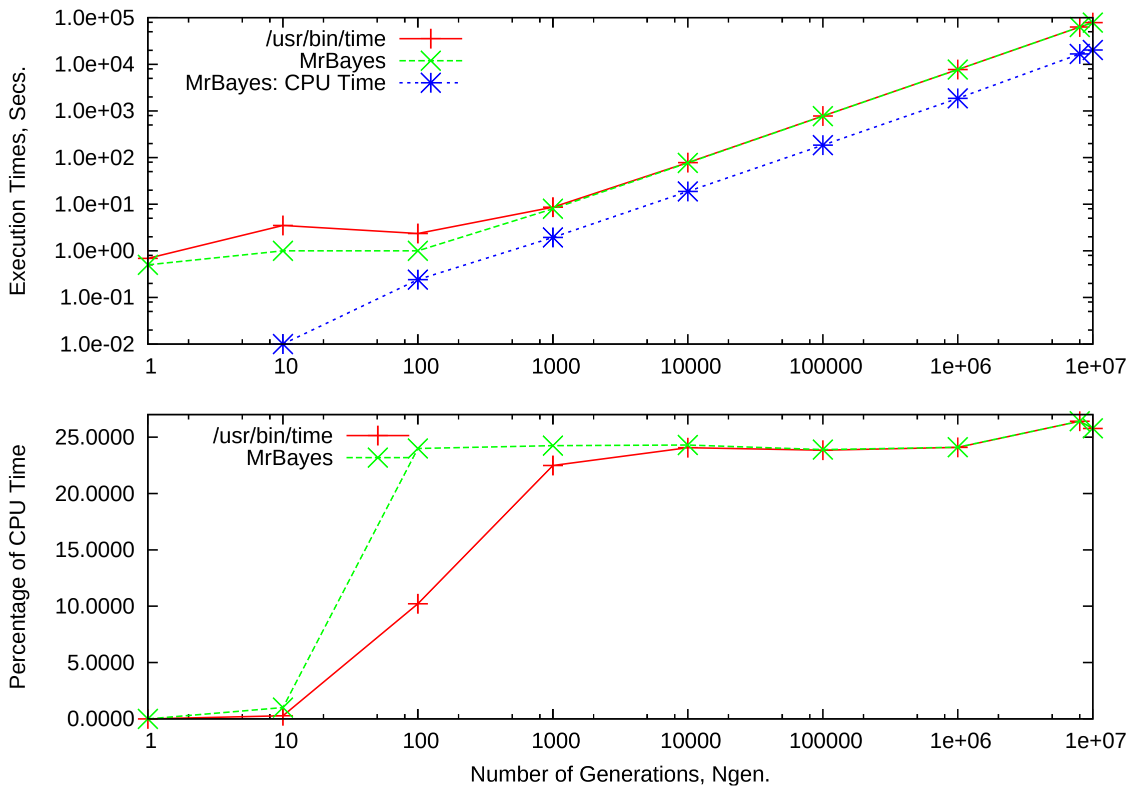

Figure 4.4: Execution Time, CPU Time, and CPU Time Percentage

ants: the I/O and the *No-I/O* variant. For this last variant I commented the function `PreparePrintFiles()`, `PrintStatesToFiles()`, and the code for the *mcmc* file creation.

I measured the total execution and the CPU times for the *No-I/O* variant and as figure 4.3 shows, except for one measurement point, the CPU time was smaller than the execution time for the *No-I/O* variant. This time difference was due to the swapping communication between chains. The lower half of figure 4.3 also presents the *time difference percentage* as I define it in equation 7.3 taking as reference time the $Time_{io}$, the time for the code with file I/O accesses. This graph shows that for number of generations greater than 100 the average value is of 72%. This means that the *No-I/O* variant took only 6.17 hours to compute 10 millions iterations instead of the mentioned 21.9 hours.

4.2.3 Serial File I/O Access Patterns

After analyzing the original code of MrBayes Version 3.1, I identified the following three *processing cycles* within one iteration: computation (A), communication (B), and File I/O (C). Figure 4.5 shows these cycles for one `mcmc` analysis executed with three runs at the same time on four MPI processes (0,1,2,3). The chains belonging to the same run are identified by the same filling pattern. Each run has a total of 5 local chains, which means a total of 15 (0,...14) global chains. The first chain in each run is the cold one, according to the discussion in subsection 2.2.6. The cold chains are illustrated in figure 4.5 as circles within squares to differentiate them from the heated chains, which are only represented through circles. In order to start computation, the chains must be distributed among the participating processes. Each process is assigned Ch_{proc} chains and those processes whose `rank` is smaller than P_{rem} get one

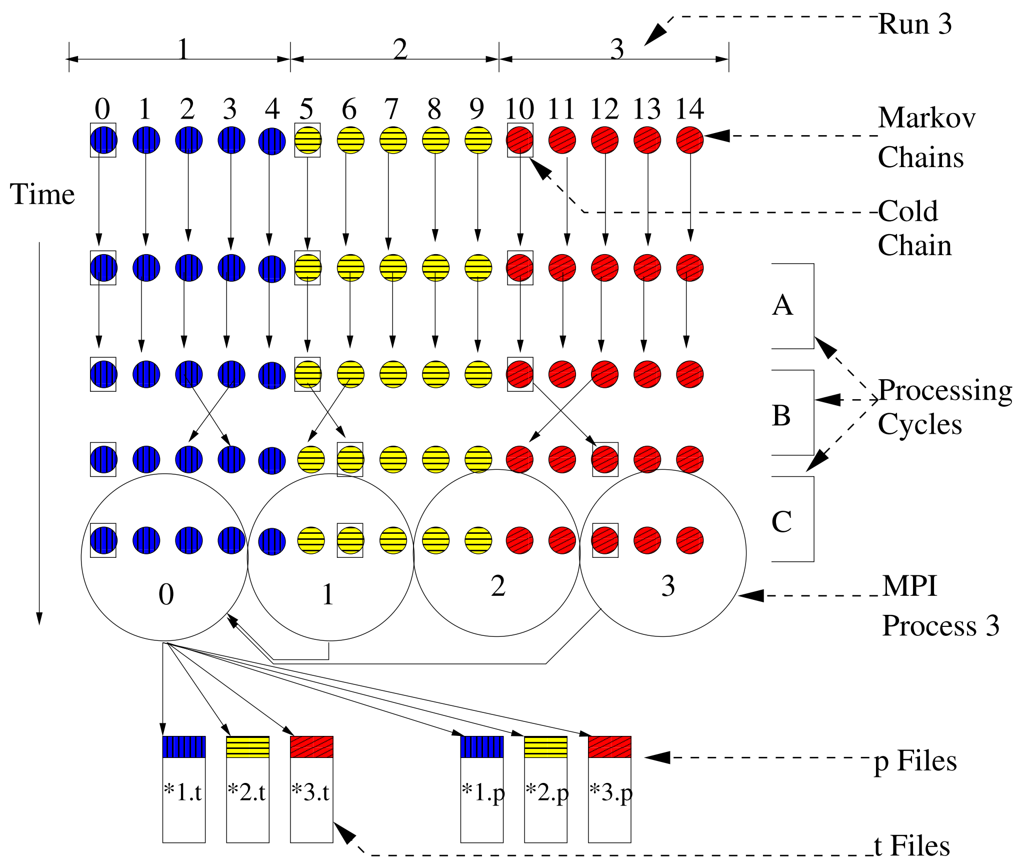

Figure 4.5: Processing Cycles: Processing cycles within an iteration involving I/O activity for an `mcmc` analysis computed by four MPI processes. This analysis involves 3 runs with corresponding 5 chains. During cycles A and B, computation and swapping between chains take place. During cycle C only processes having cold chain (circle within square) send their data to process 0, which writes them into the *online files*.

extra chain[3]. Figure 4.5 shows such an assignment in the time frame before the first cycle A. The most relevant time frame for this work is the file I/O cycle (C). During this cycle the computed results of the cold chains are written into the corresponding p and t files. The write operations are exclusively done by process 0. The processes that have cold chains at this point in time send their results to process 0, which writes the result into the files. I call this communication pattern *many-to-one* and notationally write it as $\{0, ..., n\} \rightarrow 0, n \geq 0$. The case shown by figure 4.5 can be written as $\{0, 1, 3\} \rightarrow 0$, to represent the fact that MPI process with ranks 0, 1, and 3 send data to process 0. A *many-to-one* pattern consists of communication and file accesses. Since such patterns appear only when the secondary storage is accessed and changes during the total execution time, I call them *temporal I/O patterns*. I also consider them I/O and not communication patterns because their existences are conditioned by the used I/O interface, sequential in this case. Figure 4.6 shows the I/O accessing in the serial version. It shows the *many-to-one* patterns as well as the actual writing into the Unix files by process 0. Figure 4.6 depicts a case with four MPI processes computing 3 runs. In a general case process 0 writes the files in the following order: $< FileName > run1.p \prec << FileName > run1.t \prec <<$

[3]For further information on these parameters see appendix A.1.1.

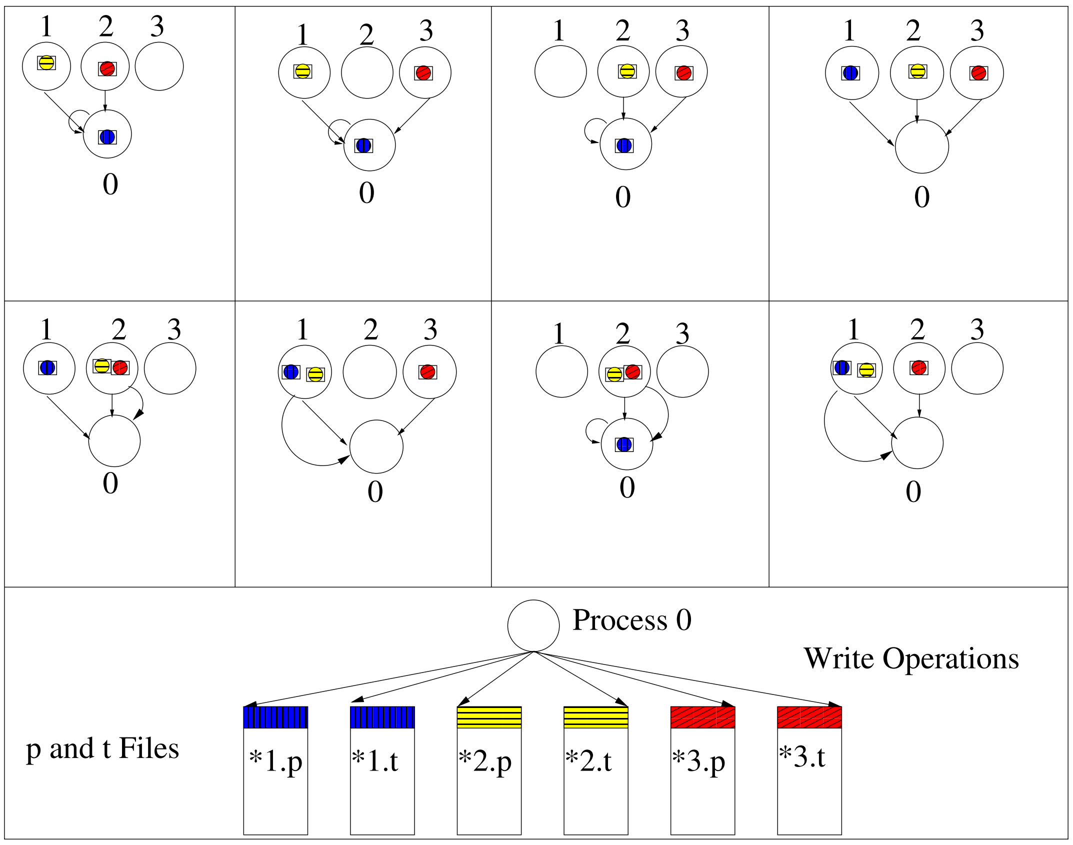

Figure 4.6: Serial File I/O Accesses: These file I/O accesses correspond to a configuration of four MPI processes (0,1,2,3) and three runs (circles within squares). While sending cold chains (arrows) to process 0, the processes generate 8 different *many-to-one* communication patterns. Solely process 0 writes the received data into the files beginning with the leftmost depicted file.

$FileName > run2.p \prec\!\!< FileName > run2.t ... \prec\!\!< FileName > run < nruns > .p \prec\!\!< FileName > run < nruns > .t$. According to the notation of table A.3 in the appendix, this means that for an experiment with n runs, process 0 writes sequentially the lines in the p and t files beginning with the line in the p file of the first run and ending with the t file of the n run. Each entry in the p and t file represents probability and topology of the phylogeny in the respective generation and run.

4.3 Summary

In this chapter I have described the experimental and analytical process conducted to characterize the file I/O activities implemented in the application's original serial I/O version. After determining the complete data flow of the application, I identified the *online files*. These files are accessed during an `mcmc` analysis and their accesses have direct influence on the total execution time. I have described the total amount of data written in these files as well as the patterns generated while doing these operations. I have also described an experiment that show that the application can benefit of a parallel I/O implementation of the *online files*.

5. Temporal Parallel I/O Access Patterns

This chapter presents a theoretical analysis of the p(MC)^3 algorithm in terms of I/O requirements. The objective of this analysis is to quantify and categorize the parallel file I/O access patterns that may appear during the execution of such algorithm on an abstract computing environment with parallel I/O interface. Only after knowing the number and type of these patterns, I propose for the application's parallel I/O implementation the *mapping at first generation* partitioning function in section 6.2.

5.1 Accessing a Parallel I/O Interface

I assume for this theoretical analysis that the algorithm runs with $np > 1$ process that write out at the same time their data into a one dimensional array parallel I/O interface[1]. I am also assuming that in each iteration a number of 2^n (n $\geq$0) pairs of Markov chains (randomly chosen by the algorithm) exchange their states within each of the selected runs.

In such an environment the cold chains, containing compulsory data to be written into secondary storage, randomly move among the processes that are responsible for the corresponding run's computation in each generation. Thus, in this work I call such a relationship of cold chains to processes and the processes' access(es) to the I/O subsystem within a given iteration a *semi-random temporal parallel file I/O access pattern* (PIO_{srt}). Even though the swaps of chains within the scope of a run may happen in a completely random manner, this is is not the case from the perspective of the allocation of Markov chains to the processes, which finally influences the way how processes access the I/O subsystem and generate in a semi-random manner the I/O access patterns. There are two factors that define this semi-randomness. The first one is the p(MC)^3 algorithm's load balacing mechanism based upon Ch_{proc} and P_{rem} (*see Sec. A.1.1*). This mechanism basically assigns Ch_{proc} Markov chains to each MPI process and evenly distributes P_{rem} Markov chains among the processes with rank numbers between 0 and $(P_{rem} - 1)$. The second factor is the non-existence of inter-run swapping between Markov chains.

[1]This one dimensional array view is supported by the ROMIO library that I used for the implementation.

5.2 Number of Parallel I/O Access Patterns

In order to determine the total possible number of *semi-random temporal parallel file I/O access patterns* which may appear during the whole execution time (given a configuration of Markov chains, runs, and processes), I introduce the following definitions:

Definition 5.1 *A swap $sio_{j_i}, j = 1, ..., n, i = 1, ..., ngen$ is an I/O swap in iteration i, if it occurs between a cold and a heated chain, within a certain run, and it involves interprocess communication.*

Definition 5.2 *$Tsio_i$ is the total number of I/O swaps that can occur in iteration i. Correspondingly, $Tsio_1 = \sum_{j=1}^{n} sio_{j_1}$ represents the total number of I/O swaps in the first generation.*

In definition 5.1 and 5.2, n can take a maximum value equal to $nruns*(nchains-1)$[2]. This corresponds to the case, in which each MPI process computes one single chain. Based upon definitions 5.1 and 5.2, I propose equation 5.1 to determine the total number of *semi-random temporal parallel file I/O access patterns, $T_{PIO_{srt}}$* as follows:

$$T_{PIO_{srt}} = \begin{cases} 2^{Tsio_1}, \; nruns > 1 \\ np, \; nruns = 1 \end{cases} \tag{5.1}$$

Equation 5.1 expresses that conducting the analysis with one run ($nruns = 1$), any of the np processes can write the cold chain. Thus, the number of different I/O patterns equals the total number of processes np. The fact that an I/O swap sio_{j_i} may or may not occur is expressed through the binary base.
Using equation 5.1 to determine the possible number of *semi-random temporal parallel file I/O access patterns* for the configuration of figure 4.5, for example, I identify the I/O swaps shown in figure 5.1. This means that for this configuration there are 8 possible different *semi-random temporal parallel file I/O access patterns,* which may semi-randomly appear through the whole execution time. In this particular case, these patterns correspond to the 8 *many-to-one* patterns that I show in figure 4.6.

5.3 Types of Parallel I/O Access Patterns

In this section I propose a categorization in four types for the $T_{PIO_{srt}}$ different *semi-random temporal parallel file I/O access patterns* that can be generated as discussed in section 5.2. In order to be able to propose this categorization, I introduce definition 5.3 and 5.4. I also introduce definition 5.5 as complementary to definition 5.4. For these definitions I assume that the same I/O operation (write, read, sync, etc.) is conducted at a given point in time.

Definition 5.3 $\Pi_{IO}(\Delta\tau)_i = (\pi, \omega, \sigma)(\Delta\tau)_i$ *is a semi-random temporal parallel I/O access pattern in the ith time interval $(\Delta\tau)_i$, $i \geq 1$. In this expression each of the three arguments is a field, a one dimensional array. The elements of field $\pi[0...(np-1)]$ are scalar and represent the processes' ranks. In the offsets' field $\omega[[[...[...]...]_l]_k^j], j = 0, 1...\delta, k = 0...(np-1), l = 1...\lambda$ the elements can be scalar or*

[2]See appendix A.1.1 for notation.

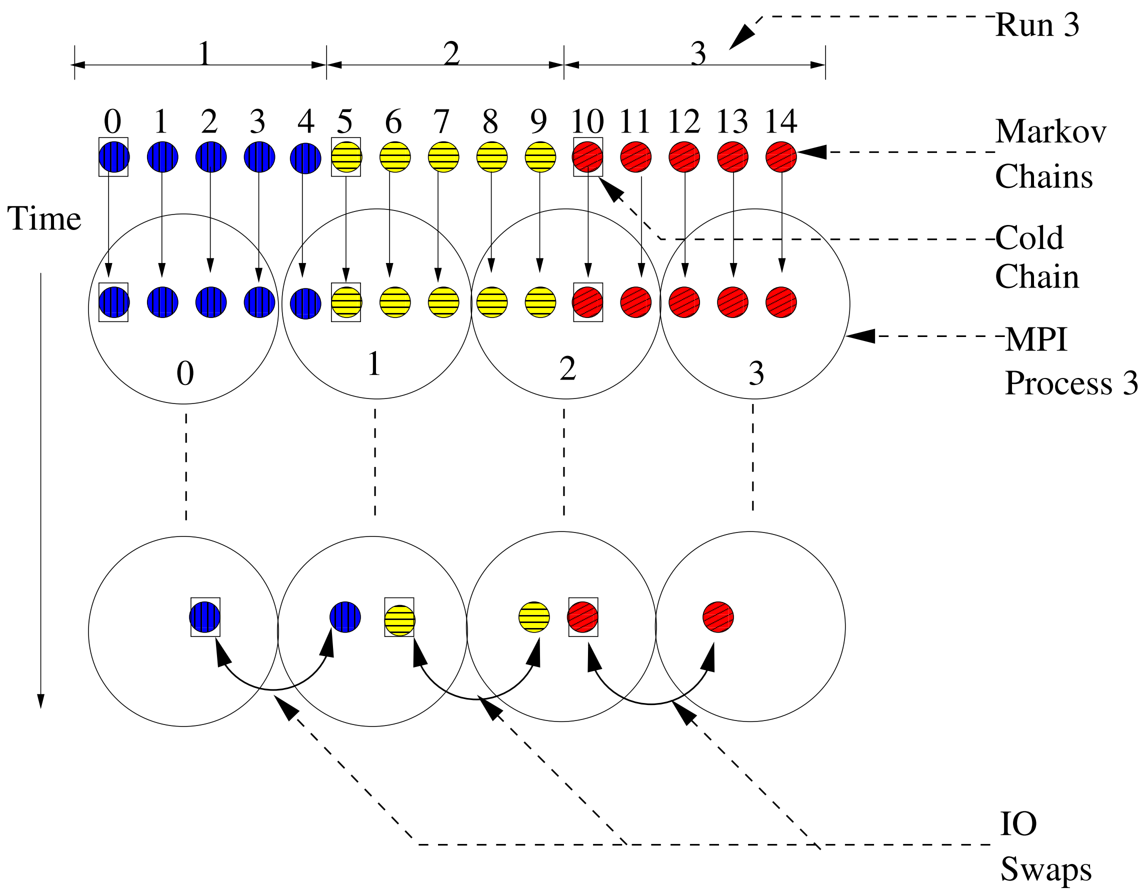

Figure 5.1: Total Number of I/O Swaps during the first iteration for a configuration with 3 runs, 5 chains, and 4 MPI Processes

fields depending on the chosen value for δ, which defines the nesting level of fields. A value of $\delta = 0$ means that the elements of ω are scalar. The scalar elements of these fields represent offsets in secondary storage to be accessed at the same time during $(\Delta\tau)_i$ by process k. The field σ contains the sizes corresponding to the offsets in ω.

In definition 5.3 the variable *np* stands for the number of chosen MPI processes and the variable λ represents the number of lines to be kept in main memory with the *parallel line buffering* mechanism (*see* Sec. *6.3.1*). In the parallel I/O version the amount of data in bytes from the perspective of one single MPI process that corresponds to $\lambda = 1$ results from multiplying twice the number of assigned runs to this process by the size of the *p* and *t lines*.

Definition 5.4 *$PIO_{k,l}((\Delta\tau)_i), k = 1...np, i, l \geq 1$ is an I/O process. This is an MPI process, which is involved in l I/O operation(s) within a time interval $(\Delta\tau)_i$.*

Definition 5.5 *A Null I/O process $PNIO_{k,0}((\Delta\tau)_i), i \geq 1$ is a process that points to the element $\omega[[[NULL]_0]_k]$ $((\Delta\tau)_i)$ within the time interval $((\Delta\tau))_i$. It does not access the I/O subsystem in $((\Delta\tau)_i)$.*

In definition 5.3 the values contained in field ω can be represented without nested fields. Nevertheless, I propose this notation in order to assign a *context* to the I/O access. This context is a compositum to express the kind of data for the access and is application dependent. The different number of contexts can be represented through the δ variable, which I call the *degree of parallel I/O context.* I represent n different contexts as $\delta = (n-1)$. For the application in this work I have defined 2 different values for the *degree of parallel I/O context.* I have defined $\delta = 0$ to represent the parallel I/O at the run level and $\delta = 1$ for accesses at the t and p *lines* level. Figure 7.2 shows a set of patterns with $\delta = 0$ within an iteration and figure 6.2 shows a set of five different *semi-random temporal parallel file I/O access pattern* with $\delta = 1$ that can be generated within one iteration.
Based upon the concept of a *semi-random temporal parallel file I/O access pattern* (definition 5.3), an I/O process (definition 5.4), and a null I/O process (definition 5.5), I propose a classification of four different types of *semi-random temporal parallel file I/O access patterns* at the run level $\delta = 0$ as I show in table 5.1.
I denote such patterns as *semi-random temporal* to refer to the fact that the values of the parameters that define them (including k and l) change between two consecutive time frames $(\Delta\tau)_i$ and $(\Delta\tau)_{i+1}$ in a semi-random manner.
If the assignment of cold chains to the processes is bijective and load balanced there are no I/O swaps $((nruns \bmod np) = 0; nruns \geq 2)$, the exponent of equation 5.1 is zero. This means that under these conditions one single pattern of type *Complete-1*, in the proposed categorization of table 5.1, is generated. For simplicity I call it pattern π_1.

Type	**Description**	**Set Description**
Complete-1	Each I/O process is involved in exactly one I/O operation	$np = \{\bigcup_{k=1}^{np} PIO_{k,1}\}$
Partial-1	Each I/O process is involved in exactly one I/O operation, but the number of these is smaller than np, the total number of processes.	$np = \{\bigcup_{k=1}^{A} PIO_{k,1}\} \bigcup \{\bigcup_{k=A+1}^{np} PIO_{k,0}\}$
Complete+1:	Same as Complete-1, but at least one I/O process is involved in at least two I/O operations	$np = \{\bigcup_{k=1}^{A} PIO_{k,1}\} \bigcup \{\bigcup_{k=A+1}^{np} PIO_{k,l\geq 2}\}$
Partial+1:	A variant of Partial-1, but as in the case of Complete+1 here at least one I/O process is involved in at least two I/O operations.	$np = \{\bigcup_{k=1}^{A} PIO_{k,0}\} \bigcup \{\bigcup_{k=A+1}^{B} PIO_{k,1}\} \bigcup \{\bigcup_{k=B+1}^{np} PIO_{k,l\geq 2}\}$

Table 5.1: Types of *semi-random temporal parallel file I/O access patterns,* $\delta = 0$

5.4 Summary

In this chapter I have introduced formal definitions for the description of *parallel I/O access patterns* within an iteration and throughout the whole execution time of the p(MC)^3 algorithm while using a parallel I/O interface. I have proposed an expression to quantify the different number of *semi-random time parallel I/O access patterns* that can appear during the algorithm's execution time. Furthermore, I have proposed a four type categorization for these patterns at the run level ($\delta = 0$).

6. Parallel I/O Adaptation

The main objective of this chapter is to present the parallel I/O adaptation that I have proposed for the p(MC)^3 algorithm. Basically I followed two objectives through this parallel I/O adaptation. The first one was to eliminate the I/O bottleneck inherent to the *many-to-one* communication patterns during I/O activities, in order to reduce the algorithm's execution time. The second one was to enable the application to compute comparatively more runs than its serial I/O counterpart, in order to correspondingly increment the phylogeny's approximation computed by an `mcmc` analysis.
The first step of this parallel I/O adaptation consisted in proposing a parallel file format to store the generated data. Section 6.1 presents one such format. In section 6.2 I discuss the *mapping at first generation* partitioning function for I/O operations. This is a novel function for partitioning a parallel file among many processes whose temporal and spatial parallel I/O accesses vary in a semi-random manner. Subsection 6.3.1 describes how the *MFG* partitionig function interacts with MPI_hint based physical distribution functions. It especially describes the *parallel line buffering*, a mechanism that I propose to select different *varstrip chunk size* values.

6.1 Parallel File Format

There are different possible format configurations in which the generated total amount of data expressed in equation 4.1 can be stored in one single parallel file. In this work I propose the format presented in figure 6.1. It consists of one header that contains metadata information on the corresponding trailer. The trailer is divided in areas for each run. These areas are subdivided in p and t areas that correspondingly contain the p and t *lines*. These areas store the parameters of the computed trees and their topologies, respectively. The format shown in figure 6.1 corresponds to the results of one `mcmc` analysis.

6.2 The *MFG* Partitioning Function

This section describes the *mapping at first generation* partitioning function, after having determined the total amount of data and its type in chapter 4 and the corresponding *semi-random temporal parallel file I/O access patterns* generated from the p(MC)^3 algorithm in chapter 5.
As I have mentioned in subsection 4.2.3 each serial I/O write operation consists of two phases: the *many-to-one* communication pattern and the actual write operation into 2*n files. The second phase has a time complexity of $O(n)$ for one write operation, with n representing the number of runs being computed. I propose in this work

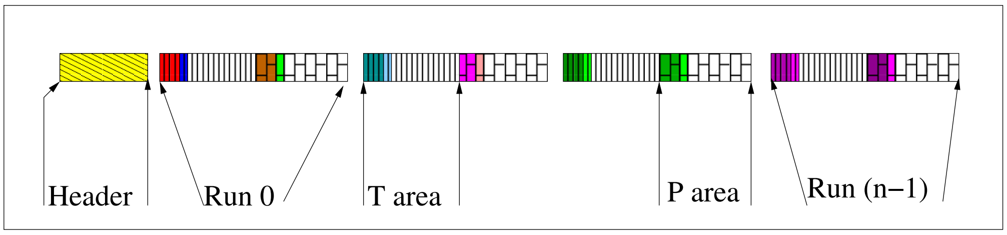

Figure 6.1: One proposed parallel file format for MrBayes3.1

the *MFG* partitioning function to store the data into a parallel file with the format described in section 6.1. Due to this partitioning function all processes are able to write their data at the same time into the parallel file. In a *Complete-1* pattern, for instance, the time complexity of one write operation is $O(c)$ independently on the computed number of runs. For other types of patterns categorized in table 5.1, c is defined by the slowest process involved in the write operation. The value of c can be reduced through the selection of adequate application and system parameters, such as access patterns and distribution function related parameters as presented in chapter 7.
Figure 6.2 depicts a set of *semi-random temporal parallel file I/O access patterns* with $\delta = 1$ that can be serviced with the *MFG* partitioning function, including a special case for one single MPI process. Table 6.1 presents the pseudo-code of the *MFG* partitioning function in a `C` based pseudo-language. This pseudo-language uses the same notation and definitions from chapter 5 and the appendix A.

6.2.1 Parallel File Map, Local Runs

The time interval $(\Delta\tau)_i, i \geq 1$ from definition 5.3 in this application refers to a time frame at the end of each iteration *gen*. During $(\Delta\tau)_1$, at the end of the first iteration, each process computes the size of the p and t *lines*. Based upon this information and the provided file format ϕ, the total number of runs ρ, the total number of generations $ngen$, and the sampling frequency sf, each process is capable to construct the file map μ. This map is a structure that consists of offsets of all runs within the parallel file and offsets of the t and p sections within a run and corresponding sizes. This map corresponds to the union set operation of all ω and σ fields from defintion 5.3. Once the μ map has been constructed by each process during the first iteration, the next step of each process is to determine the l runs that they compute in each generation. This is dependent on the used load balancing mechanism.

6.2.2 Filling Main Memory Buffer β

Each process fills its main memory buffer β with the l runs that it has received from the *lbm* load balancing mechanism. The variable l represents the local number of runs. This main memory buffer β will be filled up to λ generations.

6.2.3 Parallel File Opening

During file opening the distribution function, the striping unit, and *varstrip chunk size* are set by each one of the computing processes to the user's provided values. If the *semi-random temporal parallel file I/O access pattern* is a π_1 and the varstrip

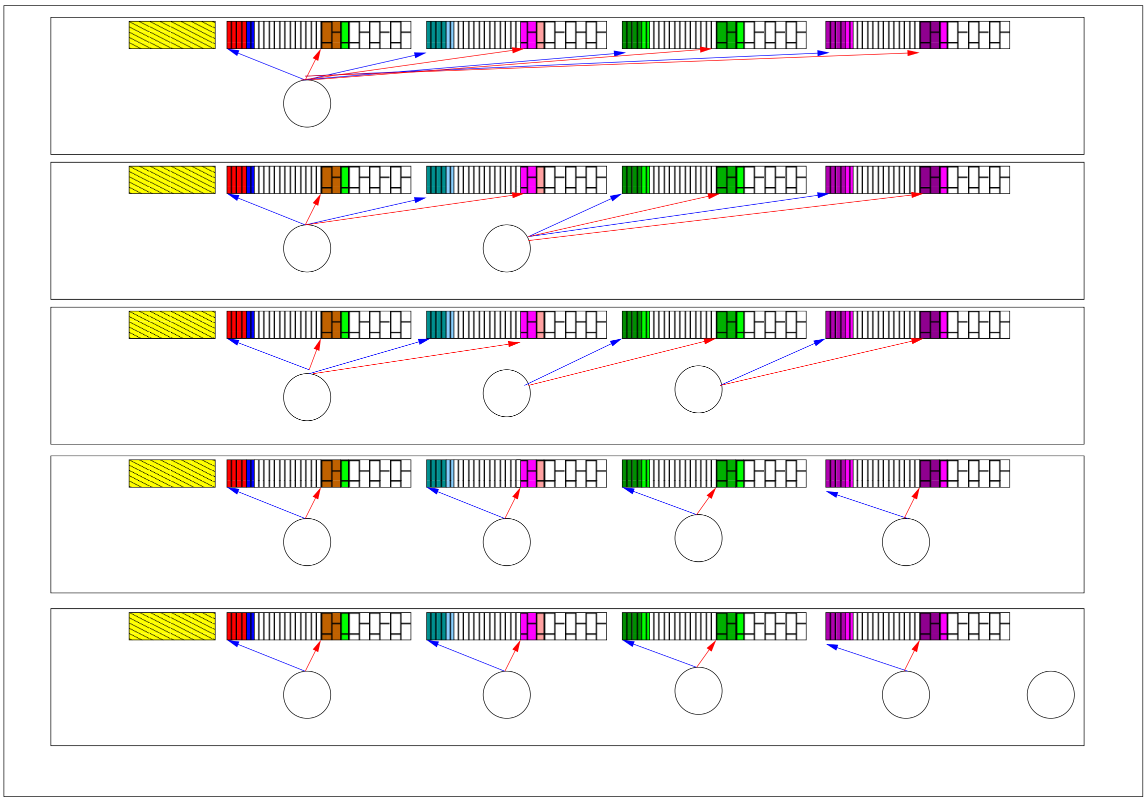

Figure 6.2: A set of *semi-random temporal parallel file I/O access patterns* with $\delta = 1$: A set of 5 different patterns that can be served with the *MFG* partitioning function. This is an example of an `mcmc` analysis with 4 runs that is executed with 1 - 5 MPI processes.

distribution function is chosen by the user, the *varstrip chunk size* will be set to the value of λ. If no parallel I/O parameters are selected by the user, the simple_stripe distribution function with its default value will be set, else the provided striping unit value will be applied to the simple_stripe distribution function to open the file.

6.2.4 Creating the File View

Before finally writing into the parallel file, each I/O process constructs its own view of the file $\prec_{file}$. In this work such a parallel file view $\prec_{file}$ is one data structure per process that contains in a linearized representation the offsets and sizes, where the set of p and t *lines* contained in β will be written into the parallel file.
This $\prec_{file}$ is constructed based upon the file map μ, the current writing generation wg, and in the case of the varstrip distribution function the *parallel line buffering factor* λ. Within the writing iteration wg each process constructs a one dimensional array which contains offsets and corresponding sizes. This array is an association of each offset contained in ω from definition 5.3 to its corresponding sizes contained in σ. In order to implement this association using the programming elements of MPI-2, I constructed an `MPI_Datatype` based upon one array of only offsets and another of only sizes with the `MPI_Type_create_hindexed` that had the data to be written. Using the `MPI_File_set_view`, I associated these datatypes to the corresponding file handle.

```
/*Process global variables:*/
$ngen$                            /* Total number of generations*/
$gen$                             /* Current generation */
$wg$                              /* Writing Generation */
$sf$                              /* Data Sampling Frequency */
$\rho$                            /* Total number of runs */
$\lambda$                         /* *parallel line buffering* factor*/
$lbm$                             /* Load balancing mechanism */
$pf$                              /* Parallel file handle */
$\phi$                            /* Parallel file format */
$\mu$                             /* Parallel file map */
                                  /* Distribution Functions:  */
$s$                               /* Simple_stripe */
$su$                              /* striping unit */
$v$                               /* Varstrip */
$vscs$                            /* *varstrip chunk size* */
$\pi_1$                           /* Complete-1 pattern */
                                  /* at $\delta = 0$ level*/
/*Process local variables:*/
$\beta$                           /* Buffer in main memory */
$l$                               /* Number of runs */
$\prec_{file}$                    /* Process's file view */
while($gen \leq ngen$){
$\mathbf{PIO}_{k,l\geq1} \wedge \mathbf{PNIO}_{k,l=0}$:
if (($\rho$ % np) == 0) { $\pi_1$ = YES }
if($gen == 1$)                    /**mapping at first generation**/
                                  {Compute $\mu(\phi, \rho, ngen, sf)$}
if(($gen$ % $sf$) == 0)
                                  {Determine $l$ based on $lbm$
                                  FillBuffer$\beta(l * \lambda)$}
if($gen == \lambda$)
                                  {if(($\lambda == 1$) $\wedge$ ($s$ selected))
                                  {PFopen($su$)}
                                  if(($\lambda \geq 1$) $\wedge$
                                  ($v$ selected) $\wedge$ ($\pi_1 == YES$))
                                  {PFopen($vscs$)}}
if(($gen$ % $wg$) == 0) {
$\mathbf{PIO}_{k,l\geq1}$:
                                  CreateType $\prec_{file}$ $using(\mu, wg, \lambda)$
                                  Associate $\prec_{file}$ to $pf$
                                  PFWriteOut($pf, \beta$)
$\mathbf{PNIO}_{k,l=0}$:
                                  Barrier
}}
```

Table 6.1: Pseudo-code of the *mapping at first generation* Partitioning Function.

6.2.5 Parallel Write Out

A process that buffers at least one run in its main memory β participates in the parallel file write out operation performed in each writing generation wg.
Each I/O process writes out its β into the file using the `MPI_File_write`. During this time the $PNIO$ processes, definition 5.5, call the `MPI_barrier`.

6.3 Accessing Flexible Distribution Functions

The user can select at the application level the type of physical distribution function, with which the parallel file system opens the parallel file. Once a function has been chosen, the user can also set the striping unit value or the *varstrip chunk size* for the simple_stripe or varstrip distribution function, correspondingly.
The selection of the function is limited by the type of *semi-random temporal parallel file I/O access patterns* to be generated during execution. For a constant π_1 pattern both functions can be chosen. For any sequence of the patterns categorized in table 5.1 at $\delta = 0$ or table at $\delta = 1$, only the simple_stripe function should be chosen. In chapter 7 I present thorough information about choosing optimal distribution functions and their parameters.

6.3.1 *The Parallel Line Buffering Mechanism*

For the usage of the varstrip distribution function with the *MFG* partitioning function I propose the *parallel line buffering* mechanism. In each generation g each process that has cold chains, buffers in main memory β the lines for the t and p area of these chains. In this work I have defined as default value for the *varstrip chunk size* within the varstrip distribution function a value corresponding to $\lambda = 1$, which means that each process writes the content of its main memory buffer β within each generation. Nevertheless, if the size of these lines are relatively small this might yield to unnecessary I/O overhead, which might decrease performance. In order to overcome this situation, I propose the *parallel line buffering* mechanism that consists in keeping the results of more than one generation in the main memory buffer before writing the data into the file, $\lambda > 1$. Figure 6.3 shows the synchronization of the `C` functions that I have used to implement the parallel line buffering mechanism. This graph depicts a case of an `mcmc` analysis with 20 iterations when retaining 6 lines in main memory ($\lambda = 6$)[1]. The number in circles, 1 - 4, show the chronological order in which the `C` functions are called.

6.3.2 Proposed λ_{max}

Buffering results in main memory should not activate the operating system swapping. In equation 6.1 I propose an expression for the computation of λ_{max} for a given configuration so that operating system swapping is not activated. In this expression M_T represents the physical main memory size, M_{ST} the swapping threshold memory size, M_{Proc} the process memory size, M_{Line} the line memory size, and R_l the number of local runs assign to an MPI process.

$$\lambda_{max} = \frac{M_T - (M_{ST} + M_{Proc})}{M_{Line} * R_l} \tag{6.1}$$

Depending on the I/O cluster configuration, different values for λ_{max} can be computed across all I/O nodes. In order not to activate operating system swapping on

[1]A value of $\lambda = $ y can be selected with `dnmlines = y` within the `mcmc` prompt.

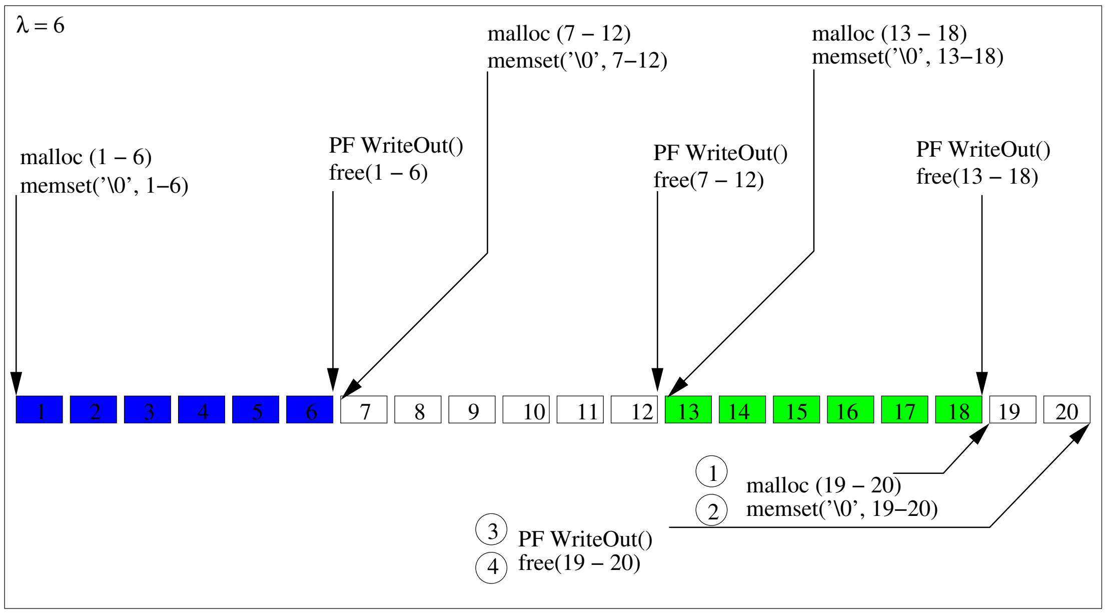

Figure 6.3: Synchronization of `C` functions for the implementation of the *parallel line buffering* mechanism: This example shows the execution of 20 iterations while retaining 6 lines in main memory ($\lambda = 6$). The number in circles (1 - 4) show the order of the `C` calls.

any node, the smallest λ_{max} should be used. Furthermore, I propose to use an upper limit of only 75% of this allowed maximum number. If during the first generation the operating system activates swapping, I propose to use $\lambda = 1$. Equation 6.1 also provides information on the number of runs that can be selected before activating operating system swapping based upon a chosen λ and the line memory size. For a π_1 pattern, load balanced, this can be determined through the multiplication of R_l by the number of I/O nodes.

6.4 Summary

In this chapter I have presented the two main contributions of this work. Firstly, at the application level I have described the *mapping at first generation* partitioning function. This is a partitioning function for applications in which time parallel I/O access patterns are semi-random during execution. Nevertheless, the application must have a defined file format and during the first generation each line type must be computed so that using the format and these initial values, all offsets and sizes within the file can be determined in a map during the first generation. This map must contain information about the position in the file of a line type at a given generation. During a writing generation a process with lines to be written into secondary storage looks up the corresponding file position in the map for the lines in question at the given writing generation. Finally it writes into those positions. In this particular case, I have implemented the *MFG* function for an application with a set of *semi-random temporal parallel file I/O access patterns.* Nevertheless, the strength of the *mapping at first generation* is its independency, due to the map existence, from the number of processes that are involved in an I/O operation. Therefore it is also suited for application with random temporal parallel I/O access patterns.
Secondly, at the application level I have also proposed the *parallel line buffering* mechanism to set the *varstrip chunk size* value at the parallel file system within the varstrip distribution function for π_1 patterns.

7. Evaluation

The purpose of this chapter is to present the set of experiments that I conducted to show the performance of the *mapping at first generation* partitioning function in conjuction with the varstrip and the simple_stripe distribution function. In this work this performance is expressed in terms of speedup and *time difference percentage* values as defined under subsection 7.3.1.
The speedup of a parallel I/O MPI based program running on an I/O cluster computer is conditioned by different setups of the parallel processing environment and the program's parameters. From these setups and parameters in this evaluation I took into consideration the type of distribution function and corresponding settings, the used number of physical I/O nodes, the number of MPI processes per node, the total number of runs, and the number of Markov chains per run.
The application has parameters whose values change the execution time independently of the used I/O interface. One of such parameters is the temperature with which the chains are provided. In this evaluation I used a temperature middle value of 0.5 and default values for similar parameters.

7.1 Overview of Experiments

The following enumeration presents an overview of the set of conducted experiments in this evaluation:

1. Set of experiments to demonstrate the support of all identified temporal I/O access pattern types generated from the p(MC)^3 algorithm by the *mapping at first generation* partitioning function in conjuction with both distribution functions (*see Sec. 7.4*): Subsection 7.5 presents the validation of the *MFG* function with the simple_stripe distribution function to handle all patterns and with the varstrip distribution function to handle π_1 patterns. Furthermore, subsection 7.5.1 demonstrates that these patterns present different speedup values.

2. Experiments to determine π_1 pattern speedup (*see Sec. 7.6*): Subsection 7.6.1 describes the experiments that I conducted to show the speedup of the *MFG* partitioning function with both distribution functions to compute π_1 patterns that generate different amount of data in each iteration $data_{gen}$ and have different number of runs. Subsection 7.6.2 describes the speedup behavior in terms of different MPI processes on 2, 4, and 8 physical nodes.

3. Pattern π_1 speedup in terms of distribution function parameters (*see Sec. 7.7*): Subsection 7.7.1 presents the experiments that I conducted to determine the speedup dependency on the striping unit values for the simple_stripe distribution function. Correspondingly subsection 7.7.2.1 describes the experiments to demonstrate the functionality of the *parallel line bufferig* mechanism for the varstrip distribution function. Furthermore, for this mechanism in subsections 7.7.2.2 - 7.7.2.3 I demonstrate how different λ values provide different speedup values for different amounts of data.

4. Speedup in terms of number of iterations (*see Sec. 7.8*): Section 7.8 describes experiments that I conducted to show the speedup's behavior in terms of the number of iterations for 8 runs (the smallest number of runs to exploit parallel I/O functionalities on the used I/O environment described under section 7.2) and 64 runs. Finally section 7.9 describes an experiment that I conducted to provide a speedup comparison to compute 128 runs with a π_1 pattern.

7.2 I/O Environment

The deployment of the *MFG* partitioning function with the simple_stripe distribution function supports each one of the time parallel I/O access patterns that I have categorized in table 5.1. Nevertheless, as we mention in [LuLu05] and [EGVL05] we propose the varstrip distribution function for spatial application access pattern of type 1 (*see Sec. 3.1*). In the context of this application this corresponds to a π_1 pattern with $(nruns \bmod np) = 0$. Therefore, in order to compare the performance of the *mapping at first generation* function with both physical distribution functions, pattern π_1 should be supported at all levels of the parallel I/O environment as I describe in this section.
In this evaluation I utilized 8 nodes of a 9 nodes I/O cluster. Each of these nodes has an SMP architecture with 2 processors, a 1GB main memory, and one secondary storage device with a capacity of 80GB. The nodes 01-05 are provided with two extra 160 GB disks that work as RAID0[1]. These nodes communicate with each other over a gibabit ethernet using a star topology. Their technical characteristics are included under appendix A.3.10.1 and their performance measurements can be found under [PuVe10].
At the parallel file system I used pvfs-2.8.1 that already included our proposed varstrip distribution function. The configured parameters in the `config.log` file's header are shown in appendix A.3.9.1. Each PVFS2 node was configured using the *all in one* configuration, in which each node has the role of client, server and metadata server at the same time. The corresponding used `pvfs2.conf` file is shown in appendix A.3.9.2.
I used the mpich2-1.0.8p1 as parallel I/O interface library, to which I applied our software code that enables the applications to choose the distribution function types and set their corresponding parameters. The `config.log` file's header for the used MPICH2 is included under appendix A.3.8.1.

7.3 Input Matrices and Metrics

At the parallel application level I used two sets of input matrices. One set consisted of biological DNA strings and the other consisted of synthetical strings.
The set of biological strings consisted of matrices with 8, 12, 16, 32, 64, 128, 256, 512,

[1]Each of these nodes has an operating system swapping partition space of 2.93 GB.

768, and 1024 taxa. Each of these taxon had a DNA character length of 2086 characters. These were matrices of anatidae, whose biological context can be found in [GoDW09]. I also used the 12 taxa with character length 898 from the `primates.nex` file that is provided with MrBayes3.1. Detailed information about the input matrices can be found under appendix A.3.2.

In order to use any input matrix, I wrote a program called `gtaxa`. This program creates synthetical taxa in an input file conform with the nexus file format [MaSM97]. It randomly creates an input matrix based upon the given number of taxa and string length[2].

The dimensions of an input matrix can be expressed in terms of the number of taxa and the corresponding DNA string lengths or in bytes. Depending on the used input matrix for an analysis, a certain amount of data $data_{gen}$ is generated to be written into secondary storage in each iteration for a chosen number of runs.

In order to establish the correspondence between matrix dimensions and $data_{gen}$, I synthetically created matrices with number of taxa between 10 and 10000 and whose string length were 10, 100, 1000, and 10000 characters. Using these matrices as input, I ran 8 runs 10 iterations on eight MPI processes and physical nodes. Figure 7.1 presents the measured $data_{gen}$ values for number of taxa between 10 and 90. Figure 7.1 shows in the lower half that using different input dimensions yields different execution times, but in the upper part it shows that the amount of data $data_{gen}$ generated and written in each iteration solely correlates with the number of taxa. Figure A.1 in the appendix depicts this correlation for up to 10000 taxa.

The evaluation in this work strongly depended on the amount of data $data_{gen}$ generated in each iteration. Nevertheless, it is not obvious to determine it. Therefore, besides the information from figure 7.1 and A.1 table A.5 and A.6 in the appendix provide cross referencing information. This information includes: the input matrix sizes, distribution function parameters, and the amount of data $data_{gen}$ generated in each iteration. This last parameter is expressed in these tables in terms of *physical degree of I/O parallelism* and *depth of I/O parallelism.* For these two terms I used a reference value of 64 KB, the simple_stripe distribution function striping unit default value.

7.3.1 Speedup, Efficiency, and Time Difference Percentage

This subsection lists the metrics that I use to express the performance measured in this evaluation. In the context of this work I define the speedup in equation 7.1 based upon Amdahl's law.

$$speedup = \frac{Time_{serial_{IO}}}{Time_{parallel_{IO}}} \tag{7.1}$$

In this equation $Time_{serial_{IO}}$ and $Time_{parallel_{IO}}$ represent the execution time of the serial and parallel I/O program versions, respectively. In equation 7.2 I define the corresponding efficiency in terms of the number of processing units $efficiency_{CPU}$.

$$efficiency_{CPU} = \frac{speedup}{NumCPUs} \tag{7.2}$$

In this expression $NumCPUs$ represents the number of used CPUs.

Another metric that I use to compare two or more execution time values is what I call in this work the *time difference percentage* as I define it in expression 7.3

$$\Delta_{perf} = \frac{Time_{ref} - Time_i}{Time_{ref}} * 100 \tag{7.3}$$

[2]The `gtaxa` is also included under the source code tree of the parallel I/O version.

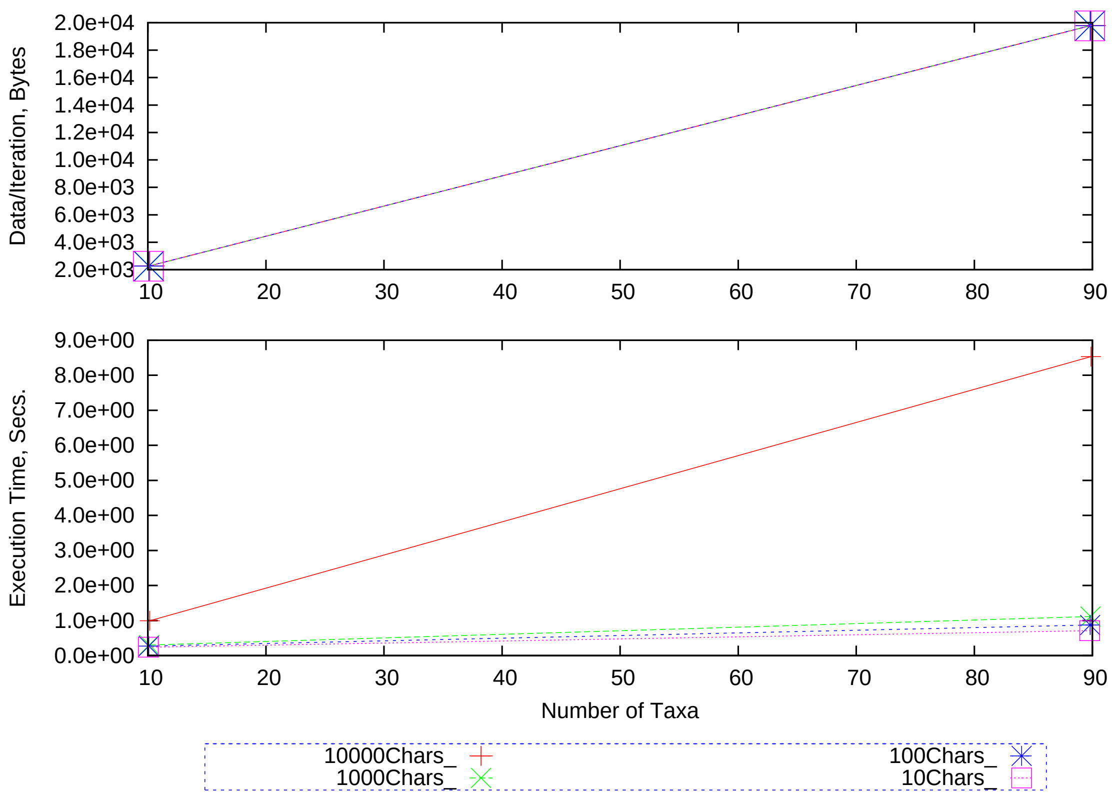

Figure 7.1: Amount of data generated in each iteration: Data generated in each iteration in terms of different taxa and four different string lengths. In the lower half the corresponding execution times are presented.

In equation 7.3 $Time_{ref}$ corresponds to the reference execution time that I select by applying certain criteria. In this evaluation some values assigned to $Time_{ref}$, for instance, correspond to the execution time with the default striping unit or *varstrip chunk size* values. The $Time_i, i = 1...n$ variable represents the n execution time values to be compared.

7.4 *Mapping at First Generation* Validation

The first question to answer is whether or not the categorized semi-random access patterns in table 5.1 are correctly served by the combination of the *mapping at first generation* with the simple_stripe distribution function and its striping unit of 64 KB and with the varstrip distribution function, in the case of the π_1 pattern. In the set of measured pattern configurations I did not conduct extra measurements for the *Partial+1* type. From the patterns shown in figure 7.2 such a pattern is created based upon pattern *Partial-1*, when one of the MPI processes writes two runs and within the same iteration two processes do not write any data at all. Thus, the functionality of such pattern can be deduced upon the functionalities of the *Partial-1* and *Complete+1* patterns.
In this work I consider correct patterns servicing, if independently of the pattern chosen during the first iteration, at the end of an analysis the application creates the parallel file with the appropriate format and content. Furthermore, the parallel file must also be stored on the I/O nodes with the required distribution functions' settings.
Once this has been clarified, the next question to answer is whether or not these

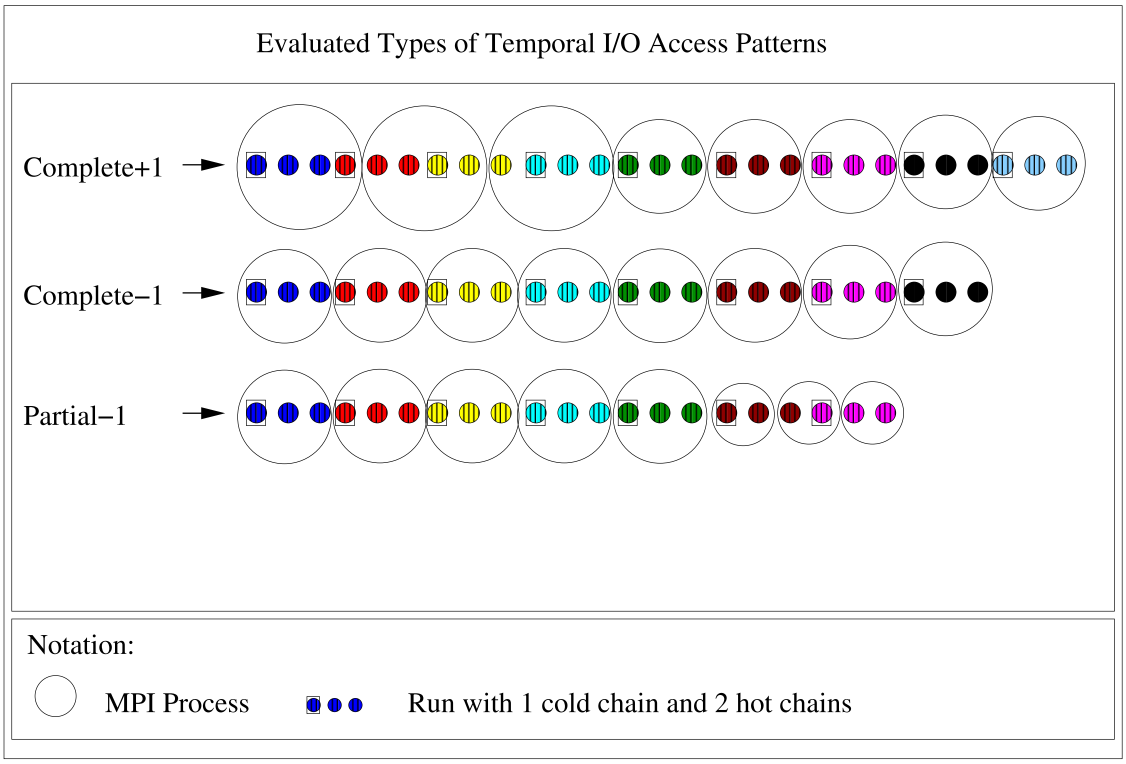
Evaluated Types of Temporal I/O Access Patterns
Complete+1
Complete−1
Partial−1
Notation:
MPI Process
Run with 1 cold chain and 2 hot chains

Figure 7.2: Evaluated types of temporal I/O parallel access pattern, representation at the $\delta = 0$ level, used to test the functionality of the *MFG* partitioning function in combination with the simple_stripe distribution function and the varstrip distribution function. This last distribution function serves only the *Complete-1* pattern.

patterns are served with the same speedup on an I/O environment as the one discussed in section 7.2. This section presents the set of experiments that I conducted to answers these two questions.
For the set of experiments described in this section I used the patterns depicted in figure 7.2. This figure shows the configurations in terms of MPI process, runs, and Markov chains that I utilized to generate these patterns.

7.5 Support of the p(MC)^3 I/O Temporal Access Patterns

This section describes the set of measurements conducted to validate the *mapping at first generation* partitioning function in conjuction with default values for distribution function related parameters to serve the categorized p(MC)^3 patterns from table 5.1. In the case of the π_1 pattern I conducted measuremets of the *mapping at first generation* performance with both distribution functions. I ran each one of the patterns shown in figure 7.2 for one iteration using an input matrix that generated a parallel file with a size of 5.4 KB. This experiment showed a worst case when all processes write at the same time into the parallel file from the logical point of view, but due to the small data size a tiny parallel file is written into one single physical I/O node by the simple_stripe distribution function and its striping unit default value[3]. The input synthetical matrix's header is included under appendix A.5.1.

[3]In the rest of this evaluation in most of the cases the parallel file is stored on many I/O nodes indepedently of the striping unit value.

The generated parallel files were stored with the simple_stripe distribution function and its striping unit default value for each pattern. In the case of the *Complete-1* or π_1 pattern, I also conducted an analysis, in which the parallel file was stored with the varstrip distribution function and its default *varstrip chunk size* value.
For each case I recorded the parallel file content, the information about distribution functions, and used I/O nodes. I include this information in appendix A.5.2, A.5.3 , and A.5.4, for the *Complete+1*, *Complete-1*, and *Partial-1* patterns, respectively. The content of these parallel files show that the results of many runs, in this case 7, 8, and 9, that are part of the same `mcmc` analysis are written with the defined format in section 6.1 into one single parallel file independently of the chosen temporal parallel I/O access pattern. The information on these parallel files in the subsections A.5.2.2, A.5.3.2, and A.5.4.2 of the appendix show the corresponding distribution function information, with which these files were stored. Furthermore, it also shows which nodes contain the file[4]. These very tiny parallel files were mostly stored in one single I/O node, except in the case of the varstrip distribution function while serving the *Complete-1* pattern. The parallel file for this pattern using the varstrip distribution function was stored on each one of the eight I/O nodes as shown by the `pvfs2-viewdist` output in appendix A.5.3.4. This output shows that each node contained 701 bytes of the parallel file.

7.5.1 Patterns and Speedup Values

After demonstrating how the *mapping at first generation* partitioning function in combination with the two physical distributions serves the categorized *semi-random temporal parallel file I/O access patterns* from table 5.1, I proceeded to determine whether or not these categorized patterns are served with the same speedup.
In order to answer this question, I conducted `mcmc` analyses with each one of the patterns shown in figure 7.2 for only one single iteration. In this experiment I used an input matrix that generated an amount of data with *depth of I/O parallelism* $\delta_{\pi_{ioz}} = 2$ and the parallel file was stored with the simple_stripe distribution function the striping unit of 64 KB.
Figure 7.3 presents the computed speedup values according to equation 7.1. This figure shows different speedup values for the different patterns. This figure illustrates that from all patterns the *Complete-1* or π_1 pattern presents higher speedup than the other patterns, independently of the used physical distribution. The highest one was measured with the varstrip distribution function (3.86 in this case). The smallest speedup is presented by the *Complete+1* pattern (2.85 in this case). These speedup values can be explained by looking at the number of I/O swaps and the number of overloaded MPI processes (processes that write more runs as the others) for each pattern. The *Complete+1* pattern presents 2 I/O swaps and 1 overloaded process. These two factors make the computation for one iteration slower compared to the other patterns. The *Partial-1* pattern has two I/O swaps and one underloaded process which makes the computation not very slow as in the former case, but not as fast as in the case of π_1 pattern, in which neither I/O swaps nor an overloaded process exist.

[4]This information is determined through a small perl script that calls the `pvfs2-statfs` before and after each `mcmc` analysis. In order to associate the parallel file information (especially those provided as output of pvfs2 commands) to the corresponding experiment, I also include to this information the header of the corresponding performance tracing file.

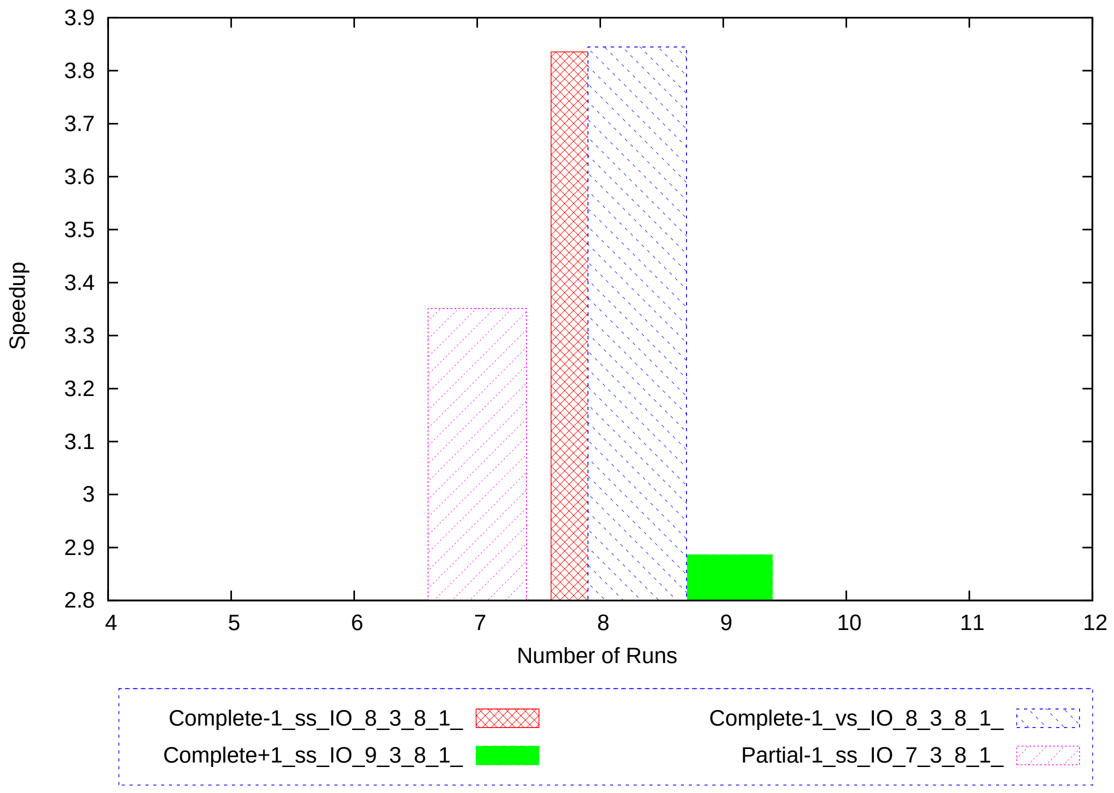

Figure 7.3: Computed speedup, equation 7.1, at generation one of three time parallel I/O access patterns supported by the *MFG* partitioning function and the two distribution functions. Speedups for an amount of data with *depth of I/O parallelism* $\delta_{\pi_{ioz}} = 2$ in each iteration. All patterns were served with the simple_stripe distribution function and its default striping unit. Only the parallel file for pattern *Complete-1* was also stored with the varstrip distribution function and a default *varstrip chunk size.* The labeling is to be interpreted as follows: The elements left from the element `IO` provide information on the type of *semi-random temporal parallel file I/O access pattern,* including the used distribution function (ss and vs stand for simple_stripe and varstrip, respectively). The numbers right from the element `IO` stand for the number of runs, chains, MPI processes, and iterations, respectively.

7.6 Speedup

Subsection 7.5.1 demonstrates that the π_1 pattern presents the highest speedup values among all patterns on the used I/O environment. It also shows that this pattern is supported by both distribution functions. The main objective of the set of experiments described in this section was to determine how the speedup for π_1 patterns changed in terms of the generated data in each iteration $data_{gen}$, the number of different physical I/O nodes, MPI processes, and runs.

7.6.1 Number of Runs

The objective of the experiments described in this section was to determine how the π_1 pattern's speedup varied with different number of runs.

For this experiment I selected an input matrix that generated an amount of data in each iteration $data_{gen}$ of 49076 bytes for 2 runs and 11.60 MB for 496 runs. I used as median value an amount of data that produced $\delta_{\pi_{ioz}} = 1$ and set as upper

limit a value of $data_{gen}$ that corresponded to $\delta_{\pi_{ioz}} = 5$. The input matrix had 1024 taxa with string length of 2086 as shown in table A.4. In this experiment I used 8 MPI processes for each measurement point, including those measurement points in which due to the generated amount $data_{gen}$ only 1, 2, 4 or 7 physical I/O nodes were accessed. Each run contained 4 Markov chains and I changed the number of runs according to the expression $nruns = 2^n, n = [1-9]$, so that the pattern was always π_1. Selecting $nruns = 1$ run was not possible, since 4 MPI processes would not get assigned any chain to compute by the load balancing mechanism.
The serial I/O version could not run $nruns = 512$ because the environment variable `_SC_OPEN_MAX` returned by `sysconf` was set to 1024 and limited the maximum number of files that a process can have open at any time, according to `sysconf (man 3)`. Thus, representing an upper limit for process 0 in the serial I/O version. In this case it needed to open 1025 files including the one with the input matrix. Due to this limitation a measurement point was set not in the above mentioned scheme at $nrun = 496$. By choosing this number of runs each MPI process and node computes exactly 248 markov chains. Thus, this measurement point was also a π_1 pattern.
Figure 7.4 depicts the computed speedup values after conducting the corresponding measurements for the serial and parallel I/O versions of the code. This figure shows the speedup presented by applying the simple_stripe distribution function with the default striping unit value. It mainly depicts the speedup's behavior in terms of the selected number of runs. For each of these speedup values it also shows the corresponding written data in each iteration as well as the number of I/O nodes that were accessed to store this data. Figure 7.5 also depicts the computed speedup values for the same set of measurement points when storing the parallel file with the varstrip distribution function.
In both figures the speedup increments while incrementing the number of accessed I/O nodes and runs. In order to be able to compare the number of accessed I/O nodes before reaching $\delta_{\pi_{iox}} = 100\%$, I used the simple_stripe distribution function without enabling the stuffing mechanism within the parallel file system, which writes small amounts of data into one single datafile in a round robin manner[5].
Before reaching $\delta_{\pi_{iox}} = 100\%$ the speedup is caused by two factors. The first one is the number of accessed physical I/O nodes and the second factor is the non-existence of the *many-to-one* communication in the parallel I/O version. These two factors produce a speedup up to 5.3 for $\delta_{\pi_{iox}} = 100\%$ when using all eight nodes of the I/O cluster. Once the maximum number of I/O nodes has been used, doubling the number of runs only contributes to a speedup of 0.7 in the best of the cases. By increasing the number of runs in our eight nodes I/O cluster, a speedup towards a maximum value of 7 is reached, independently of the applied distribution function.
Both figures show that for cases $\delta_{\pi_{iox}} < 100\%$ the different distribution functions store the same amount of data into different numbers of I/O nodes. The simple_stripe distribution function stores two runs (50 KB) into one single I/O node, since this is smaller than the default striping unit, whereas the varstrip distribution function stores it into two I/O nodes, since 2 MPI process request a *varstrip chunk size* of 25 KB.
The similar speedup values computed while using the two different distribution functions are caused by the sizes of the data generated in each iteration $data_{gen}$. For this particular experiment I selected the input matrix so that the values of $data_{gen}$ were multiple of the default striping unit. In the case of the varstrip distribution function

[5]This mechanism can be chosen in the file system configuration file as shown in appendix A.3.9.2. For further details see `../server/create.sm` in the code tree.

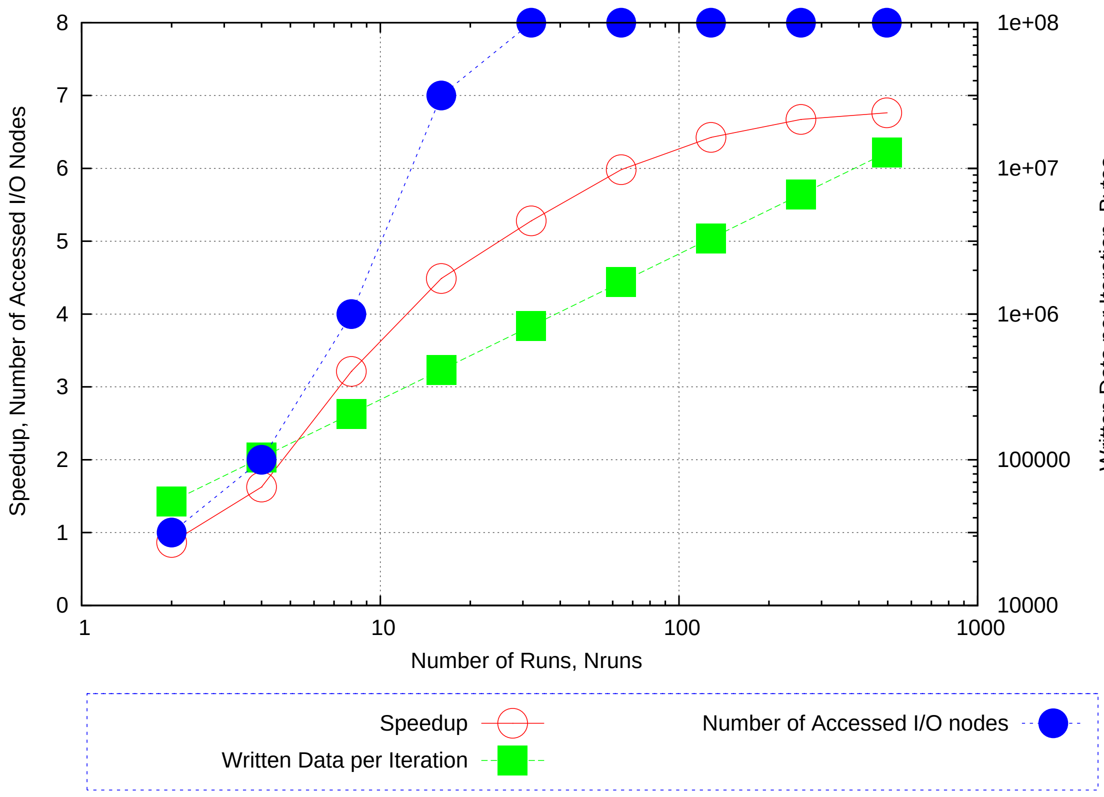

Figure 7.4: Speedup in terms of number of runs for small input matrices, $data_{gen}$ values around $\delta_{\pi_{ioz}} = 1$: This is the speedup presented by an input matrix that writes in each iteration 50 Kbytes for 2 runs up to 12 MB for 496 runs. This speedup is presented by applying the simple_stripe distribution function with the default value.

this selection is irrelevant, since it automatically adapts to the data available in the writing iteration. An example that shows the information on the parallel file when it was stored with the varstrip distribution function, is included under appendix A.6.2.1.

7.6.2 Number of MPI Processes and Physical I/O Nodes

An important factor that influences the speedup is the number of MPI processes with which a certain number of runs is computed on a given number of physical cluster nodes. In an I/O cluster with sufficient CPUs and secondary storage devices to provide parallelism, the selection of a relatively too small number of processes may not exploit parallelism, whereas a relatively too big number of processes may introduce unnecessary overhead. The objective of the set of experiments that I describe in this section was to determine the speedup for a given number of runs in terms of the number of MPI processes on a certain number of physical I/O nodes. Thus, providing information on the number of processes to maximize speedup values. Differently from the results already presented in subsection 7.6.1, in which the number of accessed physical I/O nodes was conditioned only by $data_{gen}$, in the set of experiments that I describe in this section the number of physical I/O nodes to be used was explicitly set with `mpdboot -totalnum=<8 | 4 | 2>`. Furthermore, the chosen number of MPI processes created π_1 patterns on these nodes.

7.6.2.1 Eight Physical I/O Nodes

The experiments described in this and the next two subsections were served with the simple_stripe distribution function and its striping unit default value. In the

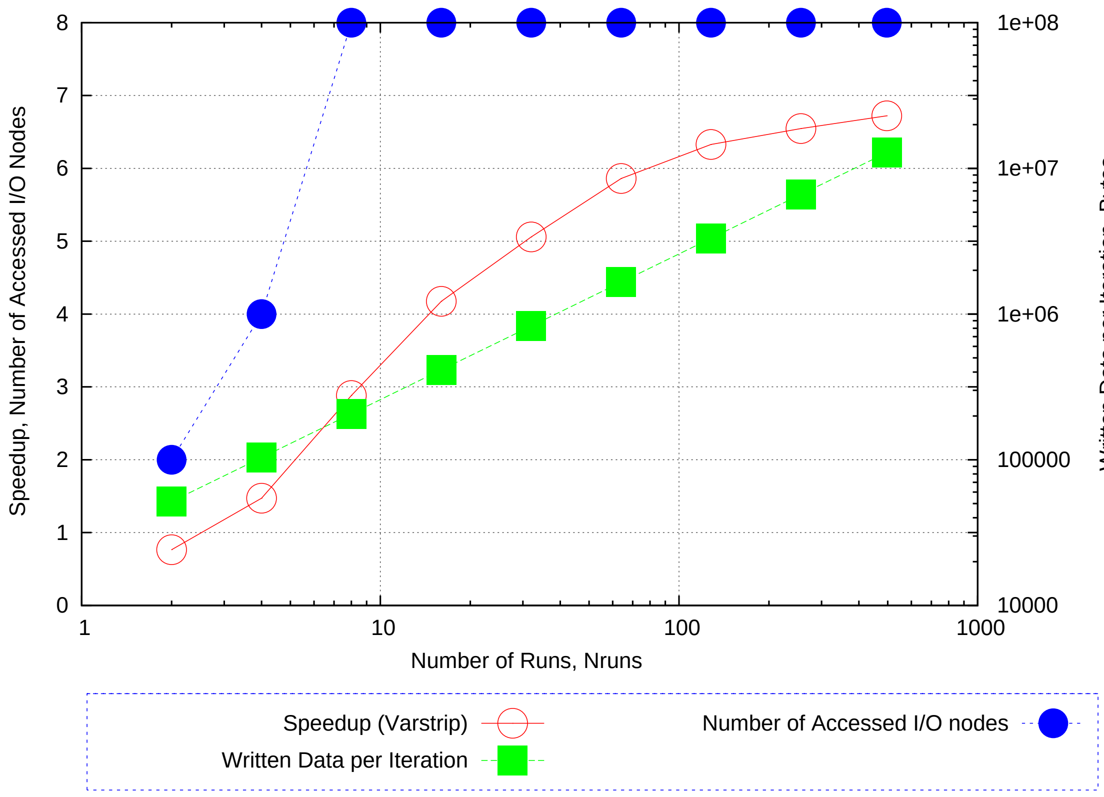

Figure 7.5: Speedup in terms of number of runs for small input matrices, $data_{gen}$ values around $\delta_{\pi_{ioz}} = 1$, varstrip: This is the speedup presented by an input matrix that writes in each iteration 50 Kbytes for 2 runs up to 12 MB for 496 runs. This speedup is presented by applying the varstrip distribution function with its default *varstrip chunk size* value.

case of 8 physical I/O nodes I selected an input matrix that caused a *depth of I/O parallelism* between one and four for 32 and 256 runs respectively. I executed 10 iterations each one of these number of runs with 8, 16, 32, and 64 MPI processes. Figure 7.6 presents the computed speedup values at iteration 10
This figure shows that given a *depth of I/O parallelism* greater than one, the behavior of speedup in terms of the number of MPI processes, presents a semi-parabolic behavior. At lower number of processes the speedup increments while incrementing the number of processes up to a value, the *process saturation number*, where it reaches its maximum value and then decreases while still incrementing the number of processes. In this case speedup reached its maximum when using 16 MPI processes independently of the selected number of runs. In this set the biggest speedup improvement was measured for 256 runs that changed about 5.4 by selecting 16 instead of 8 processes. On the other hand, selecting a too high number of MPI processes, even with operating system swapping not activated as in this case, the speedup can drop to about 56% compared to the best value, as presented in figure 7.6 for the case of 32 runs.

7.6.2.2 Two and Four Physical I/O Nodes

The objective of this experiment was to determine whether or not the speedup's semi-parabolic behavior with corresponding *process saturation number* prevailed when using different number of physical I/O nodes.

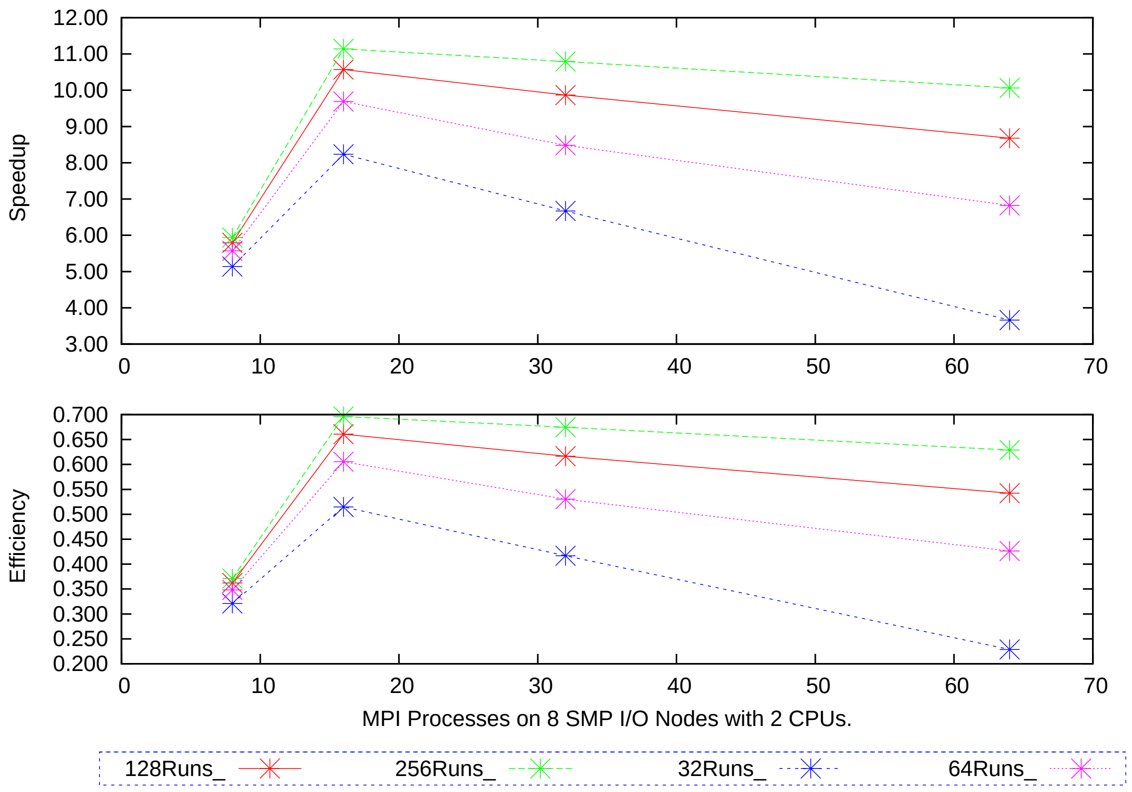

Figure 7.6: Speedup and efficiency: Speedup and efficiency, acccording to their definitions under subsection 7.3.1, in terms of MPI process numbers on 8 SMP physical I/O nodes provided with 2 CPUs. These values correspond to an amount of data generated in each generation with *depth of I/O parallelism* between one and four for 32 and 256 runs, respectively.

I conducted this experiment with the same input matrix and conditions as the one already discussed in subsection 7.6.2.1, except that I used 1 - 64 runs on 2 physical nodes. Figure 7.7 shows the speedup and efficiency computed values. Using the same input matrix as in the case with two physical nodes[6], I executed for one generation 32, 64, and 128 runs on 4 physical I/O nodes. These runs were executed with 4, 8, 16, 32, and 64 MPI processes. The used input matrix generated in each iteration amounts of data $data_{gen}$ with *depth of I/O parallelism* values of 2, 4, and 8 for the 3 mentioned number of runs, respectively. Figure 7.8 depicts the computed speedup values. On the three different numbers of physical I/O nodes, the cases of 32 and 64 runs generate exactly the same amount of data in each iteration. Figures 7.6, 7.7, and 7.8 show for these two numbers of runs that depending only on the chosen number of physical I/O nodes different speedup values were computed. In the case of 64 runs[7] at *process saturation number* I computed speedup values of 9.6, 6.6, and 3.6 for 8, 4, and 2 physical SMP I/O nodes provided with 2 CPUs, respectively. The corresponding efficiency, equation 7.2, values were 0.6, 0.85, and 0.9. The highest speedup of 11.3 with an efficiency of 0.7 was computed at *process saturation number* for 256 runs on 8 physical nodes. Figures 7.6, 7.7, and 7.8 illustrate how the *process saturation number* shifts depending on the chosen number of physical nodes. It presents values of 4, 8, and 16 for 2, 4, and 8 physical nodes. These values

[6]Synthetical input matrix with 2664 taxa of 10 characters.
[7]Similar behavior is presented by 32 runs

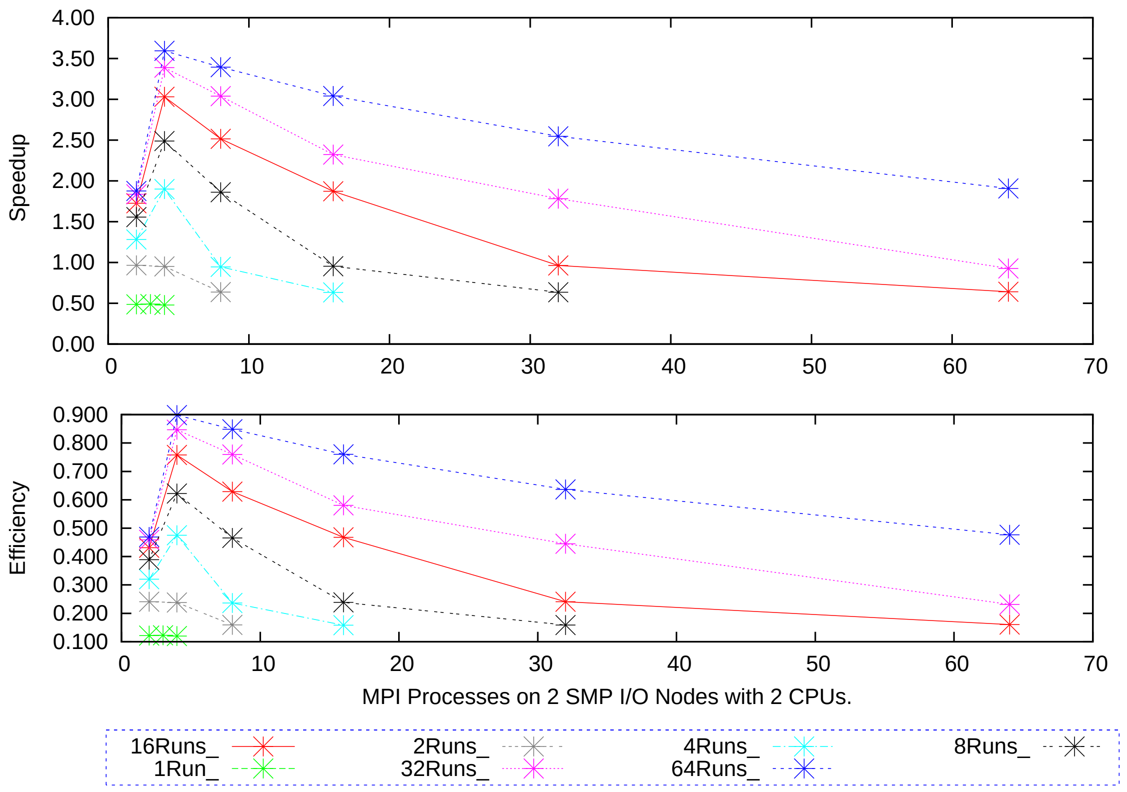

Figure 7.7: Speedup and efficiency: Speedup and efficiency, acccording to their definitions under subsection 7.3.1, in terms of MPI process in terms of MPI process numbers on 2 SMP physical I/O nodes provided with 2 CPUs. These values correspond to an amount of data generated in each generation with *depth of I/O parallelism* between four and sixteen for 1 and 64 runs, respectively.

correspond to one MPI process per CPU on the used SMP nodes[8]. The *process saturation number* was also conditioned by the assignment of runs to processes. In these cases no interprocess communication takes place to compute runs.
These speedup curves show that the same speedup value can be obtained by selecting two different numbers of MPI processes. Nevertheless, only the speedup corresponding to the smallest number of MPI processes represents for the user also small execution time. Thus, in this work I call it the *smallest-process-overhead speedup*. It is obtained while selecting a number of MPI processes less or equal than the *process saturation number*.

7.7 Speedup and Distribution Functions

The objective of the set of experiments described in this section was to determine how the speedup of a π_1 pattern served by the *MFG* partitioning function could be changed by using different striping unit and *varstrip chunk size* values.

7.7.1 *Optimal striping unit*

Provided an amount of data to be written to all I/O nodes in each generation $data_{gen}$, a number of I/O nodes N_{IO_t}, and $\delta_{\pi_{iox}} = 100\%$, theoretically a striping unit can be

[8]In the set of conducted experiments there was not notable bottleneck effects due to the fact that 2 CPUs on one single SMP node were writing their data into one single secondary storage device.

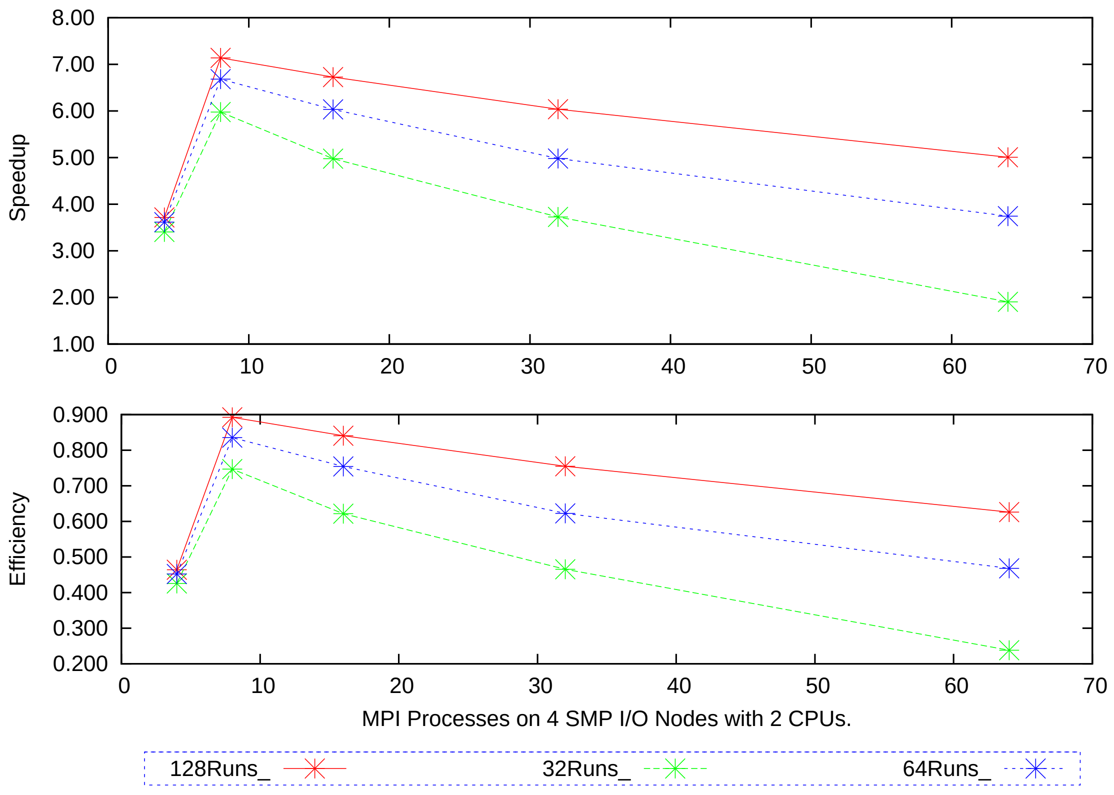

Figure 7.8: Speedup and efficiency: Speedup and efficiency, acccording to their definitions under subsection 7.3.1, in terms of MPI process numbers on 4 SMP physical I/O nodes provided with 2 CPUs. These values correspond to an amount of data generated in each generation with *depth of I/O parallelism* between two and eight for 64 and 128 runs, respectively.

chosen between a minimum $su_{min} = 1$ and a maximum value su_{max} as expressed in equation 7.4.

$$su_{max} = \frac{data_{gen}}{N_{IO_t}} \tag{7.4}$$

This section describes the experiment to determine which striping unit values between these two limits reduce or increment execution time. Selecting a striping unit value greater than su_{max} translates into $\delta_{\pi_{iox}} < 100\%$. This section also presents a complementary experiment that independently of any particular application shows the execution time behavior when using striping unit values that yield $\delta_{\pi_{iox}} < 100\%$. I conducted the first experiment using four values for $data_{gen}$ that presented $\delta_{\pi_{ioz}} = 1, 2, 4, 8$ with a 64 KB reference, respectively. This means, for example, that for $\delta_{\pi_{ioz}} = 8$ an amount of data $data_{gen} = 4$ MB was written in each iteration into the parallel file. Thus, 512 KB was written on each node in each generation. For this set of data, I measured the execution time for different striping unit values between 1 and the corresponding su_{max} values.
Figure 7.9 shows corresponding time difference percentages for these execution times according to equation 7.3. In this equation $Time_{ref}$ was assigned the measured values corresponding to $\delta_{\pi_{ioz}} = 1$ for each one of the generated amounts of data $data_{gen}$. Figure 7.9 depicts the *time difference percentage* reductions and increments due to different striping unit values in the upper and lower half, respectively. It shows that $su = 1$ induced overhead so that the execution time values were 2 -10% slower than

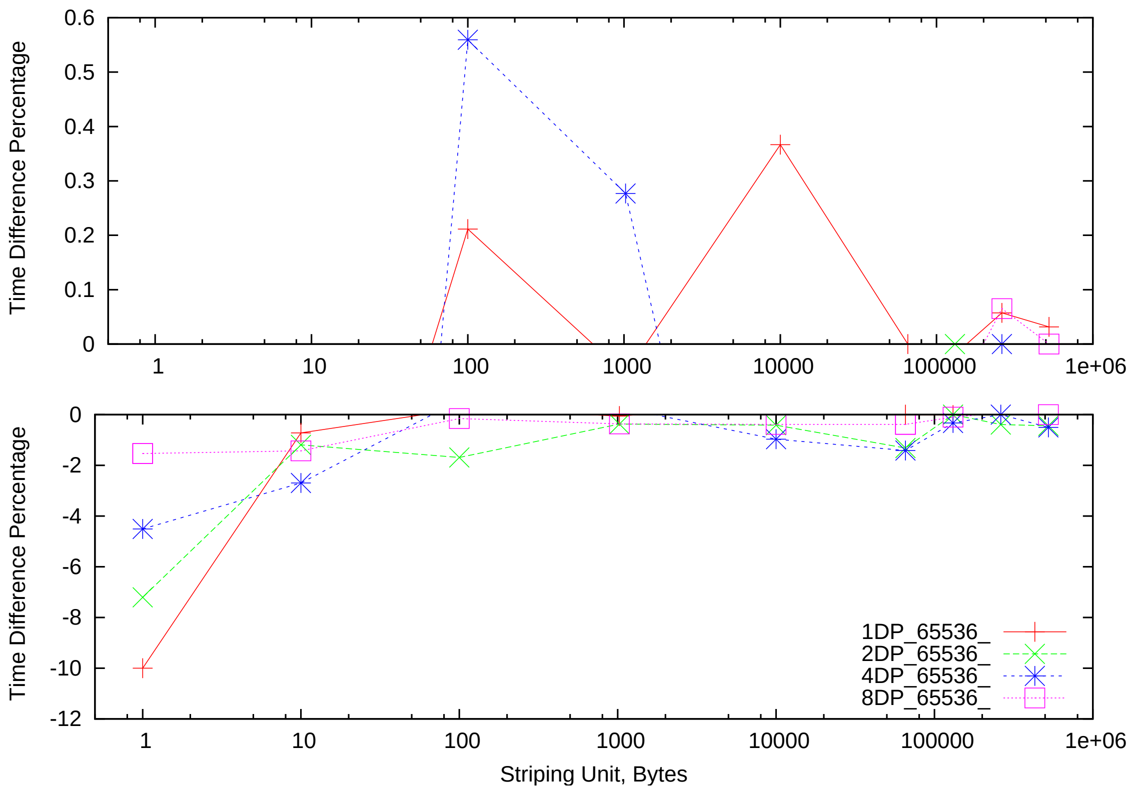

Figure 7.9: The *time difference percentages* in terms of striping unit values for a π_1 pattern: These percentages are computed according to equation 7.3 taking as $Time_{ref}$ the execution time for $\delta_{\pi_{iox}} = 100\%$, $\delta_{\pi_{ioz}} = 1$ for cases with $\delta_{\pi_{ioz}} = 1, 2, 4, 8$ using as reference 65536 bytes. Positive values translate into execution time reductions. The following example presents the labeling interpretation: The label 8DP_65536 stands for an amount of data with *depth of I/O parallelism* (DP) $\delta_{\pi_{ioz}} = 8$ while using a reference of 65536 bytes).

the corresponding su_{max} time values. Differently from the time increments a maximum time reduction of 0.6% was measured.
Figure 7.9 also shows that the *time difference percentage* also varies depending on the amount of data generated in each iteration $data_{gen}$. Smaller amount of data are more sensitive to different striping unit values. The *time difference percentage* values in terms of striping units shows an asymptotic behavior towards the su_{max} value. Striping unit values between 100 bytes and the su_{max} present smaller execution *time difference percentages.*
Figure 7.9 shows that setting $\delta_{\pi_{iox}} < 100\%$, by choosing a striping unit value bigger than the data written by each node in each iteration, incremented execution time, except in the case for $\delta_{\pi_{ioz}} = 1$ that presented a maximum time reduction of 0.05%. Nevertheless, this time reduction for 10 iterations increased while increasing the number of iterations as shown in figure 7.10. This figure depicts the measured values of a complementary experiment that I conducted to determine the execution time behavior in terms of striping unit values that are bigger than the main memory buffer's size independently of the used application. For this experiment I wrote a small program, which did not require any input matrix, to write out 300 iterations 10000 bytes from each one of the 8 nodes. For these operations I used striping unit values of 10000, 20000, and 40000 bytes. Due to the small generated amount

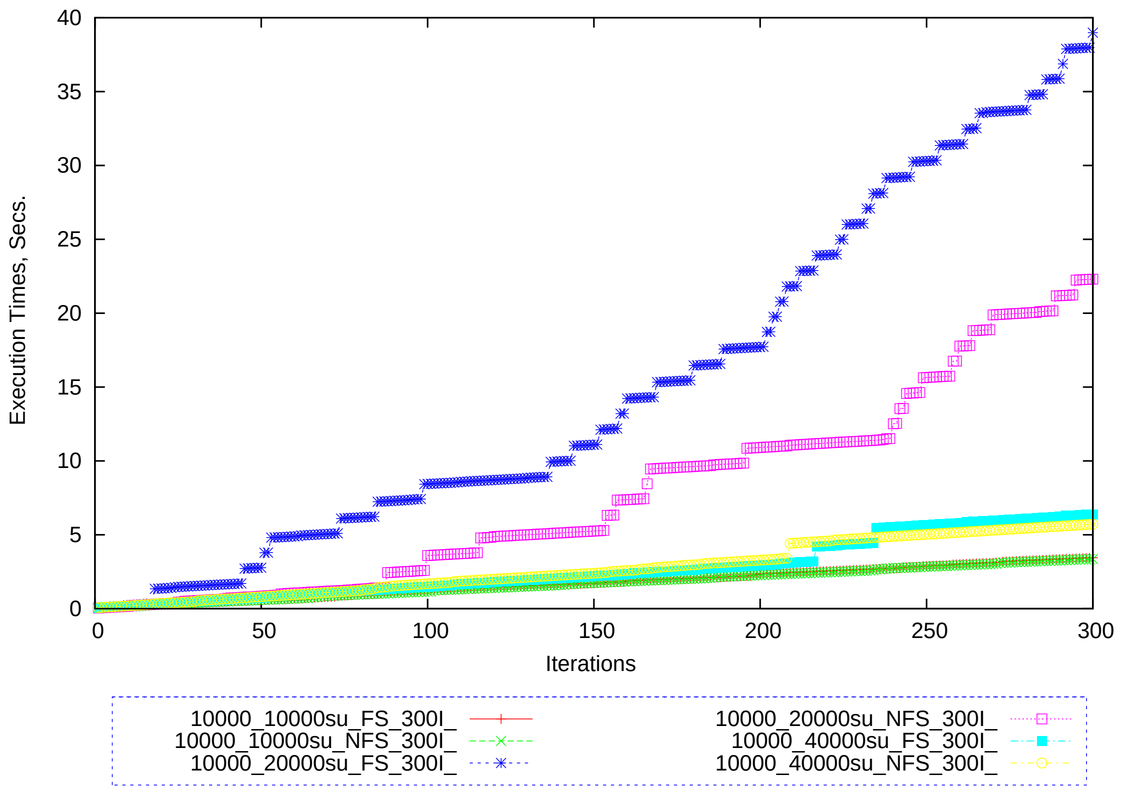

Figure 7.10: Execution time increments for $\delta_{\pi_{iox}} < 100\%$ in terms of number of iterations: Execution time measured by selecting a striping unit value multiple of the main memory buffer's size. The labeling is to be interpreted as follows: the leftmost number is the main memory buffer size in bytes, the middle one is the striping unit value and the right most the number of iterations. The labels (N)FS stand for the (no) application of the file stuffing mechanism within the parallel file system.

of data, I conducted the measurements with and without the stuffing mechanism within the parallel file system. Figure 7.10 shows that using values greater than the buffer's size, $\delta_{\pi_{iox}} < 100\%$, produced values greater than the one corresponding to $\delta_{\pi_{iox}} = 100\%$ and $\delta_{\pi_{ioz}} = 1$. This is especially notorious for higher number of iterations. Furthermore, the execution time did not remain constant across iterations, since different I/O node sets were accessed in each iteration.

7.7.2 Varstrip, Maximum and *Optimal Varstrip Chunk Size*

In the varstrip distribution function each computing process writes a chunk size of data on a given I/O node. In this work I call this amount of data the *varstrip chunk size* and I consider the buffer size after one generation as its default value. In general, this default size takes different values on the different I/O nodes depending on the degree of load balancing. Since in this work I propose to apply the varstrip function only to patterns of type π_1, the *varstrip chunk size* has the same value across all computing nodes.
The purpose of the set of experiments described in this section was to evaluate how the application of parallel line buffering to set the *varstrip chunk size* value affected execution time.

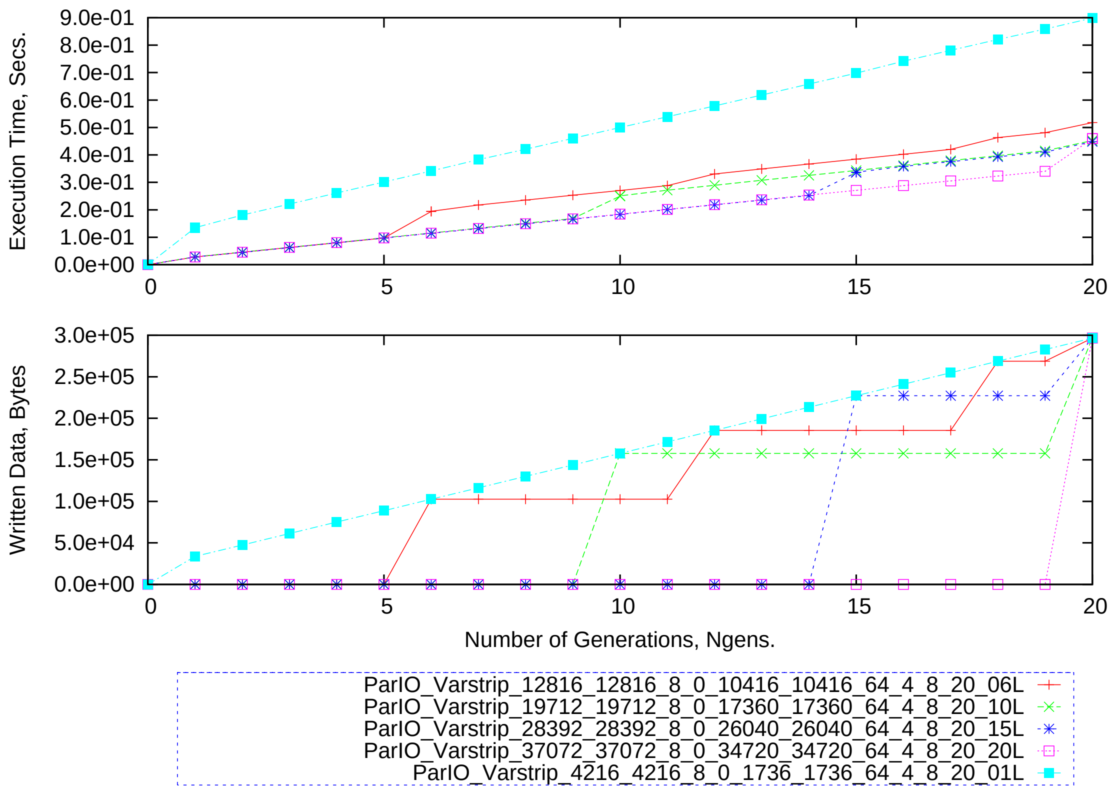

Figure 7.11: Execution time and written data into the secondary storage while keeping 1, 6, 10, 15, 20 computed results in main memory and using this value for the *varstrip chunk size* for the input matrix with 8 taxa with lengths of 2086 characters. Detailed information on the meaning of the first 12 labeling elements can be found under subsection A.4.1.2 in the appendix. The last element stands for the number of iterations to be kept in memory with the *parallel line buffering* mechanism.

7.7.2.1 *Parallel Line Buffering*

The objective of the first experiment was to evaluate the correct functionality of *parallel line buffering*. The objective was to show how *parallel line buffering* influences execution time when conducting `mcmc` analyses of matrices with a small (8) and a medium (1024) number of taxa.
One example of the used `mcmc` script is included under appendix A.3.7.2 for further information. Figure 7.11 and 7.12 depict the measured values. Figure 7.11 shows how different *varstrip chunk size* values influence execution time for the input matrix with 8 taxa and 2086 characters. I executed this experiment with 64 runs each having 4 chains on 8 MPI processes during 20 generations. In this setup I applied *parallel line buffering* to keep in main memory 1, 6, 10, 15 and 20 generations' results before writing them into the parallel file. In figure 7.11 these values are shown as the rightmost element of the labels in terms of lines. The maximum and minimum byte sizes for these lines across all processes are shown as the 7th and 8th elements of the labels, correspondingly. Since this setup was load balanced there was only one value for the *varstrip chunk size* across all MPI processes.
Figure 7.11 shows the measured execution time as well as the amount of data that was written into the parallel file by all MPI processes. This figure shows that setting the *varstrip chunk size* to a value multiple of default value influenced the execution

time. In this particular case it reduced it. It also shows that keeping different number of results in main memory influences time execution in different proportions. For instance, buffering 6 results (10416 bytes on each computing node) contributes to an execution time's reduction of 42% compared to retaining only 1 result (1736 bytes). On the other hand, the effect of keeping the result of 20 generations (34720 bytes) yielded a time reduction of 50%. In this set of measurements I computed a maximum of time reduction percentage of 50% compared to the execution time of the default *varstrip chunk size* value after 20 generations.
Depending on the input matrix size, setting the *varstrip chunk size* to values greater than the default value does not always yield a time reduction. Figure 7.12 visualizes the measured execution times for the input matrix with 1024 number of taxa. In this particular case the measured minimum execution time values, best performance, corresponded to the default value. The lower part of the picture shows the total amount of data written into the parallel file. After the first generation the processes wrote a total of 3456832 bytes (3.29 MB) into the file. From the second generation on the processes wrote a total of 1570304 (1.49 MB) after each generation. Depending on the required analyses's precisions this yields an amount of data of 14.62 GB after 10000 generations, 0.14 TB after 100000, and 1.42 TB after one million generations.

7.7.2.2 The Maximum Parallel Line Buffering Factor λ_{Max}

The results presented in figures 7.11 and 7.12 show two extrem cases of applying parallel line buffering to set the *varstrip chunk size* to a value different from the default one. Figure 7.11 shows a case, in which parallel line buffering only reduced execution times in different proportions. On the other hand, figure 7.12 shows a case, in which parallel line buffering only increased the execution times in different proportions. The only difference between these two setups was the input matrix size, which defined the amount of data in main memory in each generation.
The objective of the experiment described in this section was to determine the cause of time reduction while increasing the value of λ. Thus, this experiment consisted in measuring the execution time while keeping relatively big number of lines in main memory (8000L and 9000L) without writing them into secondary storage. I conducted the experiment with pattern π_1 on the I/O environment described under section 7.2 using an input matrix with 1024 taxa (2.1MB). Figure 7.15 shows the measured execution time values. In figures 7.15, 7.14, and 7.13 the first seven elements in the labeling have the following meanings: type of I/O interface, type of distribution function, number of runs, number of chains, MPI processes, total number of iterations, λ value (in number of lines). The string M_1024 stands for the number of taxa in the input matrix having a string length of 2086 characters and the rest of the string provides information on the different types of virtual memory monitored on one of the cluster nodes (`node01`). Figure 7.15 shows that execution with $\lambda = 8000L$ was faster than with $\lambda = 9000L$. This difference in execution time was especially notorious for small numbers of generations. For one generation it was about one order of magnitude. Since the program did not have I/O activities, only the values for the virtual memory usage under `/proc/meminfo` on all 8 I/O nodes during the execution time were monitored. Figures 7.13 and 7.14 visualize the measured values on node01[9]. In order to have similar initial conditions, the operating system swapping was turned off and on before each analysis.

[9]Except when explicitly noted the virtual memory information corresponds to the monitored memory values under `/proc/meminfo`.

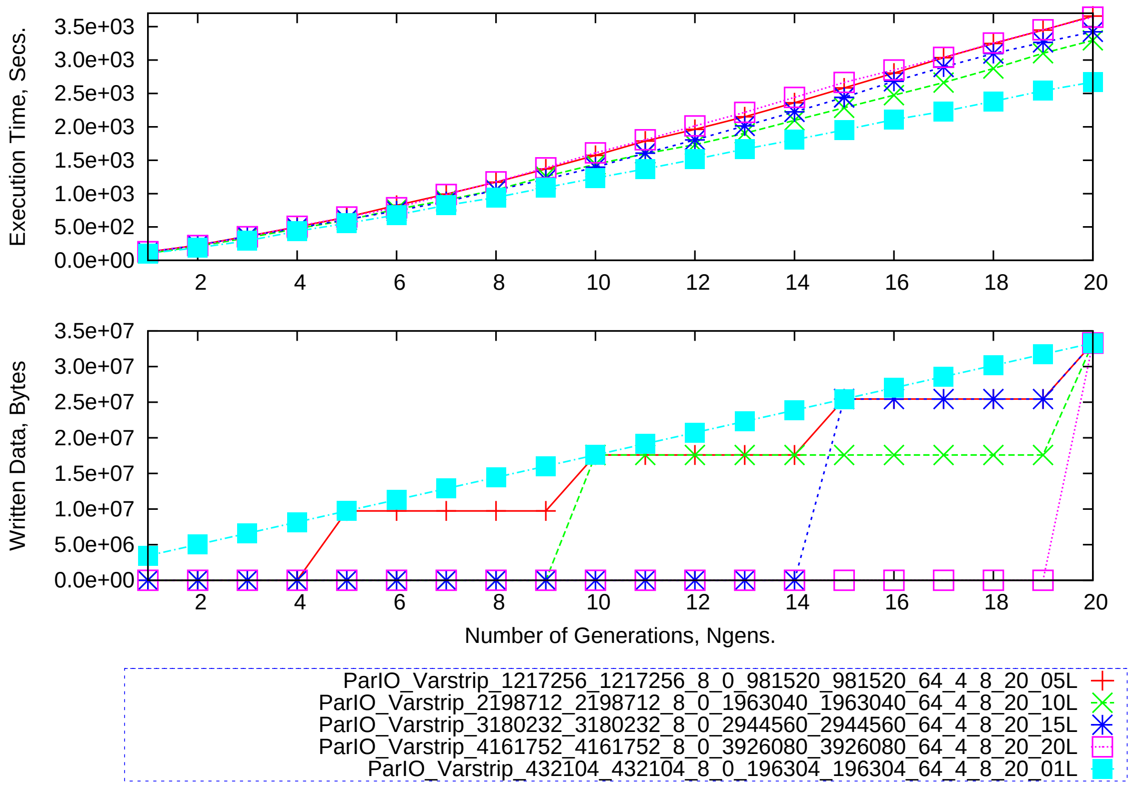

Figure 7.12: Execution time and written data into the secondary storage while keeping 1, 5, 10, 15, 20 computed results in main memory and using these values as the *varstrip chunk size* for the input matrix with 1024 taxa with lengths of 2086 characters. Detailed information on the meaning of the first 12 labeling elements can be found under subsection A.4.1.2 in the appendix. The last element stands for the number of iterations to be kept in memory with the *parallel line buffering* mechanism.

These two figures present the virtual memory usage in terms of the number of measurement points during the whole execution time. These values were polled from each node to the front end node of the testbed cluster[10]. In figure 7.13 the execution time began when the value of the available free memory dropped and finished when the memory returned to its original value. This figure also shows that during the whole analysis the operating system swapping partition was used and this usage did not remain constant. It also shows that the amount of memory used as buffer increased due to λ, whereas the memory used for caching remained constant.
Figure 7.14 shows the virtual memory usage for $\lambda = 8000L$. It clearly shows that there was no usage of the operating system swapping partition whatsoever. These two figures show that setting λ to a relatively big value decreases the amount of free memory below the value that activates the operating system swapping and thus decreases execution time. In this work I propose in equation 6.1 an expression for determining the upper limit λ_{Max} so that setting $1 < \lambda < (75\%) * \lambda_{Max}$ does not activate operating system swapping. Nevertheless, if operating system swapping is activated during the first iteration, $\lambda_{max} = \lambda = 1$ should be applied.

[10]There is no one to one correspondence between the application's iterations and the virtual memory measurement polling frequency.

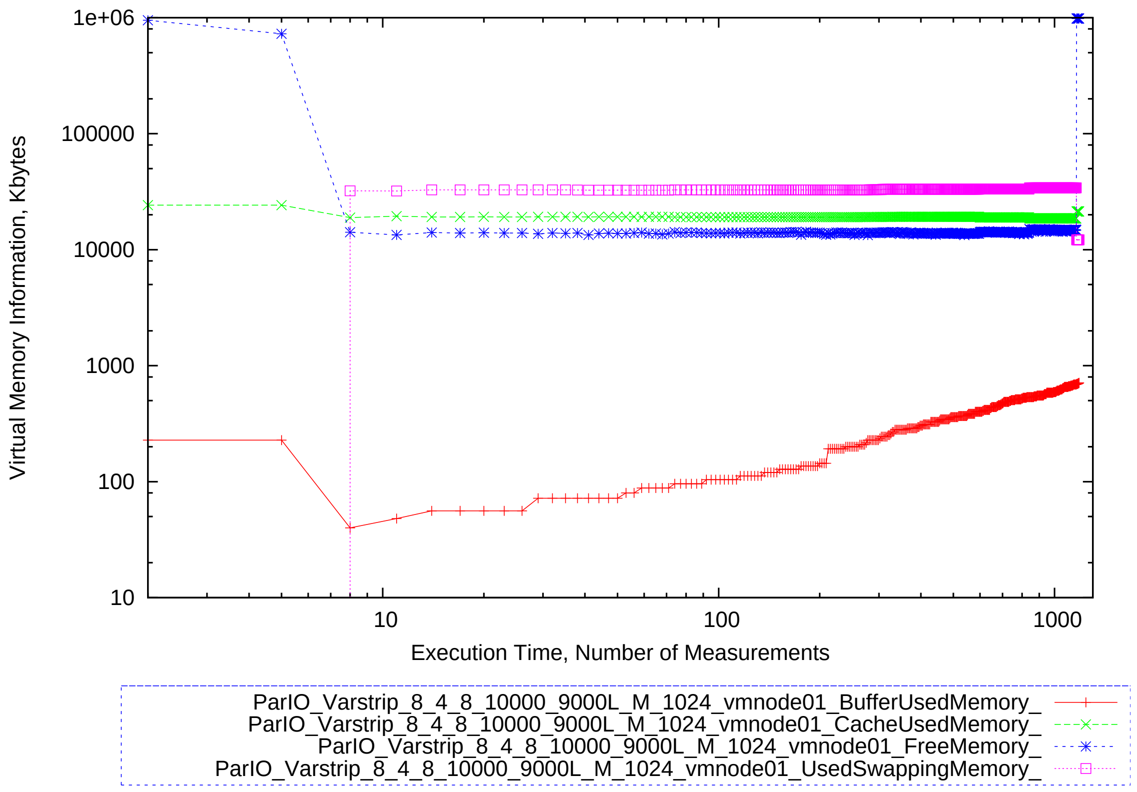

Figure 7.13: Virtual memory information on `node01`: These values were measured over the whole execution time for $\lambda = 9000L$. Operating system swapping was activated.

7.7.2.3 The Optimal Value for λ_{opt}

For a total amount of generated data over the whole execution time $data_{total}$, the value for the *varstrip chunk size* can be explicitly set between a minimum and a maximum value as expressed in equation 7.5 and 7.6, respectively. In these expressions $data_{gen}$ stands for the amount of data generated in each iteration, $wnum$ represents the total number of write operations during the whole execution time, and N_{IO_t} stands for the total number of accessed I/O cluster nodes.

$$vscs_{min} = \frac{data_{gen}}{N_{IO_t}}, \lambda = 1 \tag{7.5}$$

$$vscs_{max} = \frac{data_{total}}{N_{IO_t}}, \lambda = wnum \tag{7.6}$$

Concerning performance, one upper limiting factor constitutes the activation of the operating system swapping as I have experimentally shown in subsection 7.7.2.2 and theoretically describe in equation 6.1. The objective of this experiment was to study the execution time behavior in terms of λ values that did not activate operating system swapping, $1 < \lambda < \lambda_{max}$. Specifically, the objective was to determine under which conditions the execution time was reduced by making one single write operation with λ_{opt} compared to making many write operations with λ, for $\lambda_{max} > \lambda_{opt} > \lambda > 1$.
I conducted this experiment using $\delta_{\pi_{ioz}} = 1, 2, 4, 8$ as I have already described in subsection 7.7.1 for the simple_stripe distribution function. For each one of the $\delta_{\pi_{ioz}}$

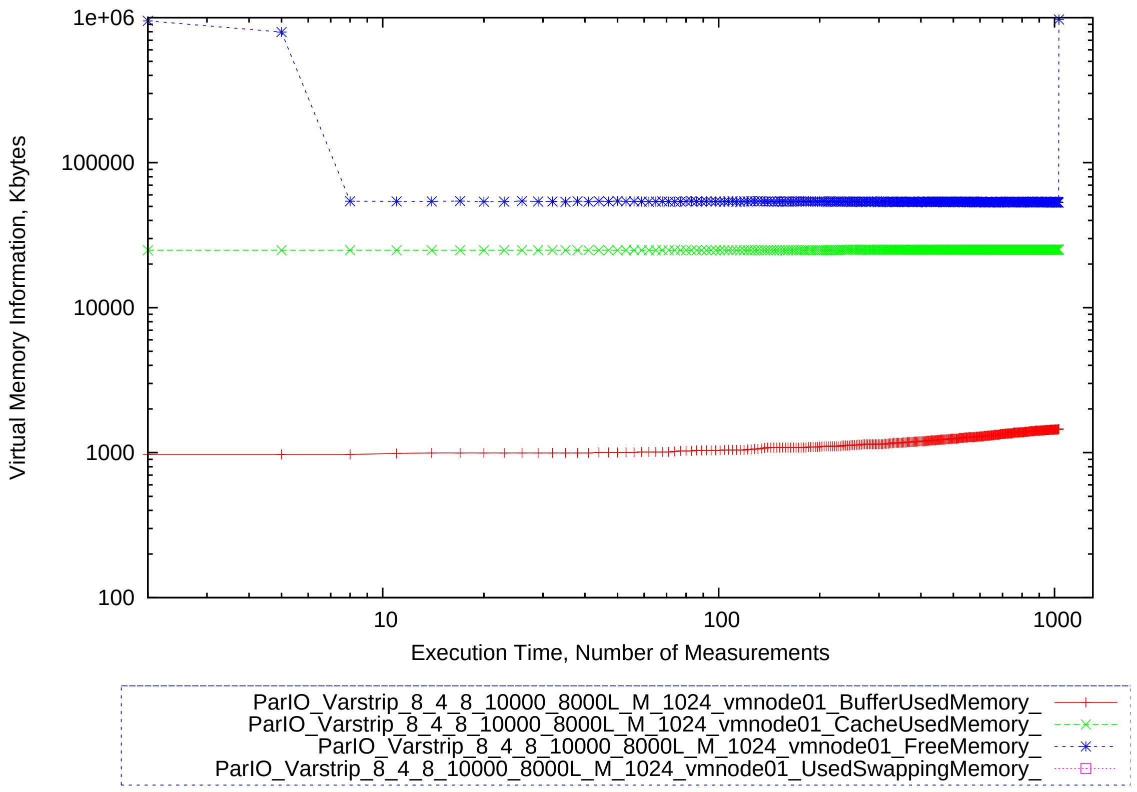

Figure 7.14: Virtual memory information on `node01`: These values were measured over the whole execution time for $\lambda = 8000L$. Operating system swapping was not activated.

values I set $\lambda = 1, 2, 4, 8, 16$ for a total of 16 iterations.
Figure 7.16 shows the corresponding time difference percentages computed according to equation 7.3 for the measured execution times. In equation 7.3 I used as $Time_{ref}$ the execution time measured for cases with $\lambda = 1$. This figure illustrates the *time difference percentage* reductions and increments due to different *varstrip chunk size* values. The execution time values used for this graph correspond to the total execution time after 16 iterations. This figure illustrates that smaller amount of data, $\delta_{\pi_{ioz}} \leq 4$, exploit the strengths of $\lambda > 1$. In the case of $\delta_{\pi_{ioz}} = 1$, for example, $\lambda = 16$ a time reduction of up to 2.5% was experimented. On the other hand, bigger amount of data, $\delta_{\pi_{ioz}} > 4$, present time execution increment when using $\lambda > 1$ (1 MB or greater amounts of data on each node in each generation). A maximum value for a time increment of -2.5% for $\delta_{\pi_{ioz}} = 8$ was computed. Thus, in such cases $\lambda = 1$ shall be applied. This figure also shows that smaller values of $\delta_{\pi_{ioz}}$ are more sensitive to λ values. Figure 7.17 also presents the amount of written data into the parallel file corresponding to the execution times used in figure 7.16. This figure depicts the total written data into the parallel file in terms of the number iterations, λ values, and the depth of I/O parallelism $\delta_{\pi_{ioz}}$ with a 64KB reference. Figure 7.17 shows that in the case of $\delta_{\pi_{ioz}} = 8$ a total of 70MB (72573136 bytes) was written into the parallel file after 16 iterations which corresponds to 0.4 Terabytes for an analysis with 100000 number of iterations.

7.8 Speedup and Iterations

Theoretically after a code has been parallelized, a steady speedup value should be expected independently of the iteration number at which this speedup is measured.

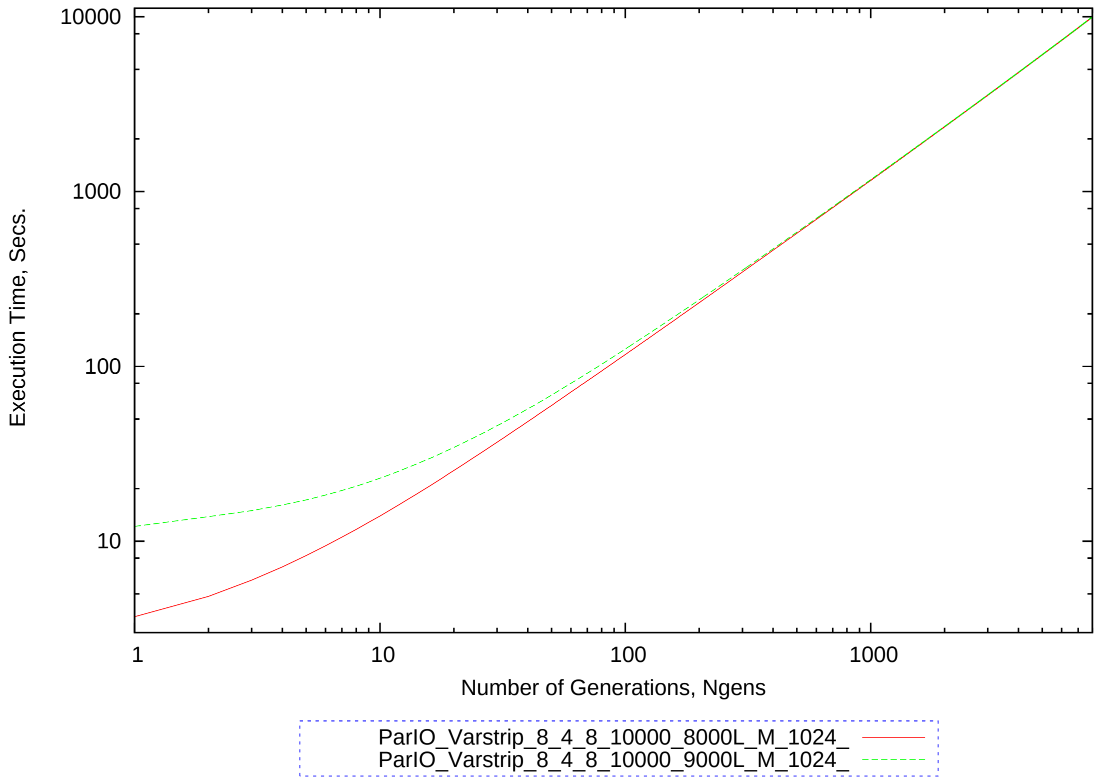

Figure 7.15: Execution times for the first 8000 generations with $\lambda = 9000L$ and $\lambda = 8000L$ without I/O operations. Detailed information on the labeling can be found under subsection A.4.1.2 in the appendix.

In the particular case of the I/O parallelization that I propose for this application, this is not the case due to the following factors: different file opening operations during the first iteration, overhead due to the *mapping at first generation* partitioning function for the parallel I/O variant during iteration one, and the amount of generated data during the first iteration is bigger than the rest of iterations, since it includes header information. Even though, this last factor exists in both code variants, it represents different I/O overhead for both codes.
Mainly due to the above enumerated factors the execution times vary across iterations of the same code variant[11]. Thus, a non-constant speedup value across iterations is to be expected. The objective of this experiment was to determine the speedup in terms of iterations.
On the already discussed cluster testbed I generated π_1 patterns using 8 MPI processes with 8 runs and 64 runs. Using input matrices that generated *depth of I/O parallelism* values between 0.5 and 32, I ran analysis for 8 iterations as depicted in figure 7.18. For the parallel I/O measurements the *varstrip chunk size* was set to $\lambda = 1$. Figure 7.18 illustrates different speedup values for the same amount of data generated in each iteration. This speedup changed in terms of the number of runs and the *depth of I/O parallelism* $\delta_{\pi_{ioz}}$. It shows that the *mapping at first generation* in combination with a distribution function, varstrip in this case, is appropriate to serve relatively big number of runs, 64 in this case, with bigger *depth of I/O parallelism* values, 16. Under these conditions a speedup of 7 was computed.

[11]For this discussion I consider that no notorius difference is induced by the application semi-randomness in any code variant.

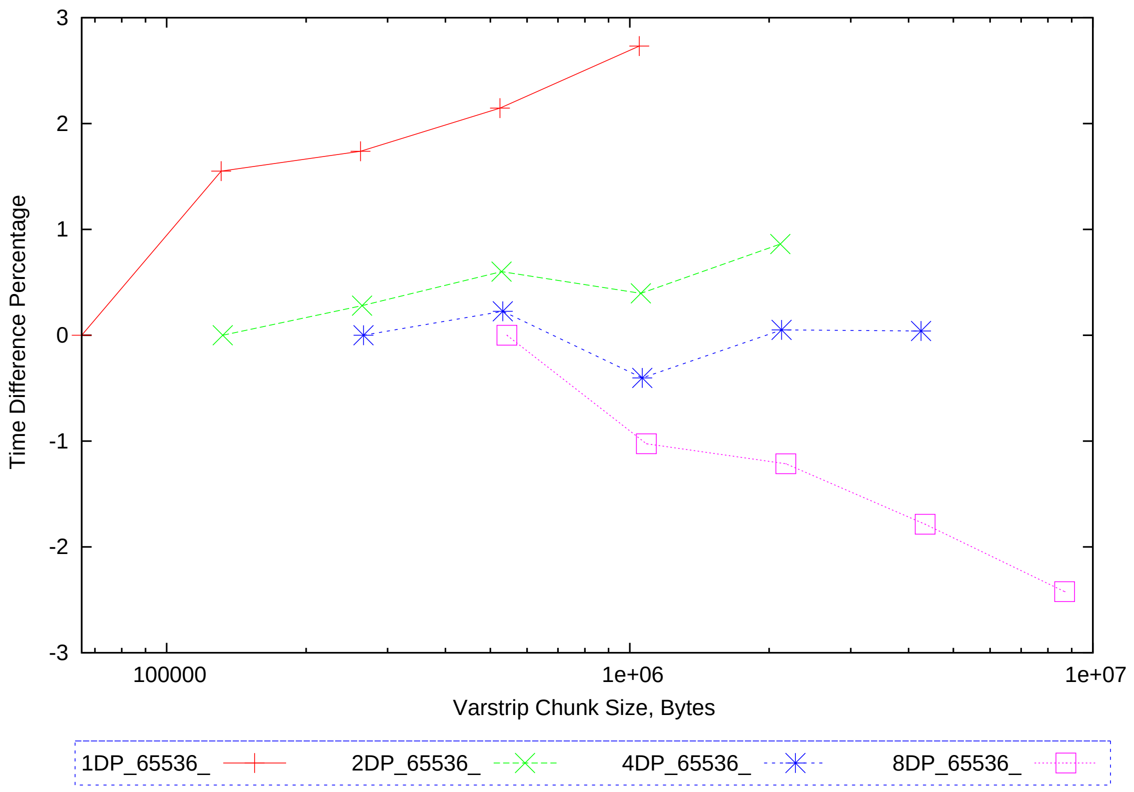

Figure 7.16: The *time difference percentages* in terms of *varstrip chunk size* values $\lambda = 1, 2, 4, 8, 16$ for a π_1 pattern: These percentages are computed according to equation 7.3 between the total execution time and $Time_{ref}$ corresponding to $\lambda = 1$ for each of the four cases $\delta_{\pi_{ioz}} = 1, 2, 4, 8$ with a 64KB reference. Positive values translate into execution time reductions.

With both number of runs the speedup in terms of iterations presents an asymptotic behavior. Nevertheless, for 8 runs the speedup increases towards a maximum value, whereas in the case for 64 it decreases towards a minimum value with the number of iterations. In the case of 64 runs and $\delta_{\pi_{ioz}} = 8, 16$ the speedup remains constant across all iterations.
With 8 runs and $\delta_{\pi_{ioz}} = 1$ there is almost no difference between the data written into secondary storage by both codes. Thus, not only the speedup has a small value, 4, but differences across iterations are not very relevant. In the case of $\delta_{\pi_{ioz}} = 32$ and $\lambda = 1$ the serial I/O code has relatively less overhead during iteration 1. Thus, the computed speedup value is 3.8. Nevertheless, for iterations greater than 1 the parallel I/O overhead due to *mapping at first generation* does not exist anymore and speedup values start to increase towards a maximum value, 5 in this case.
With 64 runs and $\delta_{\pi_{ioz}} = 0.5$ even though the amount of data is small and the *mapping at first generation* overhead is present during the first iteration, these two factors are not very significant compared to the overhead due to the number of runs. Thus, under these circumstances the parallel I/O variant is faster than the serial I/O during iteration one and the speedup has the highest value of 5 and it decreases towards a miminum value of 3.5 after 8 iterations. The non-stable effects of the overhead values during the first iteration become negligible while the amount of generated data in each iteration becomes bigger as in the case of $\delta_{\pi_{ioz}} = 8, 16$,

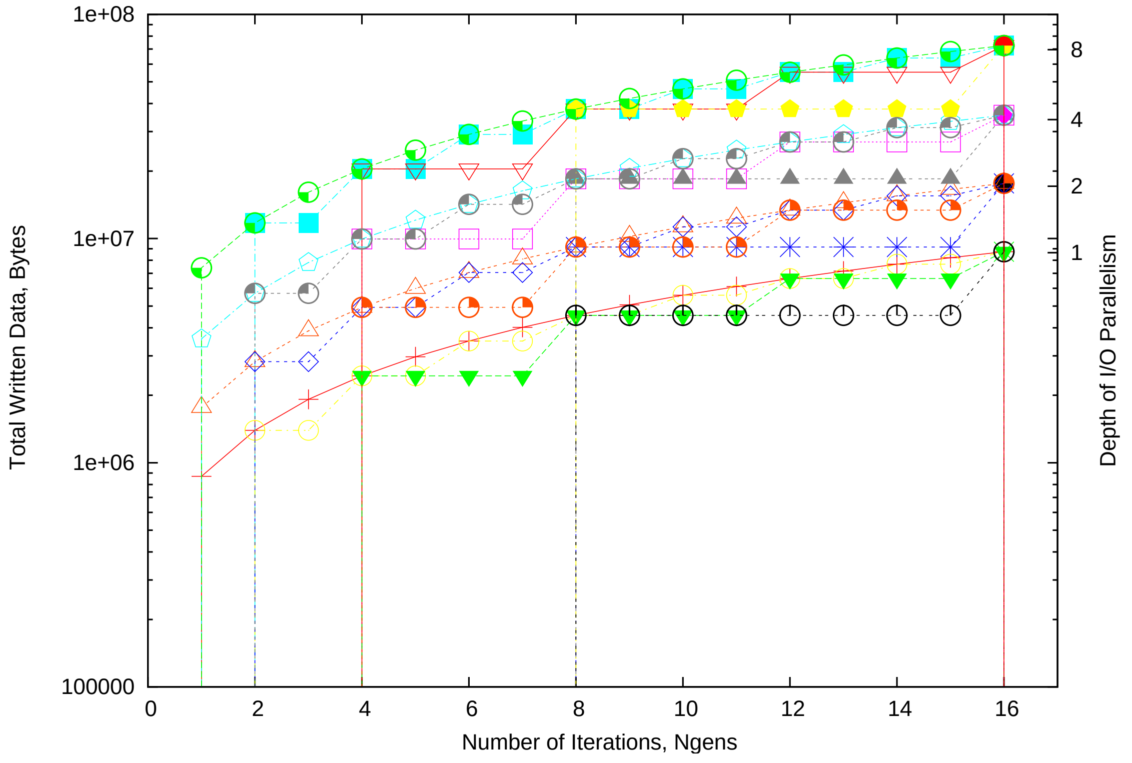

Figure 7.17: Total written data into the parallel file by a π_1 pattern over 16 iterations: In each iteration the application generates an amount of data of $\delta_{\pi_{ioz}} = 1, 2, 4, 8$ with a reference of 64KB (right y-axis). This data is written into the parallel file using the varstrip distribution function with $\lambda = 1, 2, 4, 8, 16$ over a total of 16 iterations. The curve corresponding to $\lambda = 1$ (writing into the file in each iteration) can be used as reference for each curve set.

where no significant difference was measured between speedup at iteration 1 and 8.

7.9 Speedup Comparison, Pattern π_1

Throughout the experiments that I have conducted in this evaluation, I have identified ranges of values for the striping unit and the *varstrip chunk size* that increase or decrease speedup in comparison to the default values. The objective of this experiment was to show the speedup presented by an `mcmc` analysis with 128 runs[12] with a π_1 pattern when using values for the striping unit and *varstrip chunk size* that belong to each one of these ranges, including the default values.
For this experiment I used an input matrix that generated an amount of data in each iteration with a *depth of I/O parallelism* $\delta_{\pi_{ioz}} = 376$. Using this matrix I ran 4 iterations the same `mcmc` analysis 6 times. For each execution different striping unit and *varstrip chunk size* values were used. Provided the number of runs and the *depth of I/O parallelism* the striping unit was set to the following values: 1, 1024, 65536. The *varstrip chunk size* was set to $\lambda = 1, 2, 4$.
Figure 7.19 visualizes the computed values. The upper half presents the amount of data written into the parallel file in each iteration. For the varstrip distribution function this shows the data written with $\lambda = 1, 2, 4$. After 4 iterations 75

[12]A relatively high number of runs before operating system swapping is activated

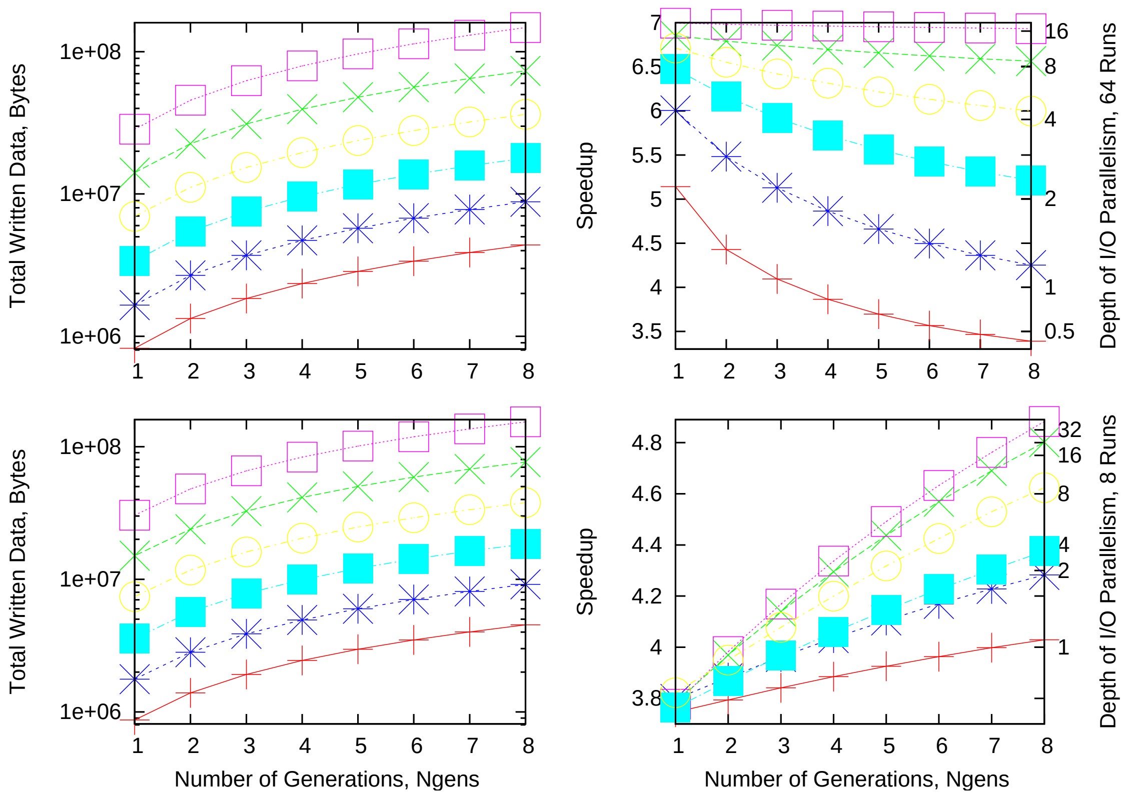

Figure 7.18: Varstrip *smallest-process-overhead speedup* in terms of number of runs: Total amount of data written into the parallel file and corresponding speedup values. These values correspond to 8 and 64 runs that generated amounts of data in each iteration, $data_{gen}$, with the *depth of I/O parallelism* values (with a 64 KB reference) presented on the rightmost y-axis.

MB (78978304 bytes) were written into the parallel file. Running this analysis over 100000 iterations generates a parallel file of 1.8 TB. In the lower half the corresponding computed speedup values are presented. Except for the case with $su = 1$, where a speedup of 6.6 was computed after 4 generations, most of the speedup values were close to one another. Nevertheless, the highest speedup in terms of distribution parameter values was computed for $\lambda = 1$, followed by $\lambda = 2$, $\lambda = 4$, $su = 1024$, and the $su = 65536$. These speedup values as well as the relationship among them corresponded to the expected values, according to the experiment results that I have described in this chapter. Even though the differences among these speedup values are relatively small, they become notorious in cases where execution times are in the range of weeks or months.

7.10 Summary

- The *mapping at first generation* functionality with physical distribution functions: In subsection 7.5 I demonstrate that the *mapping at first generation* partitioning function in combination with the simple_stripe distribution function writes the computed values of the p(MC)^3 algorithm into one single parallel file in the required format, independently of the parallel I/O pattern generated at any point in time during execution time. Furthermore, the computed results of *Complete-1* or π_1 patterns can be written into a parallel file by the *mapping at first generation* in combination with the varstrip distribution function.

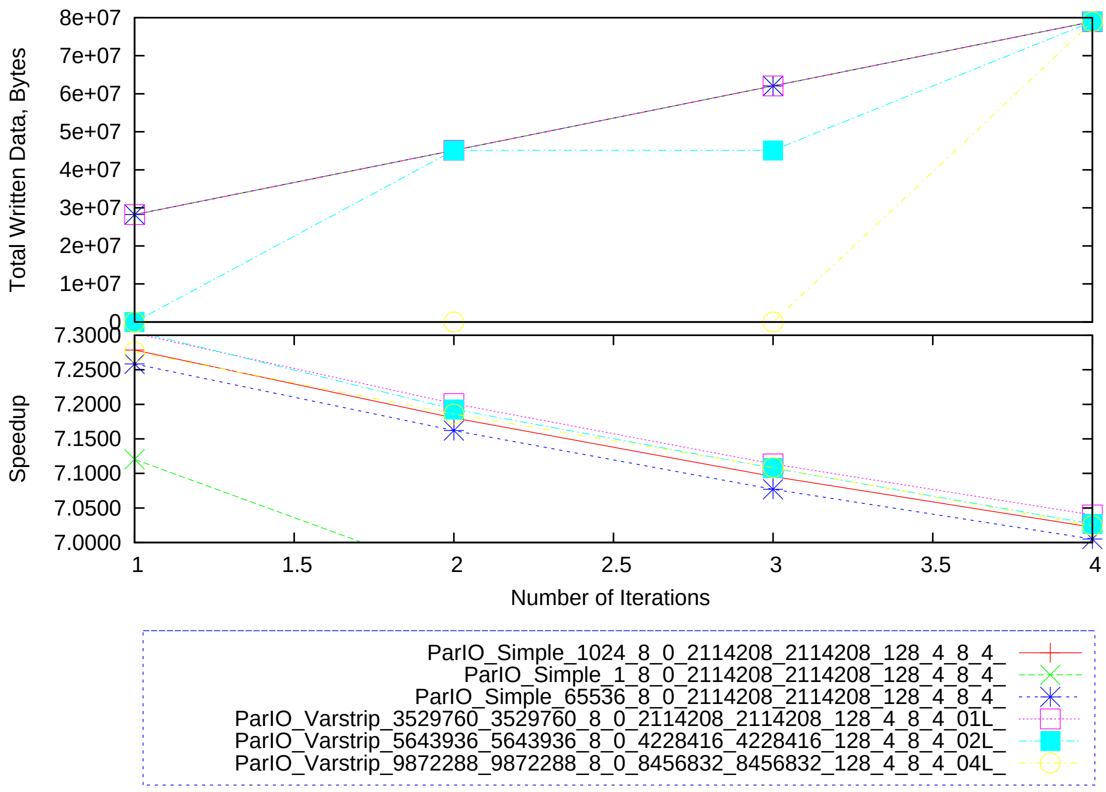

Figure 7.19: Speedup for different parallel I/O settings to compute 128 runs with a pattern π_1: The highest speedup was computed for the varstrip distribution function with $\lambda = 1$ and the lowest for the simple_stripe distribution function with $su = 1$ (7.05 and 6.6 after 4 iterations, respectively). For the simple_stripe distribution function a $su = 1024$ presents higher speedup values than the default value. The upper half shows the total amount of data written into the parallel file by all 8 MPI processes. Detailed information on the labeling for both distribution functions can be found under subsection A.4.1.2 in the appendix.

- Patterns selection and performance: Provided an I/O environment, in which the parallel file system uses the *all in one* configuration on an I/O cluster, one manner to additionally improve speedup with the *mapping at first generation* partitioning function is to run analyses with the π_1 pattern. In subsection 7.5.1 I demonstrate that this pattern presents higher speedup values than the other patterns regardless of the used physical distribution function.
- Speedup of π_1 patterns and distribution functions: The speedup values for π_1 patterns are conditioned by the balanced I/O load and the non-existence of I/O swaps in such patterns. This last property can be further supported at the parallel file system level by using the varstrip distribution function that stores the computed data on the same I/O node. Thus, it increases speedup values for π_1 patterns as I show in subsection 7.5.1 and 7.9.

- Suited application characteristics for the parallel I/O version of the code: In section 7.3 I demonstrate that from all input matrix dimensions solely the number of taxa influences the amount of data generated in each iteration. In subsection 7.6.1 I also demonstrate that higher speedup values are computed while increasing the amount of data generated in each iteration and the number of runs. Thus, the parallel I/O implementation that I propose in this work is suited to conduct `mcmc` analyses that involve higher number of taxa and runs. Under section 7.9 the highest speedup for a π_1 pattern was measured when applying the varstrip distribution function and λ_{opt}. The smallest speedup value was measured while using the simple_stripe distribution function and a striping unit smaller than 100 bytes.

- Number of MPI processes: The number of MPI processes chosen to conduct an `mcmc` analysis strongly determines the speedup of the parallel I/O application's implementation. In this work I define as the *process saturation number* the number of MPI processes that yields the highest speedup. In subsection 7.6.2.2 and 7.6.2.1 I measured *process saturation number* values of 4, 8, and 16 while using 2, 4, and 8 cluster nodes, respectively. Since the testbed cluster has only SMP nodes with 2 CPUs, these values correspond to the total number of used CPUs. Thus, resulting in an assignment of one MPI process per CPU and two CPUs per secondary storage device. Furthermore, I also show that the same speedup can be obtained while choosing two different numbers of MPI processes. Users should select numbers of processes smaller or equal than the *process saturation number* since they yield *smallest-process-overhead speedup* values, which translate into smaller execution times. Furthermore, the measurements under subsection 7.6.2 show that the difference between the speedup for number of MPI processes equal or greater than the *process saturation number* decrements while incrementing the number of runs.

- The striping unit size: By selecting an appropriate striping unit size the execution time can be reduced. In subsection 7.7.1 I show that up to a value of $\delta_{\pi_{ioz}} = 8$ for the data generated in each iteration and $\delta_{\pi_{iox}} = 100\%$, selecting a striping unit value in the range $[su_{threshhold}, su_{max}]$ provides the smallest execution time values. The maximum value $su_{max} = \frac{data_{gen}}{N_{IO_t}}$ and the minimum value is in this case $su_{threshhold} = 100$. Furthermore, I also showed how values $\delta_{\pi_{iox}} < 100\%$ yield execution increment across iterations.

- The maximum and optimal values for λ: In cases in which operating system swapping is activated during the first iteration or the amount of data has a *depth of I/O parallelism* $\delta_{\pi_{ioz}} \geq 8$, I propose to use $\lambda = 1$ for the *varstrip chunk size* value. If operating system swapping is not activated during the first iteration as a first approach I propose to use λ_{max}. The main objective of this last approach is to keep as much data in main memory before operating system swapping is activated on any of the cluster nodes as expressed in equation 6.1. Applying λ_{max} for any amount of generated data in each iteration yields reproducible total execution times among different executions, but does not necessarily yield the smallest execution time. Thus, in subsection 7.7.2.3 I demonstrate that up to an amount of data with $\delta_{\pi_{ioz}} = 4$ in each generation a value of $1 \leq \lambda \leq 16$ can be applied to reduce the execution time. Selecting higher values for λ in this range while generating amounts of data approaching $\delta_{\pi_{ioz}} = 1$ improves performance. In the case of $\delta_{\pi_{ioz}} = 1$ choosing $\lambda = 16$ instead of $\lambda = 1$ reduced the execution time by 3%.

- Speedup and iterations: The speedup does not present a constant value across iterations. It shows an asymptotic behavior in terms of the number of iterations that remains at a maximum or minimum value after a certain number of iterations. Constant speedup values across iterations are computed for higher number of runs and $\delta_{\pi_{ioz}}$ (in this evaluation 64 runs and $\delta_{\pi_{ioz}} = 16$).

8. Conclusion

8.1 Summary

This section summarizes the whole work at a high level of abstraction[1]. Chapter 2 introduced the necessary theoretical background in the area of parallel I/O and computational phylogenetical analysis in order to describe the research question under section 2.3 with the appropriate terminology. Section 2.4 pointed out other projects that have been conducted to tackle similar research questions.
Chapter 3 presented the varstrip distribution function that we have proposed for the PVFS2 parallel file system. It also discussed a variant of the simple_stripe distribution function, in which the striping unit can be set to any value. In order to compare these functions' performance values, section 3.3 described a set of metrics that I have applied in this work.
Chapter 4 described the first approach that I experimentally conducted in order to gain knowledge about the application's I/O requirements in the serial I/O version.
In chapter 5 I have theoretically categorized the number and type of *semi-random temporal parallel file I/O access patterns* that can appear during the execution time of the p(MC)^3 algorithm. In chapter 6 I have presented the two main contributions of this work: The *mapping at first generation* partitioning function to handle all categorized parallel I/O patterns and the *parallel line buffering* mechanism to set the *varstrip chunk size* value within the varstrip distribution function.
Chapter 7 described the work that I conducted to experimentally demonstrate the functionality of the parallel I/O implementation to correctly serve all identified parallel I/O patterns. This chapter showed the speedup (equation 7.1) in terms of the following parameters: Number of MPI processes, I/O nodes, runs, Markov chains, and iterations. Furthermore, in section 7.7 I have described the conditions under which the speedup can be improved by selecting the appropriate pattern, physical distribution function, striping unit, and *varstrip chunk size* values. Finally, section 7.9 described an experiment to demonstrate the speedup for the π_1 pattern for 128 runs in terms of different distribution function parameters.

[1]More detailed summaries are found at the end of each corresponding chapter. This is especially notorious for the parallel I/O adaptation chapter 6.4 and the corresponding evaluation chapter 7.10.

8.2 Conclusions

8.2.1 (Semi-)Random Parallel I/O Accesses

1. In an ideal set of random parallel I/O accesses, the number of processes parallely accessing the I/O subsystem at different addresses changes between two consecutive iterations, whereas in a semi-random parallel I/O accesses set it changes, but not necessarily between two consecutive iterations.
   In an application with semi-random based computation, the semi-randomness during computation may or not translate into semi-random accesses to the I/O subsystem. In this thesis I have shown that a π_1 pattern remains constant throughout the execution time, although the computation from one iteration to the next as well as the exchange between chain states depend on semi-random processes. In this pattern these computations occur on the same compute I/O node so that the I/O subsystem access pattern from the first iteration remains constant through the whole execution time.
   Nevertheless, if a *Complete+1* or a *Partial-1* pattern is chosen during the first iteration, the semi-randomness during computation translates also into *semi-random temporal parallel file I/O access patterns* that vary throughout execution time. On the parallel I/O environment presented in section 7.2 in this thesis I have demonstrated that different patterns present different speedup values. The highest speedup value is presented by the π_1 pattern while being served by the varstrip distribution function with an optimal *varstrip chunk size* value as described under section 7.9.

8.2.2 Parallel I/O Implementation

1. A parallel I/O implementation increases the speedup of a parallel scientific application that generates and writes into secondary storage a relatively big amount of data in each iteration [2]. In this work I have shown for this application and a chosen number of write operations that this amount of data depends on the number of taxa and number of runs.
   Since the speedup correlates with the number of runs and the number of runs correlates with the computed solution's precision, the strength of the parallel I/O implementation that I propose in this work is not only the computed speedup, but also the increment of the solution's precision at the same time. This applies in cases where higher number of runs cannot be conducted with the serial I/O version of the program due to I/O subsystem constraints[3].
   On the other hand, an application setup that generates in each iteration a relatively small amount of data over a long period of time presents lower speedup values. Nevertheless, conducting analyses with the parallel I/O version can still be benefitial for the user if the total generated data is required in a parallel file for fast postprocessing.

2. Provided the parallel I/O environment discussed in section 7.2, one relatively simple setup to exploit the benefits of the parallel I/O implementation that I

[2]For this application the amount of data in each iteration can be computed using the total amount of data express in equation 4.1 and the number of write operations.

[3]I describe such one soft constraint in subsection 7.6.1.

propose in this work, is to select whenever possible, a π_1 pattern in combination with the varstrip distribution function with its *varstrip chunk size* default value. If a π_1 pattern cannot be generated, the parallel file must be served by the simple_stripe distribution function.
The strength of setting optimal values for the *varstrip chunk size* or striping unit is especially notorious when conducting analyses with higher number of iterations that generate higher amounts of data in each one of these iterations, since under these conditions the impact of speedup differences, such as those caused by distribution function parameters tuning discussed in subsection 8.2.4.1 and 8.2.4.2, translate into relatively big execution time reductions.

3. For a π_1 pattern I have shown that the speedup varied in a semi-parabolic manner in terms of the MPI processes number. Except for the highest speedup value (attained with the *process saturation number*) the same speedup value was computed by choosing two different MPI process numbers to execute the serial and parallel I/O implementations of the program. Thus, users should select MPI process numbers smaller or equal than the *process saturation number* in order to obtain *smallest-process-overhead speedup,* the speedup that translates into the smallest execution times.
Experimentally I showed that the *process saturation number* was equal to the total used number of CPU of the used I/O cluster. Furthermore, the speedup values also approached a maximum value towards this value.

8.2.3 The *Mapping at First Generation Function*

1. In order to provide a parallel application a medium to access a parallel I/O interface independently of the number of processes, as in the case of applications with *semi-random temporal parallel file I/O access patterns,* in this work I have proposed the *mapping at first generation* partitioning function. This function creates a map of the parallel file's structure at iteration one. This map is a global data structure containing all offsets and sizes of the parallel file. It is constructed based upon a predefined file format, one of each elements that the file will contain as well as their corresponding number of entries in the file. At a writing iteration the process(es) write the data into the corresponding address(es) based upon the map and the iteration number.
Since each process has a file map, the writing operation can be executed independently of the number of processes. Thus, supporting applications with (semi-)random spatial and time I/O access patterns.

8.2.4 Physical Distribution Functions

8.2.4.1 varstrip distribution function

1. While serving π_1 patterns with the varstrip distribution function in this work I have proposed to use for the *varstrip chunk size* a default value equal to the amount of data generated by the application in each iteration ($\lambda = 1$). Thus, differently from the fixed 64KB striping unit default value of the simple_stripe distribution function, this default value automatically changes with the generated data in each iteration.
In order not to increment execution time, I have proposed to use this value

for cases, in which operating system swapping is activated during the first iteration or in cases where the amount of data has a *depth of I/O parallelism* $\delta_{\pi_{ioz}} \geq DP_{lim}$. With the testbed used in this work I measured $DP_{lim} = 8$ under subsection 7.7.2.3. One manner that I have proposed in this work to further reduce execution time with the varstrip distribution function is to apply *parallel line buffering* to set an optimal *varstrip chunk size* to λ_{opt} $(1 < \lambda_{opt} < \lambda_{max})$. In subsection 7.7.2.3 I demonstrated that up to an amount of data with $\delta_{\pi_{ioz}} = 4$ in each generation, a value of $1 \leq \lambda \leq 16$ can be applied to reduce the execution time. Selecting higher values for λ in this range while generating amounts of data approaching $\delta_{\pi_{ioz}} = 1$ improves performance. In the case of $\delta_{\pi_{ioz}} = 1$ choosing $\lambda = 16$ instead of $\lambda = 1$ reduced the execution time by 3%.

8.2.4.2 simple_stripe distribution function

1. In a round robin based physical distribution function selecting a striping unit to increase throughput for a write-only application should be done in two steps. The first one is to select[4] a su_{max} value that guarantees a *physical degree of I/O parallelism* of 100% and a *depth of I/O parallelism* of 1. The second step consists in selecting a striping unit value in the range $[1, su_{max}]$.
   In this work I have experimentally demonstrated that rather than having only one striping unit value that notoriously yields the smallest execution time, the smallest or values within 2% tolerance from this value can be obtained, when selecting the striping unit value from a range $[su_{threshhold}, su_{max}]$. The value of $su_{threshhold}$ depends on the amount of data to be written in each iteration and I/O parameters. In subsection 7.7.1 I have experimentally determined a $su_{threshhold} = 100$ for an amount of data generated in each iteration of up to $\delta_{\pi_{ioz}} = 8$.
   In this work I have shown that for an amount of written data in each iteration with $\delta_{\pi_{ioz}} = 1$, selecting values $su < su_{threshhold}$ incremented execution time up to 10% compared to the execution time measured for $su = su_{max}$. This percentage diminished while incrementing $\delta_{\pi_{ioz}}$. For $\delta_{\pi_{ioz}} = 8$ it was around 2%.

8.3 Future Research

This section lists some research topics that can be conducted based upon this work's contribution.

8.3.1 Random Parallel I/O Applications

This thesis proposes the *mapping at first generation* to handle semi-random spatial and temporal parallel I/O access patterns for a write-only case. Research can be aimed at using this function or similar ones to handle the parallel I/O accesses of applications, in which the operations such as write, read, seek, and their combinations also randomly change throughout execution time.

[4]Via user hint or automatic mechanism

8.3.2 Parallel I/O Environments

The *mapping at first generation* based parallel I/O implementation proposed in this work is a general solution that can be applied to handle fully random parallel file I/O access patterns. Its application presupposes the existence of an appropriate parallel I/O environment. Such an environment is strongly conditioned by the used hardware. Therefore research work can be conducted to study the application's parallel I/O implementation on clusters using other types of secondary storage devices (such as SDDs), other ratios between CPUs and secondary storage devices, and other parallel file systems.

8.3.3 Parallel File Systems Mechanisms

In this work I have used two types of flexible distribution functions in conjuction with the *mapping at first generation.* In the used I/O environment for a π_1 the varstrip distribution function with an appropriate *varstrip chunk size* value presents the highest speedup. Nevertheless, only the simple_stripe distribution function supports each one of the generated parallel I/O access patterns by the application. This fact demonstrates that no single physical distribution function can satisfy each one of the application's requirements. Therefore research can be aimed at augmenting the set of physical distribution functions, based upon other applications' and I/O environments' parameters. Furthermore, other methods can be developed to select these functions.

8.3.4 Parallel File Format and I/O Optimizations

The construction of the map μ for the *mapping at first generation* depends on a given file format. The file format that I used in this work corresponds to translating into spatial concatenation the time sequence in which the p and t files are written in the serial I/O version. This does not necessarily mean that this format is the best suited to provide the highest performance. Thus, future research can be addressed towards the determination of the partition function's performance in terms of different file formats and the determination of the relationship between these formats and parallel library optimizations such as collective buffering, and data sieving. Furthermore, the partitioning function's implementation with non-blocking operations can be considered.

8.3.5 Postprocessing Methods

This thesis concentrates on the execution time reduction for conducting `mcmc` analysis. Research can be aimed at parallelizing the other phylogenetical inference stages in order to minimize the total time used to conduct such an inference.
Based upon the data stored in parallel files proposed in this work, future research can be directed towards the reduction of the postprocessing time by developing parallel methods to process this data. This includes the conception of file parallel I/O methods to read the data, parallel algorithms to process it, and parallel I/O visualization methods.

8.3.6 Parallel Input

Research can also be conducted to conceive new decomposition functions in terms of computation that take into consideration input matrices that are stored in a parallel file. Thus, allowing the computation of bigger phylogenies and reducing the read in time into main memory.

A. Appendix

A.1 Notation

A.1.1 p(MC)^3 Parameters Relevant for I/O

This section presents the used notation for the p(MC)^3 parameters that are relevant for I/O activities.

Parameter	**Description**
$P_k, k = 0...np - 1$	MPI Process k. The total number of processes is np, which is a parameter of `mpiexec`.
$R_l, l = 1...nruns$	Run l. The total number of runs that are started simultaneously is $nruns$ This is a parameter of `mcmc`.
$Ch_i, i = 0...nchains - 1$	Markov chain i. The number of chains for each run is $nchains$, an argument of `mcmc`.
$Ch_{global} = nchains * nruns$	Total number of Markov chains used for an `mcmc` analysis.
$Ch_{proc} = \frac{Ch_{global}}{np}$	Number of chains assign to one process P_k in an equally load balance system.
$P_{rem} = Ch_{global} \% np$	Chains remaining after load balacing them among processes.

Table A.1: Notation for the p(MC)^3 algorithm's parameters relevant for I/O operations

A.2 Temporal and Spatial Notation

A.2.1 Spatial

This section presents the notation used to describe spatial serial states.

Parameter	Description
$A \prec B$ or $B \succ A$	Denotes the spatial relationship between the two spaces in memory It means that the A has a smaller memory address than B. There is no intersection

Table A.2: Notation for spatial parameters

A.2.2 Temporal

This section presents the notation used to describe temporal serial states.

Parameter	Description
$A \prec B$ or $B \succ A$	Denotes the temporal relationship between the two activities A and B. It means that the A activity must be finished before beginning activity B.

Table A.3: Notation for temporal parameters

A.3 Testbed Cluster

A.3.1 Application Software

For the experiments that I conducted in this work two sets of input matrices were used. One set contained biological DNA sequences as described in subsection A.3.2 and another contained synthetical sequences as described below in subsection A.3.3.

A.3.2 Biological Sequences

The following nexus file headers show the input matrices' dimensions of used matrices containing actual biological sequences. The `primates.nex` is delivered with the application's serial I/O version.

- primates.nex:

```
#NEXUS
begin data;
dimensions ntax=12 nchar=898;
format datatype=dna interleave=no gap=-;
matrix
```

- anatidae.nex:

```
#NEXUS
begin data;
dimensions ntax=125 nchar=2086;
format datatype=dna missing=n gap=-;
matrix
```

In order to have different amounts of data generated in each iteration, I used sub- and supra-matrices based on the matrix contained in `anatidae.nex`. Table A.4 provides information on the used input matrices in terms of the number of taxa, string length, input file size, and the amount of data generated in each iteration per 1 run in bytes.

Number of Taxa	**String Length**	**File Sizes**	**Generated Data pro Iteration**
		(Bytes)	(8 I/O nodes, 1 run, Bytes)
12	898	11052	308
8	2086	16948	217
12	2086	25359	722
16	2086	33787	401
32	2086	67447	769
64	2086	134756	1506
256	2086	538943	6080
512	2086	1077761	12225
1024	2086	2155400	24538

Table A.4: Sizes of input matrices with actual biological strings.

A.3.3 Synthetical Sequences

In order to thoroughly test the *MFG* function with the distribution functions using input matrix sizes that generate amount of data in each iteration with a *depth of I/O parallelism* greater than one, I wrote a small program called `gtaxa` that based upon a given number of taxa and their length, it randomly generates a taxa set in a nexus file, which immediately can be provided as input to MrBayes. I have also included this tool in the source code tree of the parallel I/O implementation. The following example shows the first 2 taxa of 1000000 in a `gtaxa` generated file:

```
#NEXUS
[TITLE: Synthetic input matrix generated from gtaxa.]
[TITLE: A very simple program that I have written to]
[TITLE: test the parallel I/O version of MrBayes3.1]
[TITLE: hipolito.vasquez@informatik.uni-heidelberg.de]
begin data;
dimensions ntax=1000000 nchar=10;
format datatype=DNA missing=n gap=-;

matrix
1
tctttactcg
2
cgcgttggag
```

A.3.4 Synthetical Sequences Input Matrix Sizes

In order to fulfill the corresponding objectives of the different experiments that I conducted as part of the evaluation, I generated with `gtaxa` a set of synthetical input matrices based upon the number of taxa, string lengths, file size, *depth of I/O parallelism* of the corresponding generated data in each generation.
Table A.5 and A.6 are included in order to ease cross referencing among these sizes.

I/O Type	**Input Matrix**	**Distribution Function Parameters**	**I/O Parallelism Conditions** $\delta_{\pi_{iox}} = 100\%$
	(8 Runs) String length = 10 Taxa, Size in KBytes	$[ss \mid \lambda]$	Depth of Parallelism $\delta_{\pi_{ioz}}$ $\downarrow$
Parallel I/O: Simple_stripe	2664, 41	$ss = 65536$	1
	5328, 83		2
	10656, 167		4
	21312, 344		8
	42624, 698		16
	85248, 1400		32
Simple_stripe	2664, 41	$ss = 1024$	512
	5328, 83		1000
	10656, 167		2000
	21312, 344		4000
	42624, 698		8000
	85248, 1400		16000
Parallel I/O: Varstrip	2664, 41	$\lambda = 1$	1
	5328, 83		1
	10656, 167		1
	21312, 344		1
	42624, 698		1
	85248, 1400		1
Varstrip	2664, 41	$\lambda = 8$	1
	5328, 83		1
	10656, 167		1
	21312, 344		1
	42624, 698		1
	85248, 1400		1

Table A.5: Set of parallel I/O measurement points for a π_1 pattern and 8 runs. In these setups in each iteration the degree of parallelism is $\delta_{\pi_{iox}} = 100\%$ and the depth of parallelism is $\delta_{\pi_{ioz}} \geq 1$.

I/O Type	**Input Matrix**	**Distribution Function Parameters**	**I/O Parallelism Conditions** $\delta_{\pi_{iox}} = 100\%$
	(64 Runs) String length = 10 Taxa, Size in KBytes	$[ss \mid \lambda\ ]$	Depth of Parallelism $\delta_{\pi_{ioz}}$ $\downarrow$
Parallel I/O: Simple_stripe	334, 5.12	$ss = 65536$	1
	668, 10.4		2
	1336, 20.9		4
	2664, 43		8
	5328, 87.2		16
	10656, 175		32
Simple_stripe	334, 5.12	$ss = 1024$	512
	668, 10.4		1000
	1336, 20.9		2000
	2664, 43		4000
	5328, 87.2		8000
	10656, 175		16000
Parallel I/O: Varstrip	334, 5.12	$\lambda = 1$	1
	668, 10.4		1
	1336, 20.9		1
	2664, 43		1
	5328, 87.2		1
	10656, 175		1
Varstrip	334, 5.12	$\lambda = 8$	1
	668, 10.4		1
	1336, 20.9		1
	2664, 43		1
	5328, 87.2		1
	10656, 175		1

Table A.6: Set of parallel I/O measurement points for a π_1 pattern and 64 runs. In these setups in each iteration the degree of parallelism is $\delta_{\pi_{iox}} = 100\%$ and the depth of parallelism is $\delta_{\pi_{ioz}} \geq 1$.

A.3.5 Data per Iteration

Complementary to the information contained in table A.5 and A.6, figure A.1 shows experimental measured values of the amount of data generated in each iteration by eight runs in terms of the number of taxa in the input matrix. These were synthetical input matrices generated with `gtaxa` for 4 different string lengths.

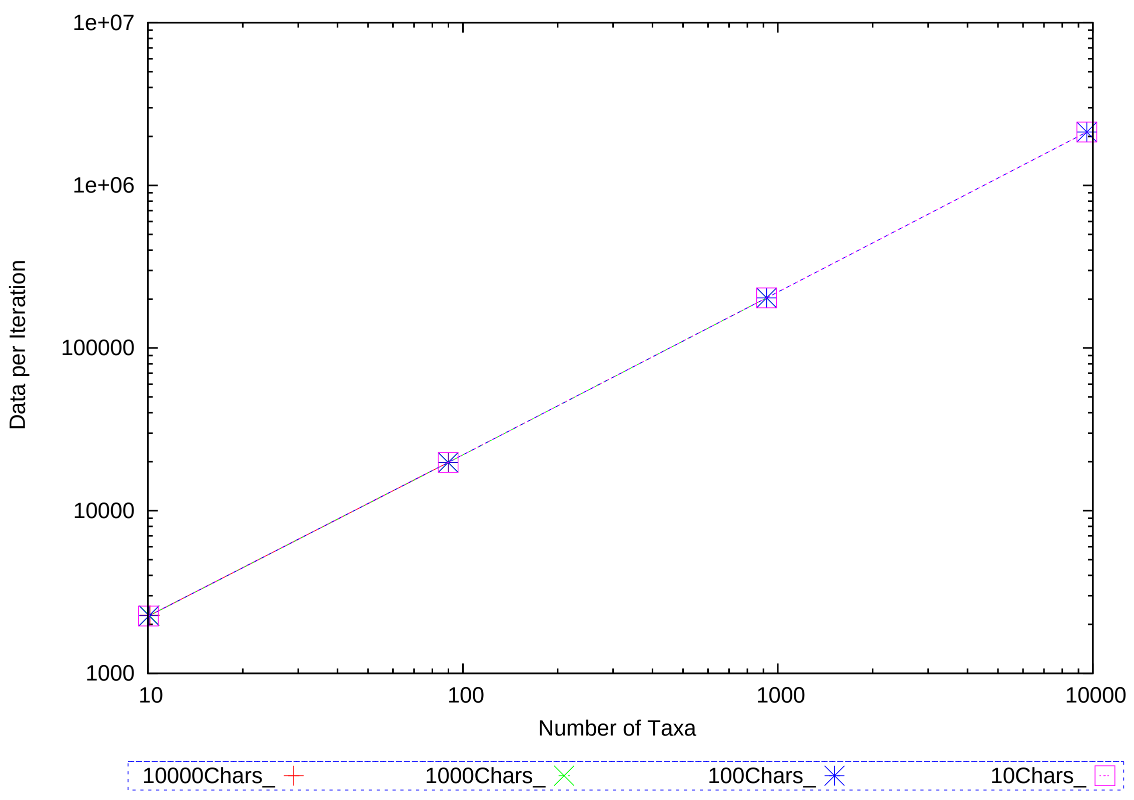

Figure A.1: Generated data per iteration by eight runs: The generated data per iteration only linearly changes with the number of taxa and it does not change with the DNA string length.

A.3.6 Scripts for `mcmc`, Serial I/O

In order to measure I/O performance, timing and written data, I have included for the serial and parallel I/O program's version the `pmsfreq`, *performance measurement sample frequency.*

```
begin mrbayes;
set autoclose=yes nowarn=yes;
    execute /home/vasquez/TracingSandBox/nexfiles/primates.nex;
    mcmc ngen=10000 samplefreq=1 nruns=496 nchains=4 \
    temp=0.5 mcmcdiagn=no \
    pmsfreq=1;
end;
```

A.3.7 Scripts for `mcmc`, Parallel I/O

A.3.7.1 Example, simple_stripe distribution function

```
begin mrbayes;
set autoclose=yes nowarn=yes;
    execute /home/vasquez/TracingSandBox/nexfiles/primates.nex;
    mcmc ngen=10000 samplefreq=1 nruns=512 nchains=4 \
    temp=0.5 mcmcdiagn=no \
    pmsfreq=1 dfunction=simple dfsetup=23 dnmlines=1;
end;
```

A.3.7.2 Example, varstrip distribution function

```
begin mrbayes;
set autoclose=yes nowarn=yes;
    execute /home/vasquez/TracingSandBox/nexfiles/anatidae8.nex;
    mcmc ngen=20 samplefreq=1 nruns=64 nchains=4 \
    temp=0.5 mcmcdiagn=no \
    pmsfreq=1 dfunction=varstrip dfsetup=1 dnmlines=20;
end;
```

A.3.8 Parallel I/O Library

A.3.8.1 `config.log File's Header`

```
../configure --prefix=/home/vasquez/SIMPLELOCALPVFS2MPICH2 \
--enable-romio \
--with-file=ufs+nfs+pvfs2 \
--with-pvfs2=/home/vasquez/SIMPLELOCALPVFS2MPICH2 \
--with-mpe \
--enable-g=meminit, dbg \
--enable-fast=O0
```

A.3.9 Parallel File System

A.3.9.1 `config.log File's Header`

```
../configure --prefix=/home/vasquez/SIMPLELOCALPVFS2MPICH2 \
--with-mpi==/home/vasquez/SIMPLELOCALPVFS2MPICH2 \
--with-mptrace
```

A.3.9.2 PVFS2 *all in one* Configuration

I call in this work the PVFS2 *all in one* configuration, the configuration in which each pvfs2 node is a client, server and metadata at the same time. The pvfs2.conf file looks as follows:

```
<Defaults>
        UnexpectedRequests 50
        EventLogging none
        EnableTracing no
        LogStamp datetime
        BMIModules bmi_tcp
        FlowModules flowproto_multiqueue
        PerfUpdateInterval 1000
        ServerJobBMITimeoutSecs 30
        ServerJobFlowTimeoutSecs 30
        ClientJobBMITimeoutSecs 300
        ClientJobFlowTimeoutSecs 300
        ClientRetryLimit 5
        ClientRetryDelayMilliSecs 2000
        PrecreateBatchSize 512
        PrecreateLowThreshold 256

        StorageSpace /tmp/pvfs2-vasquez
        LogFile /tmp/pvfs2-server.log
</Defaults>

<Aliases>
        Alias node01 tcp://node01:22333
        Alias node02 tcp://node02:22333
        Alias node03 tcp://node03:22333
        Alias node04 tcp://node04:22333
        Alias node05 tcp://node05:22333
        Alias node06 tcp://node06:22333
        Alias node07 tcp://node07:22333
        Alias node08 tcp://node08:22333
</Aliases>

<Filesystem>
```

```
        Name pvfs2-fs
        ID 1150543096
        RootHandle 1048576
        FileStuffing yes
        <MetaHandleRanges>
                Range node01 3-576460752303423489
                Range node02 576460752303423490-1152921504606846976
                Range node03 1152921504606846977-1729382256910270463
                Range node04 1729382256910270464-2305843009213693950
                Range node05 2305843009213693951-2882303761517117437
                Range node06 2882303761517117438-3458764513820540924
                Range node07 3458764513820540925-4035225266123964411
                Range node08 4035225266123964412-4611686018427387898
        </MetaHandleRanges>
        <DataHandleRanges>
                Range node01 4611686018427387899-5188146770730811385
                Range node02 5188146770730811386-5764607523034234872
                Range node03 5764607523034234873-6341068275337658359
                Range node04 6341068275337658360-6917529027641081846
                Range node05 6917529027641081847-7493989779944505333
                Range node06 7493989779944505334-8070450532247928820
                Range node07 8070450532247928821-8646911284551352307
                Range node08 8646911284551352308-9223372036854775794
        </DataHandleRanges>
        <StorageHints>
                TroveSyncMeta yes
                TroveSyncData no
                TroveMethod alt-aio
        </StorageHints>
</Filesystem>
```

A.3.9.3 Simple_stripe Distribution Function, Number of Datafiles

The following `pvfs2-stat` output provides information on a parallel file generated by 512 runs in one generation with an input matrix of 11 KB (12 Taxa, 898 characters). It was stored into one datafile with the stuffing mechanism enabled by the simple_stripe distribution function with its default striping unit value. Without this mechanism it is stored in 5 datafiles.

```
-------------------------------------------------------
  File Name     : /mnt/pvfs2-vasquez/hv12T_898S.nex.pvs
  Relative Name : /hv12T_898S.nex.pvs
  fs ID         : 1150543096
  Handle        : 2882303761517117431
  Mask          : 2704000177
  Permissions   : 644
  Type          : Regular File
  Size          : 288664
  Owner         : 13101 (vasquez)
  Group         : 100 (users)
  atime         : 1284719914 (Fri Sep 17 12:38:34 2010)
  mtime         : 1284719914 (Fri Sep 17 12:38:34 2010)
  ctime         : 1284719914 (Fri Sep 17 12:38:34 2010)
  datafiles     : 1
  flags         : none
```

The following `pvfs2-stat` output shows a parallel file stored in eight data files without stuffing for the simple_stripe distribution function.

```
-------------------------------------------------------
  File Name     : /mnt/pvfs2-vasquez/hv2664T_10S.nex.pvs
  Relative Name : /hv2664T_10S.nex.pvs
  fs ID         : 1150543096
  Handle        : 3458764513820540918
  Mask          : 2704000177
  Permissions   : 644
  Type          : Regular File
  Size          : 869832
  Owner         : 13101 (vasquez)
  Group         : 100 (users)
  atime         : 1284717336 (Fri Sep 17 11:55:36 2010)
  mtime         : 1284717336 (Fri Sep 17 11:55:36 2010)
  ctime         : 1284717336 (Fri Sep 17 11:55:36 2010)
  datafiles     : 8
  flags         : none
```

A.3.10 Hardware

A.3.10.1 Nodes

The used testbed cluster has eight working nodes. The nodes are provided with:

- Intel Server Board SE7500CW2

- Two Intel Xeon 2GHz CPUs
- 1 GB DDR-RAM
- 80GB IDE HDD
- Two 100-MBit/s-Ethernet-Ports (out of use)
- Two 1-GBit/s-Ethernet-Port (one in use) / Intel 82545EM Gigabit Ethernet Controller

A.4 Experimental Evaluation

In order to conduct the experimental evaluation, I have defined a set of performance metrics, which the program writes into the performance measurement files at certain points in time. These metrics' values are used to construct the correspondings `pvsperf` measurement performance file's name and are also included as headers in these files. Subsection A.4.1.1 and A.4.1.2 include three examples to explain the notation. In the case of parallel I/O the metrics vary depending on the used distribution function. Some notations are a little bit different from these presented here, but they follow the same order.

A.4.1 Metrics

A.4.1.1 Serial I/O

The string `SIO_32_4_128_10000_` shows an example of the metrics that were recorded for serial I/O. Its components are interpreted as follows:

- SIO: Serial I/O
- <32>: Number of runs
- <4>: Number of chains
- <128>: Number of MPI processes
- <10000>: Maximum number of generations.

I include these metrics in the corresponding performance measurement file as follows:

```
#$ Author: hipolito.vasquez@informatik.uni-heidelberg.de$
#$ Date:$
#Hipolito Vasquez Lucas's PhD Thesis Experiment's File
#Experiment's Id:     SIO_32_4_128_10000_.pvsperf
#Conducted on:        Tue Dec  1 14:19:27 2009

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData    [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O              [0 = SIO, 1 = ParIO]
```

A.4.1.2 Parallel I/O

The string `ParIO_Simple_4096_8_0_308_308_8_3_8_10000_` shows an example of the metrics that were recorded for parallel I/O in the case of using the simple_stripe distribution function. Its components have the following meaning:

- ParIO: Parallel I/O
- <Simple>: Distribution function's name
- <4096>: striping unit size
- <8>: Number of MPI processes that write a buffer of "big" size
- <0>: Number of MPI processes that write a buffer of "small" size
- <308>: "big" size in bytes
- <308>: "small" size in bytes
- <8>: Number of runs
- <3>: Number of chains
- <8>: Number of MPI processes
- <10000>: Maximum number of generations

I include these parameters in the file's header as follows:

```
#$ Author: hipolito.vasquez@informatik.uni-heidelberg.de$
#$ Date:$
#Hipolito Vasquez Lucas's PhD Thesis Experiment's File
#Experiment's Id:
ParIO_Simple_4096_8_0_308_308_8_3_8_10000_.pvsperf
#Conducted on:        Sat Dec 19 01:53:32 2009

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData    [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) StripingUnit         [ ]
```

The string `ParIO_Varstrip_669_669_8_0_308_308_8_3_8_10_` shows an example of the metrics that were recorded for parallel I/O in the case of using the varstrip distribution function. Its components have the following meaning:

- ParIO: Parallel I/O
- <Varstrip>: Distribution function's name
- <669>: Maximum value of the *varstrip chunk size*
- <669>: Minimum value of the *varstrip chunk size*
- <8>: Number of MPI processes that write a buffer of "big" size
- <0>: Number of MPI processes that write a buffer of "small" size
- <308>: "big" size in bytes
- <308>: "small" size in bytes
- <8>: Number of runs
- <3>: Number of chains
- <8>: Number of MPI processes
- <10>: Maximum number of generations

I record these metrics in the file as follows:

```
#$ Author: hipolito.vasquez@informatik.uni-heidelberg.de$
#$ Date:$
#Hipolito Vasquez Lucas's PhD Thesis Experiment's File
#Experiment's Id:
#ParIO_Varstrip_669_669_8_0_308_308_8_3_8_10_.pvsperf
#Conducted on:        Sat Dec 19 15:19:47 2009

#Setup parameters:
#1) Ngen                      [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData        [1 ... 2000000000]
#4) MaxNgen                   [10000, 100000, 1000000]
#5) Nprocs                    [2^n, n=3 ...]
#6) Nchains                   [n*default, n = 1,2,.. ]
#7) Nruns                     [2^n, n = 2, ...]
#8) Type of I/O                 [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) Max varstrip chunk size   [<#MaxVsCS> ]
#15) Min varstrip chunk size   [<#MinVsCS> ]
```

A.5 Parallel Files Information for All Patterns

A.5.1 Header of Input Matrix

```
#NEXUS
[TITLE: Synthetic input matrix generated from gtaxa.]
[TITLE: A very simple program that I have written to]
[TITLE: test the parallel I/O version of MrBayes3.1]
[TITLE: hipolito.vasquez@informatik.uni-heidelberg.de]
begin data;
dimensions ntax=16 nchar=2000;
format datatype=DNA missing=n gap=-;
```

A.5.2 *Complete+1* Pattern

A.5.2.1 Parallel File Content

```
#NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((3:0.100000,(12:0.100000,2:0.100000):0.100000):0.100000,(10:0.100000,(16:0.
100000,11:0.100000):0.100000):0.100000):0.100000,(7:0.100000,14:0.100000):0.
100000):0.100000,(8:0.100000,(9:0.100000,(5:0.100000,(((13:0.100000,4:0.100000):
0.100000,6:0.100000):0.100000,15:0.100000):0.100000):0.100000):0.100000):0.
100000,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59800.826 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((3:0.100000,(12:0.100000,2:0.100000):0.100000):0.100000,(10:0.100000,(16:0.
100000,11:0.100000):0.100000):0.100000):0.100000,(7:0.100000,14:0.100000):0.
100000):0.100000,(8:0.100000,(9:0.100000,(5:0.100000,(((13:0.100000,4:0.100000):
0.100000,6:0.100000):0.100000,15:0.100000):0.100000):0.100000):0.100000):0.
100000,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59800.826 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((((12:0.100000,10:0.100000):0.100000,13:0.100000):0.100000,4:0.100000):0.
100000,9:0.100000):0.100000,15:0.100000):0.100000,(((7:0.107483,11:0.100000):0.
129502,14:0.100000):0.096775,(3:0.100000,((2:0.100000,16:0.100000):0.100000,5:0.
100000):0.100000):0.100000):0.100000):0.100000,6:0.100000):0.100000,8:0.100000,1
:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59671.911 2.934 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((16:0.100000,(((((15:0.100000,5:0.100000):0.100000,7:0.100000):0.100000,(9:0.
100000,10:0.100000):0.100000):0.100000,((8:0.100000,11:0.100000):0.100000,(14:0.
100000,(12:0.100000,4:0.100000):0.100000):0.100000):0.100000):0.100000,3:0.
100000):0.122039):0.105621,(6:0.100000,13:0.100000):0.134417):0.100000,2:0.
100000,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59678.061 2.962 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((16:0.100000,(3:0.100000,(7:0.100000,6:0.100000):0.100000):0.099669):0.
080284,10:0.100000):0.073754,(12:0.100000,((4:0.100000,9:0.100000):0.100000,8:0.
100000):0.100000):0.100000):0.100000,14:0.100000):0.100000,15:0.100000):0.100000
,5:0.100000):0.100000,(11:0.100000,(13:0.100000,2:0.100000):0.100000):0.100000,1
:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59802.907 2.854 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((9:0.115380,16:0.100000):0.092160,((((14:0.100000,((6:0.100000,3:0.100000):
0.100000,(2:0.100000,11:0.100000):0.100000):0.100000):0.100000,13:0.100000):0.
100000,4:0.100000):0.100000,15:0.100000):0.100000):0.102879,7:0.100000):0.100000
,12:0.100000):0.100000,8:0.100000):0.100000,(5:0.100000,10:0.100000):0.100000,1:
0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59741.693 2.910 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((((14:0.100000,9:0.100000):0.100000,5:0.100000):0.100000,12:0.100000):0.
100000,16:0.100000):0.100000,11:0.100000):0.100000,15:0.100000):0.100000,7:0.
100000):0.100000,(3:0.100000,4:0.100000):0.100000):0.100000,(2:0.100000,(10:0.
100000,((8:0.100000,13:0.100000):0.100000,6:0.100000):0.100000):0.100000):0.
100000,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59652.455 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((4:0.100000,(11:0.100000,(9:0.100000,((13:0.100000,7:0.100000):0.100000,15:0.
100000):0.100000):0.100000):0.100000):0.100000,(14:0.100000,3:0.100000):0.100000
):0.100000,(10:0.100000,((12:0.100000,(16:0.100000,8:0.100000):0.100000):0.
100000,6:0.100000):0.100000):0.100000):0.100000,(5:0.100000,2:0.100000):0.100000
,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59773.754 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9891715578]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((11:0.100000,(((((16:0.100000,12:0.100000):0.100000,(14:0.100000,3:0.100000)
:0.100000):0.107438,7:0.100000):0.073683,(4:0.100000,10:0.100000):0.100000):0.
136028,(5:0.100000,(13:0.100000,2:0.100000):0.100000):0.100000):0.100000):0.
100000,15:0.100000):0.100000,9:0.100000):0.100000,6:0.100000):0.100000,8:0.
100000,1:0.100000);[ID: 9891715578]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59926.068 2.917 0.250000 0.250000 0.250000
0.250000
```

A.5.2.2 Information on Parallel File

```
#Experiment's Id:      ParIO_Simple_65536_7_1_401_802_9_3_8_1_.pvsperf
#Conducted on:         Thu Sep 23 18:23:04 2010

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData     [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) StripingUnit          [ ]
#1)Ngen          #2)ExecTime          #3)OverallWrittenData
#4)MaxNgen  #5)Nprocs #6)Nchains #7)Nruns #8)IOType #9)MaxBufInMM
#10)MinBufInMM #11)WithMaxVal #12)WithMinVal #13)DFunction
#14)StripingUnit
0           0.000000           0 1 8 3 9 1 802 401 1 7 0 65536
                                                #
server: tcp://node01:22333
bytes used: 0Kbytes
server: tcp://node02:22333
bytes used: 0Kbytes
server: tcp://node03:22333
bytes used: 0Kbytes
server: tcp://node04:22333
bytes used: 0Kbytes
server: tcp://node05:22333
bytes used: 0Kbytes
server: tcp://node06:22333
bytes used: 0Kbytes
server: tcp://node07:22333
bytes used: 0Kbytes
server: tcp://node08:22333
bytes used: 8Kbytes
```

A.5.3 *Complete-1* Pattern

A.5.3.1 Parallel File Content, simple_stripe distribution function

```
#NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((7:0.100000,8:0.100000):0.100000,(16:0.100000,(2:0.100000,(3:0.100000,(13:0.
100000,(5:0.100000,(((11:0.100000,15:0.100000):0.100000,14:0.100000):0.100000,12
:0.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000,
(6:0.100000,9:0.100000):0.100000):0.100000,(10:0.100000,4:0.100000):0.100000,1:0
.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59641.147 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((6:0.100000,(16:0.100000,(((((8:0.100000,2:0.100000):0.100000,7:0.100000):0.
100000,4:0.100000):0.100000,(5:0.100000,(3:0.100000,9:0.100000):0.100000):0.
100000):0.100000,14:0.100000):0.100000):0.098402):0.099448,(10:0.100000,(13:0.
100000,((11:0.100000,15:0.100000):0.100000,12:0.100000):0.100000):0.100000):0.
100000,1:0.099446);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59474.926 2.897 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((8:0.100000,13:0.100000):0.100000,11:0.100000):0.100000,(((5:0.100000,12:0.
100000):0.100000,((7:0.100000,(3:0.100000,6:0.100000):0.100000):0.100000,2:0.
100000):0.100000):0.100000,10:0.100000):0.100000):0.100000,(4:0.100000,15:0.
100000):0.100000):0.100000,16:0.100000):0.100000,(14:0.100000,9:0.100000):0.
100000,1:0.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59956.233 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((5:0.100000,8:0.100000):0.100000,(4:0.100000,(13:0.100000,14:0.087779):0.
138379):0.088645):0.100000,10:0.100000):0.100000,(11:0.100000,((16:0.100000,15:0
.100000):0.100000,(3:0.100000,(6:0.100000,((7:0.100000,2:0.100000):0.100000,12:0
.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000,9:0.100000,1
:0.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -60013.210 2.915 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((4:0.100000,3:0.100000):0.100000,8:0.100000):0.100000,(((((2:0.100000,5:0.
100000):0.100000,13:0.100000):0.100000,(14:0.100000,6:0.100000):0.100000):0.
100000,10:0.100000):0.100000,7:0.100000):0.100000):0.100000,12:0.100000):0.
100000,(15:0.100000,11:0.100000):0.100000):0.100000,9:0.100000):0.100000,16:0.
100000,1:0.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59883.509 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((((7:0.100000,3:0.100000):0.100000,((13:0.100000,2:0.100000):0.100000,(4:0.
100000,5:0.100000):0.100000):0.100000):0.100000,(11:0.100000,(16:0.100000,14:0.
100000):0.100000):0.100000):0.100000,15:0.100000):0.100000,10:0.100000):0.100000
,6:0.100000):0.100000,(8:0.100000,9:0.100000):0.100000):0.100000,12:0.100000,1:0
.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59978.170 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((10:0.100000,(15:0.100000,(((2:0.100000,9:0.100000):0.100000,6:0.100000):0.
100000,((12:0.100000,(5:0.100000,8:0.100000):0.100000):0.100000,16:0.100000):0.
100000):0.100000):0.100000,7:0.100000):0.103340,((4:0.100000,(13:0.
100000,14:0.100000):0.100000):0.100000,(11:0.100000,3:0.100000):0.134124):0.
144228,1:0.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59692.885 2.982 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9564357168]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((5:0.100000,14:0.100000):0.100000,12:0.100000):0.100000,15:0.100000):0.
100000,(7:0.100000,4:0.100000):0.100000):0.100000,8:0.100000):0.131820,((6:0.
100000,(16:0.100000,3:0.100000):0.100000):0.148784,13:0.100000):0.099300):0.
100000,(2:0.100000,(9:0.100000,(11:0.100000,10:0.100000):0.100000):0.100000):0.
100000,1:0.100000);[ID: 9564357168]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59520.234 2.980 0.250000 0.250000 0.250000
0.250000
```

A.5.3.2 Information on Parallel File, simple_stripe distribution function

```
#Experiment's Id:     ParIO_Simple_65536_8_0_401_401_8_3_8_1_.pvsperf
#Conducted on:        Thu Sep 23 18:34:49 2010

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData    [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) StripingUnit         [ ]
#1)Ngen          #2)ExecTime          #3)OverallWrittenData
#4)MaxNgen  #5)Nprocs #6)Nchains #7)Nruns #8)IOType #9)MaxBufInMM
#10)MinBufInMM #11)WithMaxVal #12)WithMinVal #13)DFunction
#14)StripingUnit
0           0.000000           0 1 8 3 8 1 401 401 0 8 0 65536
                                                  #
server: tcp://node01:22333
bytes used: 0Kbytes
server: tcp://node02:22333
bytes used: 0Kbytes
server: tcp://node03:22333
bytes used: 0Kbytes
server: tcp://node04:22333
bytes used: 0Kbytes
server: tcp://node05:22333
bytes used: 8Kbytes
server: tcp://node06:22333
bytes used: 0Kbytes
server: tcp://node07:22333
bytes used: 0Kbytes
server: tcp://node08:22333
bytes used: 0Kbytes
```

A.5.3.3 Parallel File Content, varstrip distribution function

```
#NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((((3:0.100000,(7:0.100000,13:0.100000):0.100000):0.100000,(2:0.100000,8:0.
100000):0.100000):0.100000,6:0.100000):0.100000,9:0.100000):0.100000,11:0.100000
):0.100000,16:0.100000):0.100000,15:0.100000):0.100000,(10:0.100000,(4:0.100000,
(12:0.100000,(14:0.100000,5:0.100000):0.100000):0.100000):0.100000):0.100000,1:0
.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
```

```
1 -59582.035 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((((9:0.100000,((7:0.100000,16:0.100000):0.066799,13:0.100000):0.096944):0.
089945,6:0.100000):0.100000,11:0.100000):0.100000,5:0.100000):0.100000,3:0.
100000):0.100000,(12:0.100000,(2:0.100000,15:0.100000):0.100000):0.100000):0.
100000,(14:0.100000,10:0.100000):0.100000):0.100000,8:0.100000):0.100000,4:0.
100000,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59858.754 2.854 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((15:0.100000,16:0.100000):0.100000,(((9:0.100000,13:0.100000):0.100000,(7:0.
100000,10:0.100000):0.100000):0.100000,11:0.100000):0.100000):0.100000,(4:0.
100000,(2:0.100000,14:0.100000):0.100000):0.100000):0.100000,((12:0.141216,(8:0.
100000,(5:0.100000,6:0.100000):0.100000):0.100000):0.071414,3:0.100000):0.145419
,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59539.721 2.958 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((3:0.100000,(((4:0.100000,(16:0.100000,((14:0.100000,11:0.100000):0.100000,9:0.
100000):0.100000):0.100000):0.100000,7:0.100000):0.100000,((8:0.100000,15:0.
100000):0.100000,((12:0.100000,6:0.100000):0.100000,((5:0.100000,2:0.100000):0.
100000,10:0.100000):0.100000):0.100000):0.100000):0.100000):0.100000,13:0.100000
,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59897.228 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((((2:0.100000,14:0.100000):0.100000,3:0.100000):0.100000,11:0.100000):0.
100000,5:0.100000):0.100000,(15:0.100000,16:0.100000):0.100000):0.100000,13:0.
100000):0.100000,8:0.100000):0.100000,9:0.100000):0.100000,((4:0.100000,7:0.
100000):0.100000,((10:0.100000,12:0.100000):0.100000,6:0.100000):0.100000):0.
100000,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59741.747 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((10:0.100000,15:0.100000):0.100000,(16:0.100000,((6:0.100000,11:0.100000):0.
100000,(2:0.100000,13:0.100000):0.100000):0.100000):0.100000):0.100000,7:0.
100000):0.100000,((3:0.100000,14:0.100000):0.100000,((((8:0.100000,9:0.100000):0
.100000,5:0.100000):0.100000,4:0.144079):0.157540,12:0.100000):0.066826):0.
100000,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59556.943 2.968 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((((13:0.100000,(7:0.100000,((5:0.100000,(12:0.100000,(6:0.100000,8:0.100000
):0.100000):0.100000):0.100000,10:0.100000):0.100000):0.100000):0.100000,(11:0.
100000,15:0.100000):0.100000):0.100000,3:0.100000):0.100000,14:0.100000):0.
100000,16:0.100000):0.100000,2:0.100000):0.100000,9:0.100000):0.100000,4:0.
100000,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59829.956 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9399782165]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((13:0.100000,15:0.100000):0.100000,((12:0.100000,4:0.100000):0.100000,((14:0.
100000,11:0.100000):0.100000,(((7:0.100000,3:0.100000):0.100000,8:0.100000):0.
100000,16:0.100000):0.100000):0.100000):0.100000):0.100000,(6:0.100000,(9:0.
100000,(2:0.100000,10:0.100000):0.100000):0.100000):0.100000):0.100000,5:0.
100000,1:0.100000);[ID: 9399782165]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59881.849 2.900 0.250000 0.250000 0.250000
0.250000
```

A.5.3.4 Information on Parallel File, varstrip distribution function

```
#Experiment's Id:     ParIO_Varstrip_701_701_8_0_401_401_8_4_8_1_.pvsperf
#Conducted on:        Thu Sep 23 19:15:43 2010
```

```
#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData     [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) Max varstrip chunk size   [<#MaxVsCS> ]
#15) Min varstrip chunk size   [<#MinVsCS> ]
#1)Ngen          #2)ExecTime          #3)OverallWrittenData
#4)MaxNgen  #5)Nprocs #6)Nchains #7)Nruns #8)IOType #9)MaxBufInMM
#10)MinBufInMM #11)WithMaxVal #12)WithMinVal #13)DFunction
#14)MaxVsCS #15)MinVsCS
0           0.000000           0 1 8 4 8 1 401 401 0 8 1 701 701
dist_name = varstrip_dist
dist_params:
0:701;1:701;2:701;3:701;4:701;5:701;6:701;7:701
Metadataserver: tcp://node06:22333
Number of datafiles/servers = 8
Datafile 0 - tcp://node01:22333, handle: 5188146770730808832 (47fffffffffff600.bstream)
Datafile 1 - tcp://node02:22333, handle: 5764607523034232832 (4ffffffffffff800.bstream)
Datafile 2 - tcp://node03:22333, handle: 6341068275337656832 (57fffffffffffa00.bstream)
Datafile 3 - tcp://node04:22333, handle: 6917529027641080832 (5ffffffffffffc00.bstream)
Datafile 4 - tcp://node05:22333, handle: 7493989779944504832 (67fffffffffffe00.bstream)
Datafile 5 - tcp://node06:22333, handle: 8070450532247925237 (6ffffffffffff1f5.bstream)
Datafile 6 - tcp://node07:22333, handle: 8646911284551348736 (77fffffffffff200.bstream)
Datafile 7 - tcp://node08:22333, handle: 9223372036854772736 (7ffffffffffff400.bstream)
```

A.5.4 *Partial-1* Pattern

A.5.4.1 Parallel File Content

```
#NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((6:0.100000,14:0.100000):0.100000,(4:0.100000,(((12:0.100000,11:0.100000):0.
100000,15:0.063617):0.121922,3:0.100000):0.147583):0.100000):0.100000,10:0.
100000):0.100000,(16:0.100000,(2:0.100000,(8:0.100000,(5:0.100000,((13:0.100000,
9:0.100000):0.100000,7:0.100000):0.100000):0.100000):0.100000):0.100000):0.
100000,1:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59623.159 2.933 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((((8:0.100000,10:0.100000):0.100000,16:0.100000):0.100000,((9:0.100000,11:0.
100000):0.100000,12:0.100000):0.100000):0.100000,(6:0.136832,((2:0.100000,(15:0.
100000,((7:0.100000,4:0.100000):0.100000,14:0.100000):0.100000):0.100000):0.
068446,3:0.100000):0.131484):0.100000):0.100000,13:0.100000):0.100000,5:0.100000
,1:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59588.682 2.937 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
```

```
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((((5:0.100000,(14:0.100000,(15:0.100000,10:0.100000):0.100000):0.100000):0.
100000,(8:0.100000,((6:0.100000,(13:0.147512,(16:0.100000,(12:0.100000,(2:0.
100000,4:0.100000):0.100000):0.100000):0.100000):0.157187):0.149553,(3:0.100000,
9:0.100000):0.100000):0.100000):0.100000):0.100000,7:0.100000):0.100000,11:0.
100000,1:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59258.991 3.054 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
((3:0.100000,(10:0.100000,(15:0.100000,(14:0.100000,((8:0.100000,(9:0.100000,7:0
.100000):0.100000):0.100000,(4:0.100000,11:0.100000):0.100000):0.100000):0.
100000):0.100000):0.100000):0.067528,(((((13:0.100000,5:0.100000):0.100000,12:0.
100000):0.100000,6:0.100000):0.100000,2:0.100000):0.100000,16:0.198973):0.031958
,1:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59561.675 2.898 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(11:0.100000,((9:0.100000,6:0.100000):0.100000,(((16:0.100000,14:0.100000):0.
100000,2:0.100000):0.100000,(3:0.100000,(((5:0.100000,12:0.100000):0.100000,4:0.
100000):0.100000,(13:0.100000,(8:0.100000,(7:0.100000,(10:0.100000,15:0.100000):
0.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000,1
:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59959.488 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((5:0.100000,9:0.100000):0.100000,10:0.100000):0.100000,2:0.100000):0.100000,
12:0.100000):0.100000,(((15:0.100000,4:0.100000):0.100000,3:0.100000):0.100000,(
(13:0.100000,14:0.100000):0.100000,(8:0.100000,(16:0.100000,(11:0.100000,(6:0.
100000,7:0.100000):0.100000):0.100000):0.100000):0.100000):0.100000):0.100000,1:
0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59853.261 2.900 0.250000 0.250000 0.250000
0.250000 #NEXUS
[ID: 9386433625]
begin trees;
   translate
       1 1,
       2 2,
       3 3,
       4 4,
       5 5,
       6 6,
       7 7,
       8 8,
       9 9,
      10 10,
      11 11,
      12 12,
      13 13,
      14 14,
      15 15,
      16 16;
   tree rep.1 =
(((((((15:0.100000,4:0.100000):0.100000,(5:0.100000,(13:0.100000,((3:0.100000,8:
0.100000):0.100000,12:0.100000):0.100000):0.100000):0.100000):0.100000,(9:0.
100000,((11:0.100000,2:0.100000):0.100000,14:0.100000):0.100000):0.100000):0.
100000,10:0.128103):0.112084,16:0.094770):0.100000,7:0.100000):0.100000,6:0.
100000,1:0.100000);[ID: 9386433625]
Gen LnL TL pi(A) pi(C) pi(G) pi(T)
1 -59540.571 2.935 0.250000 0.250000 0.250000
0.250000
```

A.5.4.2 Information on Parallel File

```
#Experiment's Id:     ParIO_Simple_65536_1_7_0_401_7_3_8_1_.pvsperf
#Conducted on:        Thu Sep 23 18:39:59 2010

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData    [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) StripingUnit          [ ]
#1)Ngen         #2)ExecTime          #3)OverallWrittenData
#4)MaxNgen  #5)Nprocs #6)Nchains #7)Nruns #8)IOType #9)MaxBufInMM
#10)MinBufInMM #11)WithMaxVal #12)WithMinVal #13)DFunction
#14)StripingUnit
0           0.000000           0 1 8 3 7 1 401 0 7 1 0 65536
                                                   #
server: tcp://node01:22333
bytes used: 0Kbytes
server: tcp://node02:22333
bytes used: 0Kbytes
server: tcp://node03:22333
bytes used: 0Kbytes
server: tcp://node04:22333
bytes used: 0Kbytes
server: tcp://node05:22333
bytes used: 0Kbytes
server: tcp://node06:22333
bytes used: 8Kbytes
server: tcp://node07:22333
bytes used: 0Kbytes
server: tcp://node08:22333
bytes used: 0Kbytes
```

A.5.5 Parameters' Database

For the set of experiments I created a small database. Figure A.2 shows the corresponding entity relationship model diagram. Furthermore, table A.7 shows these parameters in a stack manner.

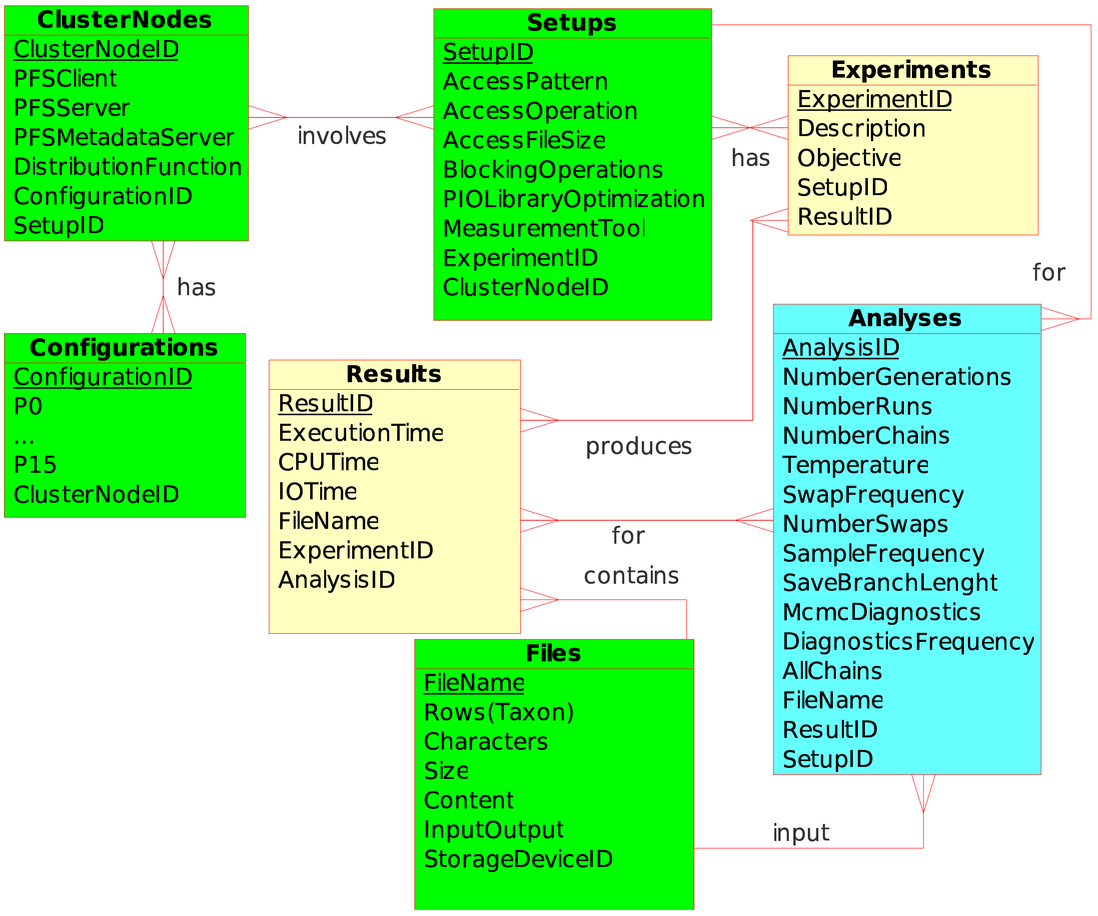

Figure A.2: Entity relationship diagram model representing the database of the evaluation experiments

A.6 Definitions

A.6.1 Computer Architecture

Definition A.1 *A Monoprocessor System is a basic computing system consisting of one processor and one memory hierachy, which consists of registers, caches, main memory, and one hard disk secondary storage.*

A.6.2 Parallel File Information

This section includes information on the generated parallel file while conducting the evaluation.

Table: Setups	**DB Field Name**
Application Level	
Access Pattern:	AccPa9
Access Size:	AccSi9
Operation:	AccOp9
File Size	AccFS9
Partition Load Balance:	PLoBa9
Blocking non-blocking Operations:	BloOp9
PIO Library Level	
PIO Library Optimization:	PioLO9
Parallel File System Level	
Process per Compute Node:	ProCN9
Interconnect between CN and I/O N:	ICIon9
Type of Distribution Function:	DisFu9
Physical Level	
Existing load balance:	ExLBa9
Storage Capacity Homogeneity:	StCaH9
Measurement Tools	
Measurement Tool	MeTol9
Table: Analyses	**DB Field Name**
IO relevant Application's parameters	
Input File	
Dimensions:	Dimen9
`mcmc` **Related Parameters**	
Ngen (Number of Generations):	NGene9
Nruns (Number of Runs):	NRuns9
Nchains (Number of Chains):	NChai9
Temp (Temperature):	Tempe9
Swapfreq (Swap Frequency):	SwapF9
Nswaps (Number of Swaps):	NSwap9
Samplefreq (Sample Frequency):	SFreq9
Savebrlens (Save Branch Lengths):	SBLen9
`mcmc` **Diagnostics Related Parameters**	
mcmcdiagn (`mcmc` Diagnostics):	McmcD9
Diagnfreq (Diagnostic Frequency):	DiagF9
Allchains (All Chains):	AChaD9
Table: Results	
Application Level	
Execution Time:	ExTim9
CPU Time:	CpTim9
IO Time:	I/O Tim9
Physical Level	
Final Load Balance:	FiLoB9
Application related results	
Generated File Size:	GFiSi9

Table A.7: Measurements parameters and their database field names

A.6.2.1 Data File and I/O Nodes Correspondency

This is one sample of the parallel file's information provided by `pvfs2-viewdist`. This corresponds to a file generated with the varstrip distribution function with a *varstrip chunk size* of 3.62 MB.

```
#$ Author: hipolito.vasquez@informatik.uni-heidelberg.de$
#$ Date:$
#Hipolito Vasquez Lucas's PhD Thesis Experiment's File
#Experiment's Id:
#ParIO_Varstrip_3803000_3803000_8_0_2205390_2205390_8_4_8_8_.pvsperf
#Conducted on:        Mon Aug 23 05:09:17 2010

#Setup parameters:
#1) Ngen                  [1 ... [10000 | 100000]
#2) ExecTime
#3) OverallWrittenData    [1 ... 2000000000]
#4) MaxNgen               [10000, 100000, 1000000]
#5) Nprocs                [2^n, n=3 ...]
#6) Nchains               [n*default, n = 1,2,.. ]
#7) Nruns                 [2^n, n = 2, ...]
#8) Type of I/O             [0 = SIO, 1 = ParIO]
#9) Max Buffer size in MM before write [ ]
#10) Min Buffer size in MM before write [ ]
#11) LB: Number of Procs with Max Buffer [<#WithMaxVal>]
#12) LB: Number of Procs with Min Buffer [<#WithMinVal>]
#13) Distribution Function: [0 = SS, 1 = VS]
#14) Max varstrip chunk size   [<#MaxVsCS> ]
#15) Min varstrip chunk size   [<#MinVsCS> ]
#1)Ngen          #2)ExecTime          #3)OverallWrittenData
#4)MaxNgen  #5)Nprocs #6)Nchains #7)Nruns #8)IOType #9)MaxBufInMM
#10)MinBufInMM #11)WithMaxVal #12)WithMinVal #13)DFunction
#14)MaxVsCS #15)MinVsCS
0           0.000000           0 8 8 4 8 1 2205390 2205390 0 8 1 3803000 3803000
dist_name = varstrip_dist
dist_params:
0:3803000;1:3803000;2:3803000;3:3803000;4:3803000;5:3803000;6:3803000;7:3803000
Metadataserver: tcp://node06:22333
Number of datafiles/servers = 8
Datafile 0 - tcp://node01:22333, handle: 5188146770730808832 (47fffffffffff600.bstream)
Datafile 1 - tcp://node02:22333, handle: 5764607523034232832 (4ffffffffffff800.bstream)
Datafile 2 - tcp://node03:22333, handle: 6341068275337656832 (57fffffffffffa00.bstream)
Datafile 3 - tcp://node04:22333, handle: 6917529027641080832 (5ffffffffffffc00.bstream)
Datafile 4 - tcp://node05:22333, handle: 7493989779944504832 (67fffffffffffe00.bstream)
Datafile 5 - tcp://node06:22333, handle: 8070450532247925237 (6ffffffffffff1f5.bstream)
Datafile 6 - tcp://node07:22333, handle: 8646911284551348736 (77fffffffffff200.bstream)
Datafile 7 - tcp://node08:22333, handle: 9223372036854772736 (7ffffffffffff400.bstream)
```

Bibliography

[ADHR04] Gautam Altekar, Sandhya Dwarkadas, John P. Huelsenbeck and Fredrik Ronquist. Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. *Bioinformatics* 20(3), 2004, Pag. 407–415.

[AlGo89] G. S. Almasi and A. Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA. 1989.

[BCST$^+$08] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press, Pag. 44:1–44:12.

[CDKM$^+$01] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald and Ramesh Menon. *Parallel Programming in OpenMP.* Academic Press, San Diego, CA. 2001.

[ChPa90] Peter M. Chen and David A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News.* ACM, June 1990, Pag. 322–331.

[CoFe96] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems* 14(3), 1996, Pag. 225–264.

[CoLR90] Thomas H. Cormen, E. Leiserson, Charles and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press. 1990.

[Cort99] Toni Cortes. Software RAID and Parallel Filesystems. In Rajkumar Buyya, Editor, *High Peformance Cluster Computing*, Pag. 463–496. Prentice Hall PTR, 1999.

[Croc89] T. W. Crockett. File concepts for parallel I/O. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, New York, NY, USA, 1989. ACM, Pag. 574–579.

[CuSG99] David E. Culler, Jaswinder Pal Singh and Anoop Gupta. *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA. 1999.

[DEKM98] Richard Durbin, Sean R. Eddy, Anders Krogh and Graeme Mitchison. *Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press. 1998.

[DHJL$^+$97] DasGupta, He, Jiang, Li, Tromp and Zhang. On Distances between Phylogenetic Trees. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[Dong05] T.; Simon H.; Strohmaier E. Dongarra, J.; Sterling. High-performance computing: clusters, constellations, MPPs, and future directions. *IEEE Computational Science and Engineering* 7(2), March-April 2005, Pag. 51–59.

[DoSe98] Kevin Dowd and Charles Severance. *High performance computing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA. 1998.

[EGVL05] Tobias Eberle, Frederik Gruell, Hipolito Vasquez and Thomas Ludwig. Distribution functions in PVFS2, hints in MPICH2 and performance measurements, April 31 2005.

[FeCB06] Xizhou Feng, Kirk W. Cameron and Duncan A. Buell. PBPI: a High Performance Implementation of Bayesian Phylogenetic Inference. In *SC2006*, SC '06. IEEE, November 2006.

[Fels83] J. Felsenstein. Statistical inference of phylogenies. *Journal of the Royal Statistical Society* Band 146, 1983, Pag. 246–272.

[FoCo94] G. C. Fox and P. D. Coddington. Parallel Computers and Complex Systems, August 31 1994.

[GeSc02] Al Geist and Stephen Scott. Parallel Programming with PVM. In Thomas Sterling, Editor, *High Performance Mass Storage and Parallel I/O: Beowulf Cluster Computing with Linux.* MIT Press, Cambridge, Mass., 2002. chap. 11.

[GoDW09] J. Gonzalez, H. Duettman and M. Wink. Phylogenetical relationships based on two mitochondrial genes and hybridization patterns in Anatidae. *Journal of Zoology* (279), 2009, Pag. 310–318.

[GrLT99] William Gropp, Ewing Lusk and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA. 1999.

[HiHJ05] David M. Hillis, Tracy A. Heath and Katherine St. John. Analysis and Visualization of Tree Space. *Systematic Biology* Band 54, 2005, Pag. 471–482.

[HLMR02] John P. Huelsenbeck, Bret Larget, Richard E. Miller and Fredrik Ronquist. Potential Applications and Pitfalls of Bayesian Inference of Phylogeny. *Systematic Biology* 51(5), 2002, Pag. 673–688.

[HoLe03] Mark Holder and Paul O. Lewis. Phylogeny Estimation: Traditional and Bayesian Approaches. *Nature Reviews Genetics* Band 4, 2003, Pag. 275–284.

[HRNB01] John P. Huelsenbeck, Fredrik Ronquist, Rasmus Nielsen and Jonathan P. Bollback. Bayesian Inference of Phylogeny and Its Impact on Evolutionary Biology. *Science's Compass Review* Band 294, 2001, Pag. 2310–2314.

[HsSm04] W. Hsu and A. J. Smith. The performance impact of I/O optimizations and disk improvements. *IBM J. Res. Dev.* 48(2), 2004, Pag. 255–289.

[HuBo01] John P. Huelsenbeck and Jonathan P. Bollback. Empirical and Hierarchical Bayesian Estimation of Ancestral States. *Systematic Biology* 50(3), 2001, Pag. 351–366.

[HuRo01] John P. Huelsenbeck and Fredrik Ronquist. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics* 17(8), 2001, Pag. 754–755.

[IIRo99] Walter B. Ligon III and Robert B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

[IMOS+04] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder and Walter F. Tichy. Integrating Collective IO and Cooperative Caching into the "Clusterfile" Parallel File System. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, New York, NY, USA, 2004. ACM, Pag. 58–67.

[IsTi03] Florin Isaila and Walter F. Tichy. Clusterfile: a flexible physical layout parallel file system. *Concurrency and Computation* 15(7/8), 2003, Pag. 653–679.

[JERC+01] James V. Huber Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien and David S. Blumentahl. PPFS: A High Performance Portable Parallel File System. In Hai Jin, Toni Cortes and Rajkumar Buyya, Editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE/Wiley Press, New York, 2001. Chap. 22.

[KLSS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele and M. Zosel, Editors. *High Performance Fortran Handbook*. MIT Press, Cambridge, MA, USA. 1994.

[KuLV04] Julian Kunkel, Thomas Ludwig and Hipolito Vasquez. Dateisystem fuer parallele Systeme: PVFS: Version 2. *iX, Magazin fuer Professionelle Informationstechnik* (6), June 2004, Pag. 110–113.

[LeKa93] Edward K. Lee and Randy H. Katz. The Performance of Parity Placements in Disk Arrays. *IEEE Transactions on Computers* 42(6), June 1993, Pag. 651–664.

[Lesk03] Arthur M. Lesk. *Bioinformatik*. Spektrum, Akad. Verl. 2003.

[Leve10] Adam Leventhal. Triple-parity RAID and beyond. *Communications of the ACM* Band 53, January 2010, Pag. 58–63.

[LiRo96] W. B. Ligon and R. B. Ross. Implementation and Performance of a Parallel File System for High Performance Distributed Applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996, Pag. 471–480.

[LiRo01] Walt Ligon and Rob Ross. PVFS: Parallel Virtual File System. In Thomas Sterling, Editor, *Beowulf Cluster Computing with Linux*, Scientific and Engineering Computation, Chapter 17, Pag. 391–430. The MIT Press, Cambridge, Massachusetts, November 2001.

[LMRC04] Rob Latham, Neil Miller, Robert Ross and Phil Carns. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld* 2(1), January 2004.

[LuLu05] Hipolito Vasquez Lucas and Thomas Ludwig. Hint Controlled Distribution with Parallel File Systems. In *European PVM/MPI Users' Group Meeting*. Springer, September 2005, Pag. 110–118.

[Madh97] Tara Maja Madhyastha. *Automatic Classification of Input/Output Acess Patterns*. Dissertation, Champaign, IL, USA, 1997.

[MaRe97] Tara M. Madhyastha and Daniel A. Reed. Exploiting Global Input/Output Access Pattern Classification. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, November 1997. ACM Press.

[MaRe04] Tara M. Madhyastha and Daniel A. Reed. Learning to Classify Parallel I/O Access Patterns. In Daniel A. Reed, Editor, *Scalable Input/Output*, Pag. 201–231. The MIT Press, 2004.

[MaSM97] David R. Maddison, David L. Swofford and Wayne P. Maddison. NEXUS: AN EXTENSIBLE FILE FORMAT FOR SYSTEMATIC INFORMATION. *Systematic Biology* 46(4), 1997, Pag. 590–621.

[May01] John M. May. *Parallel I/O for high performance computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2001.

[MiKa91] Ethan L. Miller and Randy H. Katz. Input/Output Behavior of Supercomputer Applications. In *Proceedings of Supercomputing '91*, November 1991, Pag. 567–576.

[MoTr03] Robert J. T. Morris and Brian J. Truskowski. The evolution of storage systems. *IBM SYSTEMS JOURNAL* 42(2), 2003, Pag. 205–217.

[MPI 08] MPI Forum: The Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1, June 23 2008.

[MTHC$^+$08] Jeanna Matthews, Sanjeev Trika, Debra Hensgen, Rick Coulson and Knut Grimsrud. Intel Turbo Memory: Nonvolatile Disk Cache in the Storage Hierarchy of Mainstream Computer Systems. *ACM Transactions on Storage* 4(2), May 2008.

[NiLo97] Bill Nitzberg and Virginia Lo. Collective Buffering: Improving Parallel I/O. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997, Pag. 148–157.

[PaCh98] David A. Patterson and Peter M. Chen. Storage Performance - Metrics and Benchmarks, January 1998.

[PSSM$^+$10] Seon-yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng and Joonwon Lee. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Computer Architecutere Letters* 9(1), January–June 2010.

[Pura96] Apratim Purakayastha. *Characterizing and Optimizing Parallel File Systems*. Dissertation, Durham, North Carolina, USA, June 1996.

[PuVe10] Ruprecht-Karls-Universitaet Heidelberg Parallele und Verteilte Systeme Group, Institut fuer Informatik. http://ludwig9.informatik.uni-heidelberg.de/wiki/index.php/Cluster:Hardware, February 2010.

[Quin03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGrawHill, Singapore. 2003.

[RCCK$^+$95] Daniel A. Reed, Charles Catlett, Alok Choudhary, David Kotz and Marc Snir. Parallel I/O: Getting Ready for Prime Time. *IEEE Parallel and Distributed Technology*, Summer 1995, Pag. 64–71.

[RKPH04] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost and Richard Hedges. The Parallel Effective I/O Bandwidth Benchmark: b_eff_io. In Christophe Cerin and Hai Jin, Editors, *Parallel I/O for Cluster Computing*, Chapter 4, Pag. 107–132. Kogan Page Ltd., February 2004.

[RoHu03] Fredrik Ronquist and John P. Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* 19(12), 2003, Pag. 1572–1574.

[ScHa02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02. USENIX Association, 2002.

[Simi00] Huseyin Simitci. *Adaptive Disk Striping for Parallel Input/Output*. Dissertation, Champaign, IL, USA, 2000.

[StAl10] Alexandros Stamatakis and Nikolaos Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics* Band 26, 2010, Pag. i132–i139.

[ThCh95] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. Technical Report, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Revised November 1995.

[ThGL98] Rajeev S. Thakur, William Gropp and Ewing Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proceedings of Supercomputing'98*, Orlando, FL, November 1998. ACM SIGARCH and IEEE.

[ThGL99a] Rajeev Thakur, William Gropp and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 1999, Pag. 182–189.

[ThGL99b] Rajeev Thakur, William Gropp and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS-99)*, New York, May 1999. ACM Press, Pag. 23–32.

[TKRM08] Taeho Kgil, David Roberts and Trevor Mudge. Improving NAND Flash Based Disk Caches. *SIGARCH Computer Architecture News* Band 36, June 2008.

[Volk99] J. Volker. Parallelrechner. In P. Rechenberg and G. Pomberger, Editors, *Informatik-Handbuch*, Chapter 5, Pag. 363–380. Hanser Verlag, München, 1999.

[WLRR03] Walt Ligon and Rob Ross. Parallel I/O and the Parallel Virtual File System. In William Gropp, Ewing Lusk and Thomas Sterling, Editors, *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge, Mass., 2003. chap. 19.

[WoGr97] David E. Womble and David S. Greenberg. Parallel I/O: An Introduction. *Parallel Computing* 23(4-5), 1997, Pag. 403–417.

[YaRa97] Ziheng Yang and Bruce Rannala. Bayesian Phylogenetic Inference Using DNA Sequences: A Markov Chain Monte Carlo Method. *Molecular Biology and Evolution* 14(7), 1997, Pag. 717–724.

[Zeer05] Amir Rais Zeervi. *Implementierung einer parallelen Ausgabe fuer das Stammbaumprogramm MrBayes*. Bachelor Thesis, Ruprecht-Karls Universitaet Heidelberg, Institut fuer Informatik, Arbeistsgruppe Parallele und Verteilte Systeme, Heidelberg, Germany, October 2005.