MPI-based Solution for Efficient Data Access in Java HPC

Aidan Fries[*] [†] [‡], Jordi Portell[*] [†] [‡], Yago Isasi[*] [†] [‡], Javier Castañeda[*] [†] [‡], Raül Sirvent[§], and Guillermo L. Taboada[¶]
[*]Department of Astronomy and Meteorology, University of Barcelona, Barcelona, (Spain)
[†]Institute for Space Studies of Catalonia (IEEC), Barcelona, (Spain)
[‡]Institute of Cosmos Sciences (ICC), Barcelona, (Spain)
afries@am.ub.es, jportell@am.ub.es, yisasi@am.ub.es, jcastapo@am.ub.es
[§]BSC-CNS Barcelona Supercomputing Center, Barcelona, (Spain). raul.sirvent@bsc.es
[¶]Computer Architecture Group, University of A Coruña, A Coruña (Spain). taboada@udc.es

***Abstract*—Efficient data access is extremely important for many applications in HPC. In many cases, processes running in one node will need to access data held in another node, as well as access data held in some central storage device. In I/O-intensive applications, accessing data not held in the local node can become a bottleneck, especially in cases where the remotely stored data is accessed repeatedly, and when accessing data from virtual machines such as in Java. To address this issue, we have designed and implemented a data cache system, which offers efficient data access to Java applications in HPC. This system, which we call MPJ-Cache, makes use of a Java-based message-passing implementation, such as F-MPJ, and it provides a high-level API for the accessing of data. MPJ-Cache can improve the performance of I/O operations for certain Java applications in HPC by reducing significantly the I/O overhead. In this paper, we describe MPJ-Cache, including the data communication layer, as well as the caching features of the system, and we show how it can be used to improve I/O performance for HPC applications. The comparative performance evaluation of this system against the file system of the MareNostrum supercomputer (Barcelona Supercomputing Center) has shown important performance benefits. Finally, we also show the impact of this solution on a challenging problem such as the data processing system for the ESA Gaia space mission.**

***Keywords*-Java Communications; Data Cache; F-MPJ; Gaia; GPFS; Myrinet**

I. Introduction

A typical distributed-memory HPC environment includes many computing nodes, each node containing one or more processors and each processor containing one or more cores. Each node may be connected to every other node over some high-speed, low latency network, while each node may also be connected to a central storage device.

In recent years, the most significant trends in distributed-memory HPC environments have included the move towards a larger number of cores per processor, an increased awareness of the issue of power consumption and a massive growth in the volume of data being handled. The increase in the number of available cores will typically lead to an increase in the number of parallel processes running in a computing node, and therefore an increase in the number of processes accessing data. These factors combine to make the issue of efficient data access extremely important.

In the case of I/O-intensive applications, where the processes running in the computing nodes may need to access data stored in the shared storage device, I/O can become a significant factor affecting the overall performance of the application. The importance of I/O efficiency increases with the number of parallel processes, and the problem is further amplified if the data is accessed repeatedly.

MPI continues as the leading approach for implementing inter-process communication in distributed memory environments, offering point-to-point as well as collective communication functions. However, despite the strengths and maturity of MPI, writing applications that can take full advantage of the resources of a distributed environment, such as a cluster of computing nodes, can be quite difficult. In the case of complex applications, where there may be strong dependencies between processes running in different nodes, the programming effort required to implement the communication can be considerable and is often bug-prone.

The Client-Server model is a long established and intuitive approach for making data available to a group of consumers. In order to avoid the potential bottleneck issue associated with many processes accessing a shared storage device, we designed a data cache system, which we call MPJ-Cache. This system involves the execution of Server processes in some of the available nodes, which maintain a cache of data; while the communication between the Clients and the Servers is built on top of an implementation of Message Passing in Java (MPJ), and can take advantage of any high-speed network support provided by the underlying MPJ application.

Gaia [1] is a European Space Agency mission, whose primary objective is to chart a 3D map of around one billion stars in our Galaxy. The Data Analysis and Processing Consortium (DPAC) is the organisation with responsibility to process the Gaia data. It is a policy within DPAC that all software should be written in Java. The selection of Java for this kind of large, scientific, data processing project is relatively uncommon. Therefore, the Gaia data processing represents an opportunity to study the use of Java to implement a scientific processing pipeline, and its execution in a HPC environment. In this paper, we will discuss two DPAC applications, both of which are I/O-intensive and could benefit from the use of MPJ-Cache.

The first of these applications is GAia System Simulator (GASS), which simulates the raw telemetry stream that will be generated by the satellite during its mission lifetime. Secondly, we will discuss Intermediate Data Updating (IDU), which is one of the applications that will process real Gaia data.

The rest of this paper is organised as follows. In Section II we give a general summary of the current status of Java in HPC, and in particular, on the status of Java-based data communication in HPC. In Section III, we describe the I/O problems faced by I/O-intensive Java applications in HPC, specifically the issue of I/O bottlenecks. Our solution to the aforementioned problem is presented in Section IV, where we briefly describe the communication layer of MPJ-Cache, as well as the caching features of the system. In Section V, we describe a set of tests designed to compare the performance of our data cache against direct access of GPFS. The results of these tests are given in Section VI. Finally, in Section VII, we give our conclusions, and mention some further work that we intend to carry out in this area.

II. Related work

Despite the continuing popularity of Java in general computing, and the many independent projects which have developed extensions and libraries for Java applications in HPC, the use of Java in HPC and by the scientific communities remains relatively low. We conducted an investigation into the developments for Java in HPC over the last 15 years. We looked at papers published, as well as libraries and tools that were developed to aid the use of Java in HPC. It is clearly evident that there was a very significant level of work in this area, roughly speaking, during the period 1999 to 2003. This period corresponds with the activity of the Java Grande Forum [5], a initiative amongst the Java scientific community to investigate possible additions to the Java language, and to encourage its use in HPC and scientific communities. However, since then, the number of papers and projects in this area has reduced significantly, and the number of projects which are actively developing or supporting libraries for Java in HPC now appears quite low.

It is almost certainly the case that part of the reason for the slow adoption of Java in HPC is simply due to the inertia of moving from the languages which have traditionally being used in the HPC and scientific computing communities such as Fortran and C/C++. It is also due to the initial reputation that Java acquired as providing poor performance due to it being an interpreted language. Finally, part of the problem may also be a lack of reliable and supported HPC-specific Java libraries in areas such as data communication.

COMP Superscalar (COMPSs) [4] is a runtime environment which allows for the automatic parallelisation of serial applications, for their execution in a HPC environment. This process involves COMPSs analysing the application; identifying tasks of a certain granularity within that application; determining the dependencies between these tasks; generating a task graph; and where possible, executing tasks in parallel. COMPSs also manages the input and output for each task. It is a powerful tool, allowing seemingly serial applications to become parallel, and taking advantage of the available HPC resources. However, it does not deal with the potential bottleneck associated with many processes accessing a shared storage device. Although COMPSs removes the issue from the concern of user applications, COMPSs itself may encounter I/O issues if it tries to distribute some data to a large number of nodes. Also, there are circumstances when application developers may prefer to maintain control of the actual flow of data around the available hardware. Therefore, we believe that another solution, dealing with the potential I/O bottleneck and providing application developers with a high-level, HPC-specific data-access API would be a useful contribution to Java in HPC.

Based on our investigation into the available libraries which provide data communication to Java applications in distributed memory environments, the 3 most commonly used options are: Sockets, RMI, and Message Passing. In this work we decided to focus our work on the MPJ approach as it has been reported to provide the highest performance in HPC applications on low latency networks [7]. MPJ can be implemented in a number of ways, including the use of Java RMI, Java Native Interface (JNI) — to call an underlying native message passing library, and also through the use of Java sockets. Each approach has advantages and disadvantages in the areas of efficiency, portability and complexity. The use of RMI assures portability, as it is a pure Java solution. However, it may not be the most efficient solution in the presence of any high speed communication hardware, which RMI might not take full advantage of. The use of JNI allows for the efficient use of high-speed networks using native libraries, however it has portability problems. Finally, the implementation of MPI using Java sockets requires a considerable development effort. Fortunately, we have identified implementations of MPJ: MPJ Express [6] and F-MPJ [8], which are being actively developed and supported. MPJ Express and F-MPJ both implement the same specification of MPJ — `mpiJava` 1.2 [2] — so it is easy to swap between these implementations.

III. The problem

An important trend in HPC is the move towards an ever increasing number of cores per processor, and consequentially an increase in the number of processes that are typically executed per computing node. The volume of data that these processes must handle is also increasing. Despite the availability of high performance storage devices and networks, the accessing of data held in shared storage devices can act as a bottleneck in the processing of data, if there are a large number of processes accessing the data and if it is accessed repeatedly.

The objective of this work is to find an efficient and scalable mechanism that allows for a large number of processes running in separate computing-nodes to be able to access some remote data, where the I/O performance achieved is minimally affected by the number of processes accessing the data.

A. Gaia data processing in MareNostrum

The following two applications, both of which are part of the Gaia data processing task and will run on the MareNostrum supercomputer at the Barcelona Super Computing Center (BSC), represent different use-cases where the I/O problem described in the previous section emerges as an issue.

1) GASS: During the simulation of the telemetry stream, GASS has to process over 1 billion stellar sources. These simulations require the use of many worker processes (thus far, simulations involving over 3000 cores working simultaneously have been executed). These worker processes need to repeatedly access a set of shared data-files during the execution of GASS (such as spectra and instrument calibration files). If all of the executing worker processes simply access these files directly on the GPFS, then its performance decreases to unacceptable I/O response times.

2) IDU: IDU itself is composed of several independent tasks, which run in a particular sequence and which are effectively independent serial applications. However, some of these tasks have dependencies on the output from other tasks, and the input for one task may come from the output of another task which was executed in a different computing node. Therefore data transfers are required between the execution of each task in order to deliver the correct input data to the correct process in the correct computing node.

IDU forms part of an iterative chain of processes that will process the Gaia data. This chain of processes will be executed once every 6 months over a 5 year period. Each time that IDU is executed it will process all of the data amassed at that point, so as the mission continues, the volume of data will increase reaching roughly 100TB at the end of the mission. Although the actual volume of data may not be overwhelming, the challenging aspect of the processing is the relationships within the data, and the reorganising and movement of the data, which must be carried out between tasks.

B. Scalability tests of GPFS

An initial version of IDU involved all processes reading their input data from the GPFS storage device. In order to determine how well this approach would scale to a large number of worker processes running in a large number of computing-nodes, we carried out some scalability tests. The input files for these tests were of equal size and represented a certain amount of processing. In all cases, we just ran one worker process per node.

We found that if we increased the number of IDU worker processes, and therefore the number of processes accessing the GPFS, the system scaled fairly well up to about 8 nodes — the speed-up factor being 7.4, whereas the speed-up factor for 16 nodes was just 13, as illustrated in Fig. 1. Thus, although GPFS reveals a rather good scalability, the I/O performance per node starts decreasing significantly already at just 16 nodes.

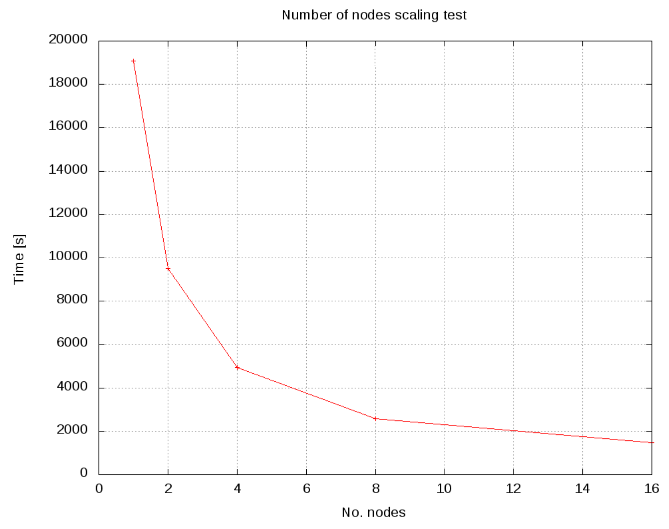

Fig. 1. Processing time of an IDU prototype over a fixed amount of data when using 1 to 16 nodes accessing directly the GPFS disk

IV. The solution — MPJ-Cache

We have designed a scheme which involves the grouping of the available computing nodes into Node Groups (NGs). Within each NG, one node is designated as the Node Group Manager (NGM). A Server process is executed in the NGM, which creates and maintains a cache of data that user application processes can query. We refer to user application processes simply as Client processes. The inter-node communication — between Clients and Servers, as well as amongst Clients — is implemented on top of an implementation of MPJ, making use of any high-speed network support provided by the MPJ implementation.

There are, of course, a number of possible configurations that a given set of nodes can be grouped into. For example, 64 nodes can be grouped into 4 groups of 16, or into 8 groups of 8 nodes. One of the objectives of the tests described in the next section was to determine the optimal NG configuration in order to maximise data access performance for a given set of files and a given application.

MPJ-Cache has two relatively distinct components within it, namely the the communication layer and the data cache. These components are illustrated in Fig 2.

A. MPJ-Cache - the communication layer

The objective of the MPJ-Cache communication layer is to provide user applications with a high-level API of data access methods useful in HPC environments, while making best use of the available communication resources such as any high-speed, low-latency networks which might be present. MPJ-Cache makes use of an implementation of MPJ to perform inter-process communication. Implementations of MPJ, such as MPJ Express and F-MPJ implement an API of methods, such as `MPI_Sendrecv()`. MPJ-Cache builds upon the MPJ API, and offers its own API

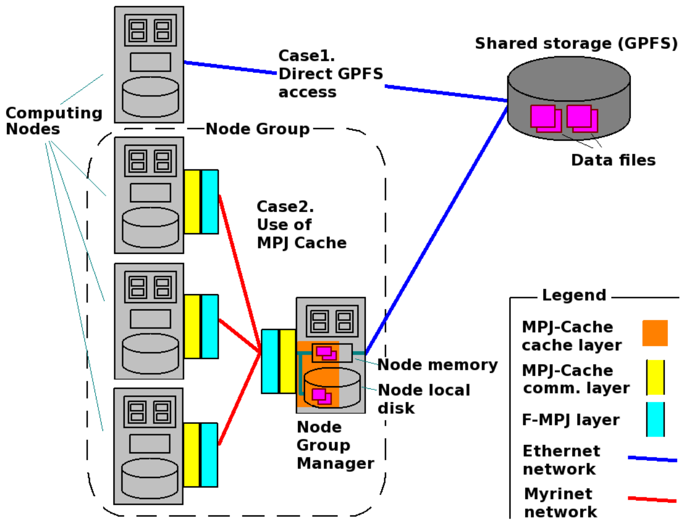

Fig. 2. MPJ-Cache components and test setup

of methods which are at a higher level of abstraction. For example, the MPJ-Cache API includes the static method `retrieveSingleFileAsByteArray()`, which allows Clients to request a file as an array of bytes. The implementation of this method involves 2 calls to the MPJ method `MPI_Sendrecv()`. Firstly, a request is made to find the size of the file. Then, once the size of the file is known, a buffer can be allocated that will contain the data and a second call to `MPI_Sendrecv()` is made requesting the actual file. However, from the perspective of the Client application, it must only make one call to `retrieveSingleFileAsByteArray()`.

The communication layer also includes file splitting and recombining functionality, which allows accessing files with sizes that exceed the maximum data size permitted by the underlying implementation of MPJ. Such large files are split into smaller chunks by the Server, which are then sent in separate messages, and finally, once all of the chunks have been received at the Client side, they are recombined and passed to the Client application.

B. MPJ-Cache - the cache

In the context of MPJ-Cache, the cache refers to the part of the Server application that maintains the actual cache of data. The Server can be configured to either store the most frequently used data in memory, on the local disk of the node that it is running on, or to simply act as a gateway, retrieving data from a remote location as requests are received. Indeed, the cache can be spread over these 3 locations. The Server maintains a list of all of the files that it is aware of: those it has in memory, those that it has on its local disk and those which might be in a remote location. The Server initializes its cache at start-up, based on its configuration, but it can update the cache during the execution of the application, depending on its configuration. A number of policies to control the maintenance of the cache including First In, First Out (FIFO) and Least Recently Used (LRU) are available.

C. Selection of MPJ Implementations

Among the available MPJ implementations, MPJ Express and F-MPJ are the libraries with a more active development, so they have been evaluated in order to select the best performer for use by MPJ-Cache. F-MPJ implements support for several interconnection networks, among them the support for InfiniBand in the low level communication device `ibvdev`, which runs directly on top of IBV (InfiniBand Verbs), and support on Myrinet in the device `omxdev`, which runs directly on top of MX. We first carried out initial tests to determine their performance in our particular production environment. MPJ Express, F-MPJ, MPICH-MX and MX utilities and were tested using a Pingpong and a Broadcast benchmark. The results of these tests are given in Tables I and II. These showed that F-MPJ performs very well in the target environment, in particular, it offers very low latency for short messages and good bandwidth with longer messages.

TABLE I
PINGPONG TESTS (POINT-TO-POINT COMMUNICATIONS)

Application	Latency (μms)	Bandwidth (MB/s)
F-MPJ	17.0	246.12
MX utilities	17.0	247.0
MPJ Express	23.2	180.8
MPICH-MX	17.0	247

TABLE II
BROADCAST TEST - 8 NODES, 1 PROCESS PER NODE

Data	MPICH-MX		F-MPJ		MPJ Express	
	Latency (μms)	BW (MB/s)	Latency (μms)	BW (MB/s)	Latency (μms)	BW (MB/s)
2MB	21.9	96.0	25.8	81.3	43.6	48.1
4MB	42.7	98.2	52.4	80.0	87.5	47.9

D. MPJ-Cache in practice

One consideration that users of MPJ-Cache must take into account is that applications wishing to make use of the system must be executed within the MPJ environment. This has an effect on how the user application can be launched. The approach that we took is to initially launch the same class — `LaunchAppsInProcesses` — in all of the MPJ processes. Within the MPJ environment each process is identified by a unique identifier called the process rank. We followed the approach that the process with rank 0 would act as a Server process, while instances of the user application would be launched in the other processes. In order to create several NGs, with each NG containing a Server process and a number of Client processes, we simply need to launch several jobs, each one starting an instance of the MPJ environment.

V. THE TESTS

In order to directly compare the performance of MPJ-Cache against direct access to GPFS, we executed a campaign of tests, designed to reflect the characteristics of real applications running in a HPC environment. These tests can be grouped into 2 main categories. In the first case, the Client processes retrieve data directly from the GPFS, while in the second case the Clients retrieve the same data using MPJ-Cache. These 2 cases are illustrated in Fig 2. Tests involving the use of MPJ-Cache can be further categorised into those involving 1 NG and those involving multiple NGs.

In order to be representative of real applications, the data communication within the tests was interleaved with "processing" by the Client applications — simulated by "sleeps" of the Client processes between each request for data. Tests involving sleeps of varying lengths were carried out. Each Client also performs an initial sleep during its initialisation to ensure that not all of the Clients begin requesting data at the same time.

In total, 96 tests were executed in this test campaign. In all cases, the tests involved Clients retrieving 20 different files. There are many test parameters which were varied including the size of the data files (1MB, 10MB, 100MB), the number of clients (16, 32, 64 or 128), the number of NGs (1, 4 or 8), and the sleep time. In the case of the MPJ-Cache tests, the Servers were configured to store all of the data in a cache in memory.

The BandWidth (BW) referred to in these results has been calculated using the time a request takes to be processed from the Clients perspective. Therefore, it includes any delay which might occur at the Server side.

A. System features

The MareNostrum Supercomputer [3] consists of 2560 JS21 blade computing nodes, each with 2 dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz, which totals 10240 cores (9.2 GFlops per core), and 8 GB memory per node. The MareNostrum was ranked 5th of the world according to the Top500 List at the time of installation (2006), although currently it is ranked 118th based on its Linpack performance (measured 64 TFlops out of 94 TFlops peak). MareNostrum nodes are interconnected through low latency Myrinet 2000 network (12 switches conform the fabric), as well as through Gigabit Ethernet, this latter used to access the 280 TB of disk storage though GPFS (General Parallel File System). The OS is SuSE Linux Enterprise Server 9, the JVM is IBM J9 1.6.0, the F-MPJ release is 0.1.0 and the MPI implementation is MPICH-MX 1.2.7.4, while the MX driver version is 1.2.7-64.

VI. Results

A. Direct GPFS access vs. MPJ-Cache with 1 NG

MPJ-Cache generally out performs GPFS in those tests with smaller files sizes and a short sleep between requests, as shown in Table III, and illustrated in Fig 3.

MPJ-Cache also outperformed GPFS in tests with large file sizes and a long sleep period, as shown in Table IV, the one exception being the case of 128 nodes, where GPFS performed best. The explanation for the poor performance of MPJ-Cache in that test is that the Server process was overloaded. In other words, the Server was receiving requests faster than it was able to handle them, therefore, a queue of requests built up.

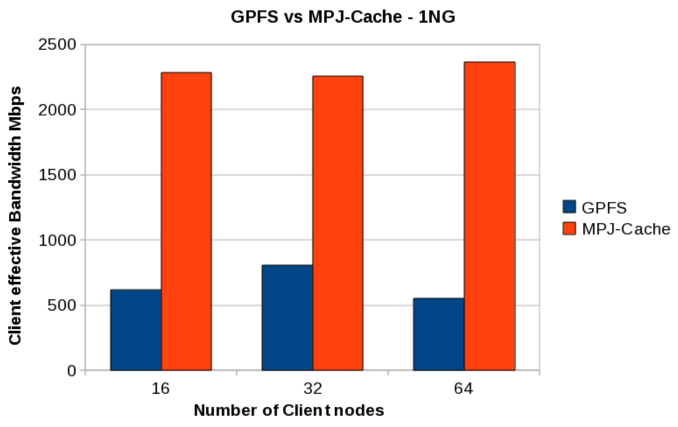

Fig. 3. Comparison of GPFS vs MPJ-Cache when using 1 NG on small files (1MB) and frequent requests (1ms)

TABLE III
MPJ-Cache Results — small files, small sleep, 1NG

Clients	Data	Sleep (ms)	GPFS BW (Mbps)	Cache BW (Mbps)	Speed-up
16	1MB	1	616	2284	3.7
32	1MB	1	806	2256	2.8
64	1MB	1	553	2365	4.3

In fact, in most of the cases where the GPFS outperformed MPJ-Cache, the difference was explained by an overloading of the Server. We confirmed this by examining the Client log files in those cases where the MPJ-Cache performed poorly. We noted that the initial requests received by the Server were processed quickly, and therefore the initial bandwidth experienced by the Clients was relatively good. However, as more Clients began to request data from the Server, the Server became overloaded and the average reply time experienced by the Clients increased — hence the decrease in the calculated bandwidth.

B. Direct GPFS access vs. MPJ-Cache with multiple NGs

With this set of tests we intend to find the optimal configuration for accessing data from a given set of nodes. Effectively we want to maximise the total amount of data that can be transferred around the system during a given period of time. We call this rate the "aggregate data rate". We must note that this data rate contains within it, the initial sleep as well as

TABLE IV
MPJ-Cache Results — large files, large sleep, 1NG

Clients	Data	Sleep (ms)	GPFS BW (Mbps)	Cache BW (Mbps)	Speed-up
16	100MB	20000	1026	1639	1.6
32	100MB	20000	1090	2164	2.0
64	100MB	20000	265	295	1.1
128	100MB	20000	796	83	0.1

the inter-request sleeps performed by the Clients in order to simulate real applications.

The highest aggregate data rate achieved by direct access of GPFS was 13.7Gbps, achieved through the use of 128 Client nodes, requesting 100MB files with a sleep of 200 milliseconds between requests. Interestingly, if the sleep was reduced to 10 milliseconds, the data rate fell to 9.1Gbps.

The highest aggregate data rate achieved through the use of MPJ-Cache was 103Gbps. This was also achieved using 128 Client nodes, but arranged in 8 NGs. Therefore, 136 nodes in total were used (including 8 MPJ-Cache Servers). The files used were 100MB each, while the sleep was 1 millisecond.

The results of this test campaign showed, that given a certain number of nodes, MPJ-Cache allows for a much higher aggregate data rate than direct GPFS access, as illustrated in Fig. 4

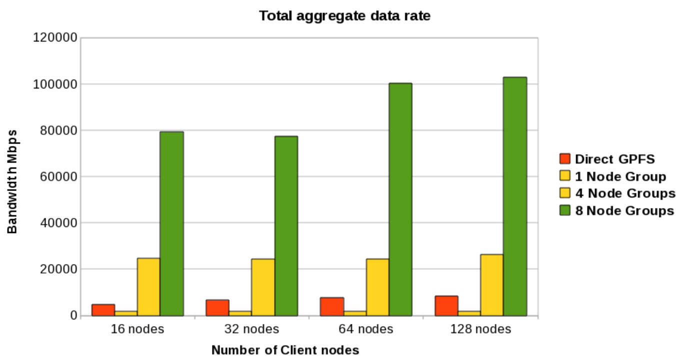

Fig. 4. Total aggregate data rate for GPFS and MPJ-Cache, using different Node Group configurations. 100MB files have been used in this case.

VII. Conclusion and Future Work

Our tests confirmed the impressive performance for high-speed, low-latency networks achievable with F-MPJ. MPJ-Cache, configured to create a single NG, offers better performance than GPFS for small numbers of Client nodes, and especially when working with small files and frequent requests. We observed in our test campaign that the available bandwidth on the Myrinet network was optimally being used, and the performance of MPJ-Cache only decreases when the Server process is overloaded with requests. Furthermore, when we move to the situation of multiple NGs, the total aggregate data rate obtainable through the use of MPJ-Cache is much higher than that obtainable for the same number of nodes directly accessing GPFS.

Although our data cache system is a relatively thin layer, sitting on top of an implementation of MPJ, we believe that there are a range of applications which could benefit from its use, not just the applications described in this paper. The creation of data caches amongst the available computing nodes can avoid the I/O bottleneck which can occur when many processes are accessing a central storage device, while the use of F-MPJ allows for the best performance to be extracted from the available network.

In the case of HPC environments which are shared by many users, one possible issue is how the applications of one user may affect the applications of other users. One of the benefits of using MPJ-Cache is that, due to the caching, there will be a reduced number of accesses of the shared disk, which should improve the disk I/O performance experienced by other users.

We intend to improve the functionality of MPJ-Cache by adding more data access methods to the API that it offers, including the addition of methods to support a “push” model from Server to Client, in addition to the existing “pull” methods which allow for the Client-Server model. Finally, we intend to test MPJ-Cache in other HPC environments, such as its use with the LUSTRE and Panasas file systems, as well as to try to identify other Java-based I/O-intensive applications, running in HPC environments, which could benefit from the use of MPJ-Cache.

Acknowledgment

This work was supported by the MICINN (Spanish Ministry of Science and Innovation) - FEDER through grant AYA2009-14648-C02-01 and CONSOLIDER CSD2007-00050. We also acknowledge the support that the F-MPJ project receives from MICINN, TIN2010-16735.

References

[1] European Space Agency. The gaia mission. http://gaia.esa.int/ [Last visited: July 2011].

[2] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. mpijava 1.2: Api specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: July 2011].

[3] Barcelona Supercomputing Center. Marenostrum supercomputer. http://www.bsc.es/plantillaA.php?cat_id=200 [Last visited: July 2011].

[4] M. Danelutto, P. Fragopoulou, V. Getov, E. Tejedor, R. M. Badia, T. Kielmann, and V. Getov. A component-based integrated toolkit. In *Making Grids Work*, pages 139–151. Springer US, 2008.

[5] M. Philippsen, R. F. Boisvert, V Getov, R Pozo, J. E. Moreira, D. Gannon, and G. Fox. Javagrande - high performance computing with java. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, 2001.

[6] A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core hpc systems using java. *Journal of Parallel and Distributed Computing*, 69(6):532–545, 2009.

[7] G. L. Taboada, J. Touriño, and R. Doallo. Java for high performance computing: Assessment of current research and practice. In *Proc. 7th International Conference on the Principles and Practice of Programming in Java (PPPJ'09)*, pages 30–39, Calgary, Alberta, Canada, 2009.

[8] G. L. Taboada, J. Touriño, and R. Doallo. F-mpj: Scalable java message-passing communications on parallel systems. *Journal of Supercomputing (In press, DOI: 10.1007/s11227-009-0270-0)*, 2011.