Cluster Comput (2014) 17:503–528
DOI 10.1007/s10586-013-0294-3

DDOps: dual-direction operations for load balancing on non-dedicated heterogeneous distributed systems

Nader Mohamed · Jameela Al-Jaroodi · Hong Jiang

Received: 15 January 2013 / Revised: 14 May 2013 / Accepted: 12 July 2013 / Published online: 30 August 2013

Abstract Given the increasing prevalence of compute/data intensive applications, the explosive growth in data, and the emergence of cloud computing, there is an urgent need for effective approaches to support such applications in non-dedicated heterogeneous distributed environments. This paper proposes an efficient technique for handling parallel tasks, while dynamically maintaining load balancing. Such tasks include concurrently downloading files from replicated sources; simultaneously using multiple network interfaces for message transfers; and executing parallel computations on independent distributed processors. This technique, DDOps, (*D*ual *D*irection *Op*eration*s*) enables efficient utilization of available resources in a parallel/distributed environment without imposing any significant control overhead. The idea is based on the notion of producer pairs that perform tasks in parallel from opposite directions and the consumers that distribute and control the work and receive and combine the results. Most dynamic load balancing approaches require prior knowledge and/or constant monitoring at run time. In DDOps, load balancing does not require prior knowledge or run-time monitoring. Rather, load balancing is automatically inherent as the tasks are handled from the opposite directions, allowing the processing to continue until the producers meet indicating the completion of all tasks at the same time. Thus DDOps is most suitable for heterogeneous environments where resources vary in specifications, locations, and operating conditions. In addition, since DDOps does not require producers to communicate at all, the network effect is minimized.

Keywords Load balancing · Heterogeneous resources · Parallel computing · Distributed systems · Dual-direction processing

N. Mohamed (✉)
College of Information Technology,
United Arab Emirates University, P.O. Box 17551, Al Ain, UAE
e-mail: nader.m@uaeu.ac.ae

J. Al-Jaroodi
Middleware Technologies Lab., Aali, Bahrain
e-mail: jaljaroodi@gmail.com

H. Jiang
Department of Computer Science and Engineering,
University of Nebraska-Lincoln, Lincoln, USA
e-mail: jiang@cse.unl.edu

1 Introduction

As computing resources become abundant and their capabilities grow, software demand grows even faster. For example, in an earlier era of processors with clock speeds around a few kilo Hertz and limited memory capacity, operating systems were small and had limited features and most applications were text-based and graphics-based applications were unheard of. As the hardware advanced to higher clock speeds, larger memory space, and better processing capabilities, software evolved even faster and started consuming whatever the hardware has to offer and demanding more. The same happened to the networking technologies. For example, when the networks were first designed and used, simple text messages and specific types of files were exchanged, which the limited bandwidth available then was able to handle. Now, the networks have advanced tremendously and can offer huge transfer capacity, yet we still consume all of it and crave for more because we demand the immediate transport of millions to billions of bytes of data (text, audio, images, video, etc.) at incredible speeds. The same applies to applications, where the computation demands increase so fast,

even the fastest computing systems cannot keep up. Therefore, we find ourselves forced to consider replication, redundancy and parallelization. However, this comes with costs and based on how, where, and when we utilize such methods, the overhead could be prohibitive.

Concurrent and parallel processing has been explored and exploited heavily over time using models ranging from the shared memory model, to message passing, to distributed objects across distributed environments. The same happened in networking due to the high demand for network capacity to handle the large amounts of data [5]. Many models and techniques were introduced to provide good results for specific requirements or environments. However, we currently face a much harder job in making parallelization work efficiently in highly distributed environments. This is further compounded by the advances of the Grid and Cloud architectures that impose much tougher challenges due to: (1) the high heterogeneity of the resources; (2) the extreme nature of the applications and their very high demand on resources; (3) The great distances covered by the networks spread worldwide; and (4) the large volumes and high complexity of data and huge files that need to be shipped around as needed [4, 6, 20]. As more techniques emerge, one of the very important issues that arise is scheduling and load balancing [20, 33, 45]. Several application domains afford parallelization, but when in use, the gain is often below the expectation. The heterogeneity of the resources and high latency in communication cause problems and offset the gains. Therefore, it is important to find schemes that will reduce the need for communication and support effective scheduling techniques that will provide some level of load balancing. Yet again, this is not easily doable since maintaining good load balancing requires some additional work (processing and communication), which in some cases can be prohibitively expensive. As we will discuss in the related work section, there are several models and techniques that offer efficient scheduling and load balancing. Some of these are static, which require prior knowledge of the environment and applications requirements. However, as the applications start executing, these models have no way to adapt to changes in the environment or requirements. The other type is dynamic load balancing, where the load balancer monitors the environment and application requirements during run-time and attempts to make adjustments to redistribute the tasks and adjust the load as necessary. This type of load balancing is generally capable of handling dynamic changes in the environment's operation attributes during run-time. However, the communication overhead imposed by the monitoring and load redistribution is usually high, which negatively affects the overall performance [48]. In addition, the response to changes may also take too long, which reduces or even eliminates the expected benefits.

In this paper we offer a new technique (DDOps) for distributing/processing parallel and distributed tasks in such a way that load balancing is automatically inherent. The main idea of DDOps is based on the notion of processing tasks in a job in opposite directions such that the processes performing the tasks will not have to contend on shared resources. In addition, the processes will work independently at their own speeds until they meet without needing to know how far the others are in their processing. As a result, DDOps avoids the common requirements of the state-of-the-art approaches, namely, the need to know any information about the resources being used before hand and the need to monitor these resources during run time, while it continues to maintain the load balance among all participating resources. The load balancing here implies that all resources available are utilized at their maximum possible performance and all will end their work at relatively the same time. Thus there will be minimum idle time for any of the resources. Furthermore, if some of the resources capabilities change during run-time, the work will still be done at relatively the same time except that some work load will shift among the resources. To better understand the key DDOps idea, we view it as being analogous to having a long fence that needs to be painted by two painters, where you could just tell them to each take one half of the fence, which can be problematic if one of the painters is slower than the other and the faster one must wait idly for the slower one to finish his half. Or, you could simply start the painters at the opposite ends of the fence and get them to move towards each other. Here the painters will work independently and will continue at their own pace until they meet somewhere along the fence and they will be done at the same time. The painters' meeting point will change depending on which painter is faster. Technically this will free the client from having to deal with the run-time changes in the environment and redistributing the load to adjust for these changes. Thus the processors will be completely independent and will not need to wait for any instructions or deal with each other during run-time.

In the remainder of the paper, we first discuss some of the current work relevant to our technique in Sect. 2 then we define the problem scope and identify its boundaries in Sect. 3. In Sect. 4, we introduce the general technique for the dual-direction operations (DDOps). In the subsequent sections we show examples of different application domains where the technique applies, including parallel file downloads in Sect. 5, parallel computations with the matrix multiplication example in Sect. 6, and message transfer over multiple network interface cards in Sect. 7. In Sects. 5, 6 and 7 we outline how DDOps is used, offer an overview of other available solutions, report on the experimental results and briefly discuss the outcomes. In Sect. 8, we discuss and analyze our technique and offer additional experimental results that apply to all domains. Finally in Sect. 9 we conclude the paper.

2 Background and related work

Computing facilities and resources are in high demand because many applications are increasingly resource intensive and require pooling multiple resources together to perform the required tasks efficiently. Parallel computing and redundant resources are very important to enable efficient execution of such applications. As a result, it has also become important to accommodate heterogeneous resources and allow applications to distribute their tasks among them. With this notion, we find ourselves facing the parallelization challenges in various domains. Although many problems and applications are theoretically highly parallelizable, the practical side imposes many constraints including communications delays, contention on resources and high coordination and synchronization costs.

In [5] authors introduce a middleware infrastructure to support parallel programming models on heterogeneous systems. This framework allows application developers to write and execute object-oriented parallel programs on heterogeneous computing resources. It offers a run-time environment using software agents to facilitate and coordinate remote executions of parallel tasks. However, this framework does not explicitly address load balancing among the resources. Authors in [30] survey various techniques to optimize parallel programming including program transformation, communication and message passing optimizations, self adaptation and load balancing. Load balancing in heterogeneous distributed environments imposes great challenges especially if the environment suffers from long transmission delays and nondeterministic load levels. To achieve load balancing in such an environment, it is important to target three goals: (1) minimizing idle time, (2) minimizing overloading, and (3) minimizing control overhead [48]. Achieving these goals requires efficient algorithms to help distribute the load evenly and devise ways to keep that balance during execution. Yet such algorithms are affected by many factors and variables that can cause new problems. For example, many load balancing algorithms require constant exchange of information, task redistribution and coordination efforts, which can be hindered by the long delays experienced in various distributed environments such as the Grid and the Cloud. The effects of these delays were explained in [15], where longer delays have proven to cause more disturbance and postpone overall system stability while trying to balance the load among the resources.

When parallel tasks are distributed, the type of environment they are in has a great effect on how and when these tasks will complete. In a homogeneous dedicated environment, tasks can be divided into equal sub tasks and distributed among the available resources. Thus we can minimize idle time, overloading and control efforts easily. However, in the same environment, if the resources are shared among multiple applications, some factors such as memory size, network bandwidth and CPU utilization may change during execution. In this case the initial equal distribution of sub tasks may no longer achieve good load balancing and some tasks will hold others until they are done which increases idle time for many resources and may also overload others. Therefore, it is necessary to device a dynamic approach that monitors the operating attributes of the resources and current loads to adjust the load when necessary. Yet this will increase control efforts and may not be possible in all cases since load adjustment may require data and/or task redistribution, which is usually very costly. On the other hand, if we have a dedicated yet heterogeneous environment, we need to have prior knowledge of the available resources and use that knowledge to fairly distribute the tasks such that faster processors, for example, will get more units to process while the slower ones will be given fewer units. The last possibility and the one that is currently very common on Grid, Cloud and Internet environments is having a shared heterogeneous environment to deal with. In this case, prior knowledge is helpful to achieve initial load balancing; however, we also need continuous monitoring and constant adjusting of the load to achieve some level of load balancing throughout execution time. Here we review some of the approaches designed for load balancing where each one has its own merits and at the same time has some drawbacks. Generally, load balancing methods cover several directions [38] and maybe viewed to have three categories involving resource aware models, prediction models and divisible load theory. From another perspective, we may categorize load balancing methods as static or dynamic.

Static methods usually acquire knowledge about the environment before execution and distribute the tasks according to this information. This would work very well for dedicated heterogeneous environments since there will be minor changes during run time. One example of this approach is described in [40] where a system of non-linear equations is solved to decide on the size of each subtask based on the systems specifications. Another example is described in [43] based on a maximum flow algorithm to minimize the subtasks assigned to the most loaded or the slowest machines. Another example of static scheduling is a local search algorithm [46] designed to solve large instances of the independent tasks mapping problem with the assistance of a GPU. On the other hand, dynamic load balancing methods rely heavily on monitoring the environment during run time and adjusting subtask allocation according to the most recent changes observed. In [27] a master/slave organization is used to distribute the tasks based on most recently observed performance of the participating machines. The master basically distributes a portion of the subtasks to the slaves and adjusts the size of the next subtask to balance the load depending on how fast the response arrives from these slaves.

In [49], a load balancing method based on the assumption of preserving the speed of computation in the consecutive iterations is introduced. The algorithm determines the load size of the next round based on the computation and communication times for the previous round. Another example is in [18] and [20] where a load balancing algorithm is designed for networks of workstations (NOW) where the nodes are shared among multiple users and multiple tasks and the available processing power varies during run time. Here again in the initial phase where some subtasks are distributed, the response time is considered to determine the size of the next round of subtasks. In both examples, constant monitoring of the tasks is required and, when the network delay is factored into account, the estimates become less realistic. For example, if there are two nodes on a NOW where one is faster than the other, but the communication delay associated with the first node is much higher, the response will arrive later for the first node than for the second. Consequently, the faster node will be assigned less work. However, in reality, it should have been given more work to process to amortize its longer communication delays.

In dynamic load balancing, load distribution may involve several methods. The easiest method is to use the master/slave model and divide the task into multiple partitions that are then distributed to the slaves through several steps. This would work well for large tasks and when the communication overhead is low. Another possibility is to create one partition per contributing node and, if one completes its work earlier than others during run-time, it is assigned some of the work relocated from other nodes. This reduces the communication overhead, but introduces the extra complexity of load reallocation, which may involve data sharing, process duplication or process migration. Another possibility is to replicate all tasks on all nodes and then determine portions of tasks each node will handle and adjust these parameters as necessary during run time. While this sounds promising, it involves high replication overhead and requires subtasks to be highly independent. Another approach is to use a distributed model where the nodes communicate with each other to exchange load and environment information and exchange tasks when necessary to balance the load among them. Yet again, this can cause various problems with non-dedicated resources. In particular, when resources are distributed over long distances, the delay factor in communication becomes predominant and reduces the gains of load balancing [15].

Another categorization for load balancing methods divides them into centralized and distributed methods. Centralized load balancing algorithms are usually simpler to design and control and we have covered several examples earlier such as [18, 20, 27] and [43]. These would work very well on stable environments and would perform best if the network delays are minimal. However, when dynamic environments are involved and high delay factors are in effect, it may be necessary to consider a distributed load balancing model. In one example a semi-distributed load balancing algorithm is described [56], where the computing environment is clustered in a mesh structure such that each node could inform the others if it was idle or if one of its neighbors is idle. Therefore, the load will be distributed among idle nodes and updated as the environment changes. In addition, a decentralized load balancing algorithm is introduced for Grid environments where the communication overhead is reduced while maintaining adequately updated state information during run-time [3]. Another example [30, 31] uses a global scheduler in an adaptive distributed load balancing model. This model relies on prior knowledge of each contributing node's capabilities, current tasks and power in addition to having best task-node mappings and expected task completion times. The model distributes large tasks first then runs smaller tasks in parallel during nodes' idle times. As a result, it reduces communication and conflict between load balancing goals, yet it may be unfair to the small tasks that need to compete for idle times. Also in [19] we find a distributed sender-initiated model that adapts to the systems operational conditions and each node can decide on the load autonomously. In this model, load balancing is handled locally by the nodes and involves coordination among the nodes, which increases the overhead.

Two additional example for load balancing on the Grid are the GridSim load balancing [55] and the Enhanced GridSim load balancing [48]. The GridSim model relies on the hierarchical structure of the Grid environment and places the load balancing control at the Grid Node level. Therefore, each Grid node handles the tasks and resources available within it to offer best achievable load balancing and reduces idle time and control overhead. In Enhanced GridSim, the model is extended to include a second level of load balancing performed at the machine (that being a cluster computer, a NOW, or any computational/resource entity that also contain multiple local resource) level where local load balancing is done within each machine. Furthermore, load balancing methods may be designed as part of the application or moved to the system level. Load balancing at the application level has the advantage of being more informed of the application's requirements, while implementing it at the system level will provide a more efficient approach as it will be better tuned to the available resources and independent of the specifics of the applications. Another approach that emerged for load balancing is to introduce it at the middleware level, thus having more links with the system while maintaining independence from the applications. A final example we include here involves process migration to achieve load balancing [53]. PS-GRADE, a graphical environment for parallel programming, is enhanced by adding a model to monitor global states to decide on the processes allocation/reallocation based on load levels. The synchronizers

gather processes states, construct the global state then issue control signals to preserve states and migrate processes to less loaded resources. The model can support a centralized synchronizer or a group of distributed synchronizers.

More recently more criteria in scheduling and load balancing became important one of which is energy efficiency. Several research groups are investigating different approaches that could offer efficient scheduling and load balancing, while maintaining an efficient use of energy. The survey [54] focuses on the characteristic of two main power management technologies: (a) static power management (SPM) systems that utilize low-power components to save the energy, and (b) dynamic power management (DPM) systems that utilize software and power-scalable components to optimize the energy consumption. In addition, [35] Outlines the role of the communication fabric in data center energy consumption and presents a scheduling approach DENS that balances energy consumption, individual job performance and traffic demands. The authors in [45] introduce a simple two phase heuristic for scheduling independent tasks that improves the previously known Min-Min heuristic. It relies on the results of the cellular genetic algorithm and provides results in a significantly reduced runtime. Furthermore, in [47] the authors show that the two phase heuristic introduced in [45] provides for a more energy-efficient approach as tasks are better distributed across resources. In [36] independent batch scheduling in computational grids uses makespan and energy consumption as the scheduling criteria. This works best with massive parallel processing of applications that require a large amount of data and results in possible reduction of cumulative power energy utilized by the system resources. The method requires gathering prior information and deciding on the schedule accordingly. NBS-EATA [34] offers an approach to address the problem of allocation of tasks onto a computational grid, while simultaneously minimize the energy consumption and the makespan subject to deadline constraints and tasks' architectural requirements. It offers energy optimizing power-aware resource allocation strategy in computational grids for multiple tasks.

In general load balancing techniques require knowledge of the available resources and their operating conditions. Table 1 offers a summary of the techniques we discussed. To satisfy needs of non-dedicated heterogeneous environments, it is important to have prior knowledge of the resources, in addition to keeping an up-to-date view of the current operational conditions of the environment during run time. As shown in [16] and [44], this is usually a very costly requirement and may significantly reduce load balancing, which will in turn result in reduced gains of parallelization. As we mentioned earlier, load balancing is achieved when we can minimize idle time (i.e. when all contributing resources complete their tasks at the same time), minimizing overload on resources and minimizing control efforts. In DDOps as we will describe later, we offer load balancing that satisfies these requirements by reducing the overhead and eliminating the need for coordination among servers.

3 Problem definition

Parallelization is a desirable approach in many domains and finding an efficient way to do it is very important. In this paper, we address parallelization from a different angle that does not only involve parallel computations, but also parallel file/data transfer and parallel networking. In that broad context, we address the particular point of task distribution and load balancing. Therefore, we need to clearly define our scope and identify the problem boundaries. To do that, we introduce the DDOps technique. DDOps applies to a specific type of processes; however it is a starting point for a more general approach to be applied to most parallel or replicated operations and applications. Therefore, it is important to clearly state the governing problem parameters and declare the operational limits. The main objective of DDOps is to introduce efficient dynamic load balancing with minimal idle time and minimal communication and coordination overhead while executing our operations in parallel over a distributed heterogeneous environment.

The first requirement to successfully use DDOps is to have multiple distributed independent tasks or a large, but easily parallelizable task. As we discussed in the introduction we address problems in three different domains that fit this requirement namely parallel network transfers over multiple NICs (Network Interface Cards), parallel file download from replicated servers, and parallel computation for data or process intensive applications such as matrix manipulations, data analysis and image processing. Such domains serve a large scope of computational problems and can benefit well from DDOps.

The second requirement is the type and boundaries of the problems. In this case the problem has to be well defined in a geometric (preferably linear) form and must have well defined boundaries (known start and end points). When partitioned, the sub-problems need to be highly independent from each other (not necessarily fully decoupled, but enough to allow the producers to work with minimum interference from each other). Generally there is a huge collection of such problems including mathematical computations such as solving systems of linear equations. Moreover, several applications involve high volumes of data to be transferred, processed and analyzed as in data mining and pattern analysis. Furthermore, considering scientific operations, several computational sites rely on receiving huge data sets from labs and observatories to use in their computations.

In addition, to make this work well we need to have replicated and redundant resources, which is already available

Table 1 Summary of load balancing techniques discussed in Sect. 2

Ref. #	Name	Type	Pros	Cons	Suitable Env.	Comments
[40]	LB for sorting	S/C	Efficient during run time as there is no additional overhead	Any changes in the environment will not be accounted for since the load is all distributed beforehand. Described for one application (sorting)	Homogeneous or heterogeneous dedicated environment such as clusters	Solves linear system of equations to determine throughput of participating nodes then partitioning the problem proportional to the determined values
[43]	LB with flexibly assigned tasks	S/C	Works well if more units than needed are available and does not add any overhead during run time	Cannot handle changes in the environment during run time. Also cannot work well with limited resources	Homogeneous and in some cases heterogeneous dedicated environments	Works well for problems with overlapping processing or data. Requires the existence of multiple processing units that any of which could be assigned a task
[35]	DENS	S/C	Energy efficient scheduling	Gathers prior information to achieve its goal	Heterogeneous environment with power concerns	Balances energy consumption, individual job performance and traffic demands
[45, 47]	Two-phase heuristic	S/C	Schedules independent tasks on multiple resources	Requires preprocessing to determine optimized task allocations	Heterogeneous environments	Improves the min-min heuristic and provides energy efficiency
[36]	Batch scheduling	S/C	Provides energy efficient scheduling for the computational Grid	Requires prior knowledge of resources capabilities and works best for massively parallel tasks	Relatively homogeneous environments	Aims to minimize makespan and energy consumption
[34]	NBS-EATA	S/C	Determines efficient allocation of tasks on computational Grids	Requires prior knowledge of resources capabilities	Relatively homogeneous environments	Aims to minimize makespan and energy consumption
[27]	M/S LB	D/C	Eliminates the deadlock issue with the last task	Communication delays are not factored, which could result in unrealistic estimates	Shared homogeneous and heterogeneous environments with stable communications	Works with master/slave model. measures performance of processors to assign proportional new load
[49]	LB	D/C	Offers good load balancing through reallocation	Requires monitoring processors to determine load reallocation which creates a high overhead. Developed for parallel solution of linear equations	Heterogeneous shared environments (E.G. NOW) with stable communication performance	Works for the master/slave model and data-intensive applications. Uses sub-structuring method of structured and free 2D quadrilateral finite element meshes
[20]	Dynamic LB	D/C	Starts with better load distribution based on pre-analysis of workstation speed and current workload	Requires a wait time to determine initial information and calculate load distribution. Creates run time overhead while waiting to workstations to finish assigned load before giving more. Cannot tolerate communications delays	Heterogeneous shared environments (e.g. NOW) with stable communication performance	Requires very fine grain partitioning and utilizes system information to determine a dynamic LB at the beginning, then relies on received responses to determine next load assignment

Table 1 (*Continued*)

Ref. #	Name	Type	Pros	Cons	Suitable Env.	Comments
[55]	New LB	D/D	Localized event management offers faster response time for load reallocation	LB effects may take too long to propagate across a large system where some areas may be overloaded while a further area is sitting idle	Heterogeneous systems with relatively uniform resources and loads	Uses neighbor monitoring to request additional loads from overloaded neighbors
[3]	Decentralized LB	D/D	Reduced communication overhead using scalable status information exchange	Overhead still exists. Information is propagated from local groups to others, thus full system update may not happen quickly	Heterogeneous grid with relatively high communication delays	Uses task classification and assigns classes to sub groups of resources thus, updates needed remain localized for most of the runtime periods
[31]	Global scheduler LB	D/D	Could work well for a large number of distributed tasks with well defined needs	Does not help when we have data intensive or identical parallel tasks. Also requires users to define their balancing instances	Heterogeneous systems and distributed tasks (not parallel applications)	Requires prior knowledge of tasks properties. Distributes large tasks first then injects smaller tasks to fill idle times
[19]	Dynamic LB with delays	D/D	Effective resources utilization bu allocating incoming loads based on calculated completion times	Works well with distributed tasks but not parallel tasks where tasks may need to be re-allocated during run time	Heterogeneous systems with some communication delays	Uses regeneration-theory approach to average overall completion time while considering heterogeneity and communication delays. Any node may initiate DLB when new tasks arrive
[48]	LB for Grid	D/D	Works well when tasks are evenly distributed among Grid nodes and resources	Imposing high loads on a small portion of the Grid hierarchy disturbs LB since tasks will need to be redistributed beyond the local subsystem	Heterogeneous hierarchical distributed Grid systems	Task-level LB with localized decision making with the hierarchical Grid structure
[53]	Process migration	D/–	Offers accurate process states and migration mechanism	Requires constant monitoring and coordination overhead is high with multiple synchronizers	Heterogeneous distributed environments	Designed for message passing model and specific environment (PS-GRADE)

LB = Load Balancing, X//Y → X = S = static, X = D = dynamic; Y = C = centralized, Y = D = distributed

in various environments usually for reliability and enhanced performance purposes. For example, many compute nodes on the Grid have huge numbers of replicated computational nodes and storage units. Furthermore, the Cloud is built based on replication and mirrored sites that are currently used to offer better performance for the Cloud clients and improve the overall availability of their services. As for the examples we use here, in the case of FTP, the requested files need to have replicated copies on multiple FTP servers that do not need to be in the same location or in direct contact with each other. In addition, in terms of parallel computations, we need to replicate the processing code and the data used. Furthermore, in the network model we need to have a sender/receiver with redundant NICs on both sides, which is currently the case in most computers as they usually have more than one NIC.

DDOps uses the producer/consumer model where the consumer is the requesting process and it is also the process in control. The producer on the other hand is one or more processes/resources that will be executing the requested tasks independently from each other and delivering the results to the consumer. In a way this is just a spinoff from the client/server and the master/slave models and we use it to avoid confusions since in some cases the server may be either a producer or a consumer. Thus, we use the producer/consumer terms to have a clearer identification of

who is doing the processing and generating the results and who is getting the results of this processing. The technique allows all producers to work independently from each other and none of them require any prior or run-time knowledge of the other producers. The consumer on the other hand is the one taking control of the process, instructing the producers to start their tasks and providing them with enough information on how and where to start their work. The consumer is also responsible for collecting and aggregating the incoming results to produce the final results required and also to inform the producers when it is time to stop their work. Furthermore, the consumer is also responsible of reassigning tasks or partitions to free producers as soon as they finish the tasks at hand. This technique works on any type of distributed environment. However, it is best suited for heterogeneous highly dynamic environments. Thus the overall benefits are higher on a widely distributed system such as the Grid and Cloud than they are on LAN or homogeneous systems.

4 Dual-direction operations (DDOps)

The novel approach in DDOps offers parallelization of large tasks with dynamic load distribution and execution without having to actually know of the conditions and capabilities of the resources involved. As a result it works very well in heterogeneous environments with very dynamic operating conditions. Here we describe the DDOps technique that enhances the performance and load balancing while reducing the overhead. We will start by explaining the base case (dual-producer), where only two producers are used then we will describe the general case for multiple producers (k-producers). The idea is based on a basic concept of no synchronization parallelization. Thus there will be no need for constant monitoring or reallocation of the load during processing. In addition, the producers will not need to perform any synchronization or coordination measures among each other nor with the consumer. Those will be handled on the consumer side alone. When a problem is partitioned, instead of alternating partition assignments from the beginning, we proceed from either end of the set of partitions until the producers meet somewhere in the middle.

To clarify the picture let us take a physical problem where a supervisor has ordered forms to be processed and two assistants to do the job. A simple way to do this is for the supervisor to divide the batch into two equal parts and give them to the assistants. However, if one assistant is faster, then the supervisor will have to wait for the slower one to finish, while the faster assistant is free. To reduce this waiting time the supervisor could further partition the batch into multiple smaller batches (partitions) and give each worker a partition to work on and then they have to come back when done to take another. Therefore, the imbalance will be reduced, but there will be delays due to the continuous communications and monitoring to organize the dispatch and return of forms in an orderly fashion. In our model all the supervisor needs to do is spread the batch of forms in a common area (something like a long line on a table) and tell the first assistant to pick up forms from one end (moving from left to right) and the other to do the same from the other end (moving right to left). Each completed form is marked and returned to its location. This way, none of the assistants will need to go back to the supervisor and they will keep working at their own pace until they both encounter a marked form (indicating the other has completed that one). As a result, whether or not the assistants work at the same speed or not will not matter since the faster one will just process more forms and they will both finish at almost the same time. In addition neither of the assistants needs to know what the other is doing or how far he/she is. As a result, none of the assistants will be idle and they will not waste time trying to coordinate with the supervisor or each other. The movement in opposite directions allows DDOps to reduce coordination and control overhead of distributing the processing while reducing the restrictions imposed by the resources and allowing the consumer to fully utilize them.

4.1 The technique for the dual-producer case

In the dual-producer DDOps we have two replicated instances of the problem resources and each DDOps producer (DDP) handles the assigned blocks from an opposite direction. The DDOps consumer (DDC) initiates the operations of the first producer (DDP_1) to start from the beginning moving towards the end (left to right) and the second (DDP_2) to start form the end moving towards the beginning (right to left), as illustrated in Fig. 1. The parallel DDPs stop working as soon as they receive a stop message from the DDC. The stop messages are sent when the DDC receives the results for two consecutive blocks, one from each producer; as it signifies that all blocks were completed. For example, with an n-block problem, DDP_1 processes the ordered blocks $b_1, b_2, b_3, \ldots, b_m$ and DDP_2 processes the ordered blocks $b_n, b_{n-1}, \ldots, b_{m+1}$. This ensures that the faster DDP will have the opportunity to process more blocks, while the slower DDP will process fewer blocks during the same time period. In addition, none of the DDPs will need to wait for the other and they do not require any additional instructions during processing. Thus,

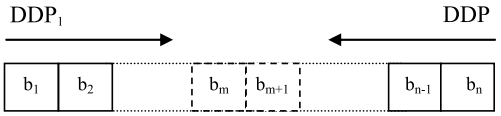

Fig. 1 Block processing and downloading directions by dual DDOps producers DDP_1 and DDP_2

they will be busy for relatively the same amount of time and will automatically achieve load balancing despite any changes in the environment's operational conditions during execution. The meeting point (m) will change depending on the capabilities of both DDPs, the overall operating conditions at both sites, and the available resources to each DDP. If both DDPs are similar and operate under very similar load and network conditions, m will fall roughly in the center. However, as soon as any of the parameters changes for one of the producers, m will move closer to the slower DDP as the faster DDP picks up more of the work. As both DDPs continue to process blocks and send their results from different directions, the DDC will put together the intermediate results and will know exactly when all the work is done and ask the DDPs to stop. This automatically solves the problem of load balancing between dual heterogeneous DDPs with dynamic loads located on heterogeneous environments with dynamic conditions. Yet it will not require any communication or monitoring from the producers side at all. Therefore, both producers will be independently working on their parts of the job without delays and without exerting any additional effort.

Each DDP maintains a separate block counter such that DDP_1 maintains an incrementing counter starting from one, while DDP_2 maintains a decrementing counter starting from the last block number. This information is easily obtained since the problem size and the number of blocks are determined at the beginning of processing. The DDC will provide this information along with the necessary data and processing instructions to the DDPs from the beginning. Since the results are sent over TCP [51], a reliable transport protocol, the blocks' first-in-first-out (FIFO) order is preserved for each TCP connection. That is, if DDP_1 sends results for blocks b_1, b_2, b_3, and b_4 to the DDC, these will be received in the same order b_1, b_2, b_3, and b_4. Although some blocks might be lost or corrupted, TCP maintains the reliability and FIFO order of delivery. These same features relieve the producers from needing to add sequence headers to each sent result, thus eliminating another possible overhead component.

The implementation of the technique on the producer side requires very few components. Each DDP maintains a counter and executes two main operations initiated by the control messages *Start* and *End* that are sent by the DDC. The *Start* message carries five parameters: the process name, *processName*, which tells the DDP which process is to be executed; the resource name, *resourceName*, which signifies either the file to be downloaded if the process is download, the result file name if parallel computation is the process and the connection name for the network transfer case; the block size *blockSize*, which tells the DDP how big each block is in bytes; the first block number *firstBlock*, which indicates the starting block; and the counter mode *counterMode*, to tell the DDP in which direction the processing will proceed ("*increment*" forcing the DDP to move forward or "*decrement*" which will make the DDP move backwards). As a result each DDP will perform the operations as requested by the DDC and continue to work until the *End* message arrives. The *End* message carries the name of the active process and tells the DDP to stop processing more blocks. Each DDP is multi-threaded to be able to handle multiple requests from different clients at the same time.

On the DDC side, more control is needed thus the implementation is a bit more complex. When the task is requested, the DDC needs to locate the available replicas of the task resources and get the information about the location and size. Using this information the DDC decides on the block size and start and end points. After issuing the *Start* messages to the DDPs, the DDC needs to manage the different results received from the two DDPs and keep track of processed blocks until all required blocks are completed before asking the DDPs to stop using the *End* messages. When the DDC needs to execute a task, it first finds the information about the available resources and the addresses for the participating DDPs. Then if necessary, DDC distributes copies of the problem (code and/or data) to all DDPs. Then it decides on a partitioning model and calculates the number of blocks and initiates the processing from the two DDPs by sending a different *Start* message to each one. For example DDP_1 is given the *Start* massage (*download, fileXYZ*, 4000, 1, *increment*) and DDP_2 is given (*download, fileXYZ*, 4000, n, *decrement*). Where *download* indicates that DDC needs to download the file named *fileXYZ* in blocks of size 4000 bytes. As the messages indicate, DDP_1 is to start from block number 1 and move forward, while DDP_2 is to start from block number n and move backwards. (i.e. If the file size is 800,000 bytes, then there will be 200 blocks of size 4000 bytes and $n = 200$). Each DDP upon receiving the *Start* message will begin file transfer accordingly. As the DDC receives file blocks, it will reconstruct the file and watch for the point where blocks meet. As soon as the DDC receives two blocks with consecutive block numbers, it knows that all blocks have arrived and immediately sends an *End* message to both DDPs. Any additional arriving blocks after that will just overwrite their earlier version so the DDC does not need to verify duplicates.

4.2 The technique for the k-producer case

The technique described in Sect. 4.1 provides an efficient solution for load balancing for parallel operations using DDOps with two producers. As we include more producers, it becomes more complicated. However, it still requires no coordination among producers. The general approach we will describe can apply to any number of producers; however, for simplicity and without loss of generality, we

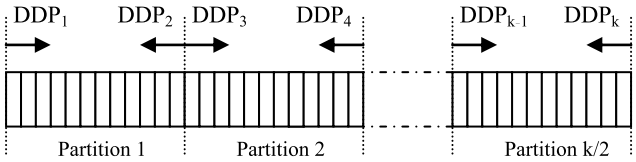

Fig. 2 Multiple DDP solution

will deal with an even number of producers, where the dual-producer method is repeated among pairs of producers. Generalizing the method for an odd number of producers requires minor changes in the producer operations. Each DDP will start processing blocks from the specified problem based on the content of the *Start* message it receives and will stop processing as soon as an *End* message arrives. However, the DDC will have to account for the difference in numbers. If we have an odd number of producers, we will pair $(k-1)$ producers and the last one is spawned into a pair of threads making up a virtual pair that will operate like the other pairs.

To demonstrate the concept, assume there are k replicated producers, where k is even. The DDPs are divided into $k/2$ pairs. In addition, the problem at hand is also divided into a large number of equal-sized blocks that are grouped into $k/2$ partitions as shown in Fig. 2. The size of each partition can either be the same for all groups or if prior information is available, it can be chosen to be proportional to the relative speed/performance of the associated DDPs such that high performing producers are given larger partitions. Each pair of DDPs handles its partition as described in the dual-producer case. During processing, any pair that finishes its partition is reassigned by the DDC to help another pair in its partition. Let us call the pair that finishes its partition a *freePair* while a pair that is still working a *busyPair*. Each *busyPair* consists of an *incrementing busy DDP*, which is working from left to right and a *decrementing busy DDP*, which is working from right to left. The partition of the selected *busyPair* is divided further into two partitions, *leftPart* and *rightPart*. After dividing the selected partition into two new partitions, one of the *free DDPs* is associated with the *incrementing busy DDP* to form a new DDP pair for the *leftPart* while the second *free DDP* is associated with the *decrementing busy DDP* to form another DDP pair for the *rightPart*. The first *free DDP* starts to work in the *decrement* mode from *the end of* the *leftPart* while the *incrementing busy DDP* continues its operation from left to right (without any changes) on the same partition. In addition, the second *free DDP* starts to work in the *increment* mode from the beginning of the *rightPart* while the *decrementing busy DDP* continues its operations from right to left on that partition, again without changes to its original work direction. This allows the *free* DDPs to help busy DDPs while the *busy* DDPs continue their operations normally without any interruptions or changes. All of this is achieved by simply sending two new *Start* messages to the *freePair* DDPs indicating the new block numbers to start from and the directions of the operation. All other pairs including the one receiving the help are not affected. Pair reassignments may be repeated several times by the DDC until there are no more DDPs needing help.

For large problems and with large heterogeneous distributed environments, it may be necessary to repeat the reassignment of pairs several times, which may cause some delays. To enhance load balancing among the DDP pairs and reduce reallocations, three greedy strategies are introduced. These strategies aim to reduce the number of reassignments, thus reducing the number of *Start*/*End* messages to be sent and resulting in more efficient operations among all pairs. In the first strategy, if there are multiple *busyPairs* that need help, we select the *busyPair* with the maximum number of unprocessed blocks in its partition to be helped first. Thus helping what appears to be the slowest of the producer pairs. The second strategy applies after selecting the pair to help. Within the selected DDP pair we associate the slower DDP (which processed fewer blocks than its partner) from the selected *busyPair* with the faster DDP (which processed the more blocks than its partner) from the *freePair* and vice versa. This makes the faster *free* DDP pick up the slack from the slower *busy* DDP, while the slower *free* DDP is paired with the faster *busy* DDP. As a result we rebalance the overall workload between the two new pairs. This strategy takes into consideration the recent history of the DDPs load by maintaining for each DDP S_i the number of blocks $C(S_i)$ it processed at the DDC. This number represents the contribution of the DDP to the processing of the blocks.

The third strategy divides the unfinished partition based on the recent progress of the four DDPs involved in the process rather than blindly dividing the remaining blocks in half. Let S_{il}, S_{dl}, S_{ir}, and S_{dr} denote the incrementing DDP on the *leftPart*, the decrementing DDP on the *leftPart*, the incrementing DDP on the *rightPart*, and the decrementing DDP on the *rightPart*, respectively. The DDPs S_{il} and S_{dr} are from the original *busyPair* that needed help while DDPs S_{dl} and S_{ir} are from the original *freePair* that will help the *busyPair*. The DDPs S_{il} and S_{dr} do not need to change their next block numbers. They continue their operations and directions. However, we need to assign new next block numbers for DDPs S_{dl} and S_{ir}. To assign the next block numbers for DDPs S_{dl} and S_{ir} based on the recent progress of the four DDPs involved in the process, we need to divide the unfinished blocks in the selected unfinished partition into two parts with sizes proportional to the recent speeds of their associated DDP pairs. Let us define a new function $P(S)$ to be the next block number that the DDP S needs to process. The number of unprocessed blocks in any partition is the next block number that the decrementing DDP needs to process minus the next block number that the incrementing

DDP needs to process plus 1.

$$\textit{Unsent fragments} = P(S_{dr}) - P(S_{il}) + 1 \tag{1}$$

We need to divide this number of blocks into *leftPart* and *rightPart*, with sizes proportional to their DDPs' processing speeds. Let the recent contribution of the *leftPart* DDPs be represented by $C(S_{il}) + C(S_{dl})$ while the contribution of the *rightPart* DDPs be calculated as $C(S_{ir}) + C(S_{dr})$. Therefore we can calculate the number of unfinished blocks for the *leftPart*, $U(\textit{leftPart})$, as:

$$U(\textit{leftPart}) = \textit{unsent blocks} \times \frac{\textit{Contibution of leftPart DDPs}}{\textit{Contribution of all four DDPs}} \tag{2}$$

$$U(\textit{leftPart}) = \big(P(S_{dr}) - P(S_{il}) + 1\big) \times \frac{C(t_{il}) + C(t_{dl})}{C(t_{il}) + C(t_{dl}) + C(t_{ir}) + C(t_{dr})} \tag{3}$$

The value of $U(\textit{leftPart})$ can be a real number but it should be truncated to an integer value. We have:

$$\textit{unsent blocks} = U(\textit{leftPart}) + U(\textit{rightPart}) \tag{4}$$

From Eq. (4), we can calculate the unfinished blocks for the right part, $U(\textit{rightPart})$, by:

$$U(\textit{rightPart}) = \textit{unsent blocks} - U(\textit{leftPart}) \tag{5}$$

$$U(\textit{rightPart}) = \big(P(S_{dr}) - P(S_{il}) + 1\big) - U(\textit{leftPart}) \tag{6}$$

From these equations, the next block numbers for DDPs S_{dl} and S_{ir} can be calculated as:

$$P(S_{dl}) = P(S_{il}) + U(\textit{leftPart}) - 1 \tag{7}$$

$$P(S_{ir}) = P(S_{dr}) - U(\textit{rightPart}) + 1 = P(S_{dl}) + 1 \tag{8}$$

For example, assume we are downloading a file named *fileXYZ* with 40 blocks of size 4000 bytes each from four servers using four DDPs (DDP_1, DDP_2, DDP_3 and DDP_4). The file blocks will be divided into two equal partitions and four replicated DDPs will transfer the blocks from the file, see Fig. 3. DDP_1 and DDP_2 will work on the first partition, while DDP_3 and DDP_4 will work on the second. This is achieved by the DDC sending the *Start* messages (*download*, *fileXYZ*, 4000, 1, *increment*), (*download*, *fileXYZ*, 4000, 20, *decrement*), (*download*, *fileXYZ*, 4000, 21, *increment*) and (*download*, *fileXYZ*, 4000, 40, *decrement*) to DDP_1, DDP_2, DDP_3 and DDP_4, respectively. Assuming all DDPs have equal operational properties all will finish around the same time. Now consider a scenario where workloads on the DDPs are different such that the second partition was completed while the first partition is still being processed. Now, consider at the time immediately after the second partition is completed the contributions of DDP_1, DDP_2, DDP_3 and DDP_4 are 4, 2, 12, and 8, respectively, as in Fig. 4(a). Based on the second greedy strategy described above we will associate DDP_4 with DDP_1 and DDP_3 with DDP_2, as in Fig. 4(b). This means that $S_{il} = S_1$, $S_{dl} = S_4$, $S_{ir} = S_3$, and $S_{dr} = S_2$. We have $C(S_{il}) = 4$, $C(S_{dl}) = 8$, $C(S_{ir}) = 12$, $C(S_{dr}) = 2$, $P(S_{il}) = 5$, and $P(S_{dr}) = 18$. We can calculate $U(\textit{leftPart})$ from Eq. (4) *as:*

$$U(\textit{leftPart}) = \text{int}\left((18 - 5 + 1) \times \frac{4+8}{4+8+12+2}\right) = 6$$

Thus we can calculate $P(S_{dl})$ and $P(S_{ir})$ from (7) and (8) as: $P(S_{dl}) = 5 + 6 - 1 = 10$ and $P(S_{ir}) = 10 + 1 = 11$

Therefore the DDC will send *Start* messages (*download*, *fileXYZ*, 4000, 10, *decrement*) and (*download*, *fileXYZ*, 4000, 11, *increment*) to DDP_4 and DDP_3, respectively. Accordingly all four DDPs are re-paired to process and send blocks from the first unfinished partition. In addition, they all have a more balanced load based on their previous performance to ensure they finish at about the same time.

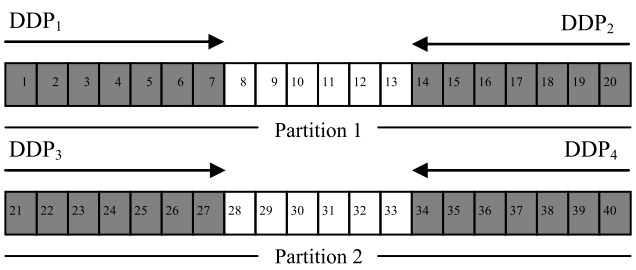

Fig. 3 4 DDPs processing two partitions at relatively equal speed

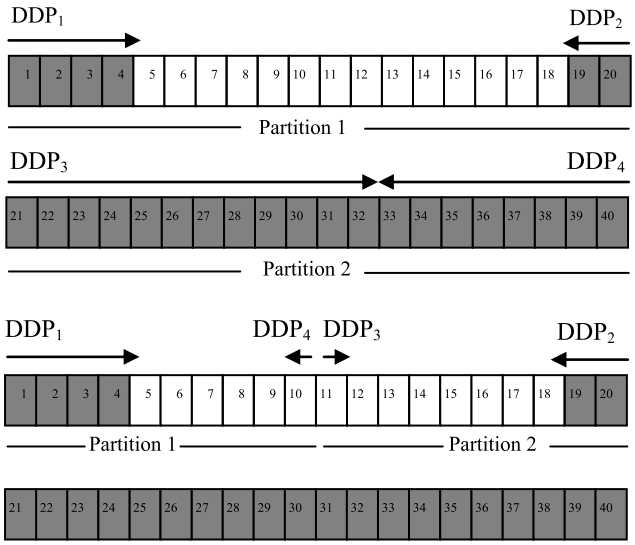

Fig. 4 *Top*: 4 DDPs processing 2 partitions, 2nd partition is completed by DDP_3 and DDP_4 while the 1st partition is still being processed. *Bottom*: rearrangement of DDPs when DDP_3 & DDP_4 start helping DDP_1 and DDP_2. DDP_3 and DDP_2 pair gets more blocks based on the partitioning heuristics

In the following sections (Sects. 5, 6 and 7) we explore three different examples of application domains where DDOps is very useful. We will cover DDFTP (Dual Direction FTP) for parallel file and data set transfers, DDPar (Dual Direction Parallel Processing) to illustrate its use in parallel processing, and DDNet (Dual Direction Network transfer where we use multiple NICs to transfer messages in parallel.

5 Dual-direction FTP (DDFTP)

Replicated FTP servers are available over the Internet to supply users with files for installing new applications such as video games and office applications, for upgrading and maintaining existing applications and systems such as Windows and virus protection software, for entertainment such as music and movies, and for specific information such as financial and geographic information. In addition, replicated FTP servers are available over Grid environments from which different Grid applications and users can download and utilize data files. These files are usually very large and contain important information. To achieve high download rates and good load balancing, it is important to be able to utilize the redundancy available and at the same time minimize the coordination overhead. Applying the technique we described earlier, we can download file partitions from replicated servers (producers) and rely on the characteristics of TCP to help in the ordering and reconstruction of the file from the delivered blocks on the client (consumer) side. The control and coordination is done on the consumer side, which relieves the servers from coordination and allows us to distribute the task among any set of distributed producers regardless of their locations or communications capabilities among each other.

Here, we use the term dual-direction FTP (DDFTP), where we use the DDOps technique for file downloads from replicated FTP servers. As described in the general model, the DDC (in this case the DDFTP client) will first issue a request to receive file information including the file size and the number of available replicas and their locations. Using this information the DDC will determine the number of producer pairs to be used (based on the number of available DDFTP servers), partition the file accordingly and send different *Start* messages to the DDFTP producer (DDP) pairs. Each DDP will find the file and start sending blocks starting from the start block number received and in the direction given by the DDC. The DDC will receive the blocks from each producer in order (courtesy of TCP) and will combine all blocks to build the file. DDC will also verify block numbers and send *End* messages to the DDPs as soon as it confirms receiving all requested blocks.

5.1 Background

The original FTP as described in RFC 959 [22] was designed to support file transfer in a distributed environment using the client/server model and serving a single connection at a time. Over the past few decades many implementations of FTP client and server software were introduced [24] and with these different implementations, several enhancements were made and given the single client/connection design the performance of FTP was improved to its best possible limits. To further improve FTP performance, researchers had to explore more drastic models involving the use of multiple servers/clients, proxy servers, multithreading, and multiple connections. Several ideas were explored and one of the earliest was the GridFTP [6]. GridFTP supports efficient mass data transfer using parallel TCP streams and data striping over multiple servers. Further enhancements and variations of GridFTP were also introduced such as the middleware to enhance reliability of GridFTP [39] and the dynamic parallel data transfer from replicated servers using the GridFTP features [57]. Another potential improvement was done as part of PFTP [11], where striping and parallel connections are used to move data from cluster to cluster. PFTP relies on the availability of a parallel file system such as PVFS (Parallel Virtual File System), which stores large files in trips across multiple cluster nodes. Another research group introduced a PFTP for high performance storage systems (HPSS) [28], where the standard FTP with special extensions is used to enhance file transfer performance for large files. A different approach for parallel file transfer is to use a proxy that controls the process. This proxy could be either close to the client or the server depending on the functionalities needed. One example is discussed in [25], where a proxy handles the client request for file download, retrieves file parts from various servers, orders the parts then delivers them to the client. The main issue here is that received parts are likely to arrive out of order, which may lead to the need for a very large buffer and may also cause delivery delays to the client. To resolve the buffer problem, some researchers suggested the use of variable block size for the download [33]. Also to enhance download times and reduce costs in the Grid environment researchers proposed a dynamic server selection scheme coupled with P2P co-allocation based on available bandwidth values between client and servers [29].

In general, various concurrent and parallel techniques are used to enhance file transfer performance and provide clients with fast and reliable file downloads. However, as in the general case of parallelism, adding more does not always result in a matching improvement level. In most cases the overhead imposed and the control issues result in lower performance gains than expected. As a result many of the models we explored offer good enhancement but still afford more [4]. In the following sub-sections we will further explore our proposed method for parallel file downloads, that will enhance

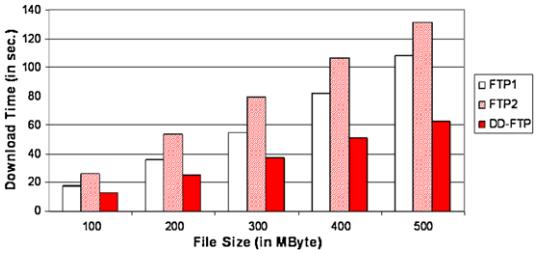

Fig. 5 FTP Vs. DDFTP on LAN with heterogeneous load

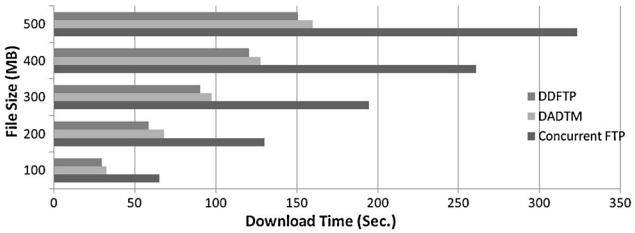

Fig. 6 Concurrent FTP, DADTM & DDFTP with heterogeneous load

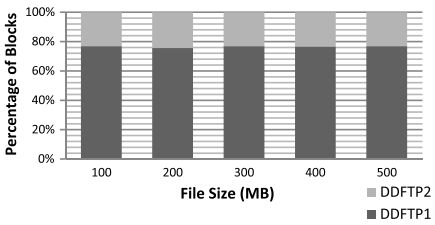

Fig. 7 No. of blocks DDFTP producers processed during download

performance while minimizing overhead. This method may also be combined with some other methods described here to further enhance performance.

5.2 Performance measurements (dual-producers)

We conducted several experiments to evaluate the performance gains of DDFTP for the dual-producer case and we compared the performance of DDFTP with the regular FTP, concurrent FTP and dynamic adaptive data transfer model (DADTM). Concurrent FTP downloads a file replicated on two FTP servers by dividing it into two equal sized partitions and downloading them in parallel. DADTM is an approach using dynamic adaptive data transfer as described in [57]. For all experiments three computers were used: two servers and a client, connected by a wired Ethernet LAN. In the first set of experiments some processing and communication loads were put on the servers while the client was dedicated for the download process. Both regular FTP (on two servers FTP_1 and FTP_2) and DDFTP performances were measured and compared for downloading five files of sizes 100, 200, 300, 400, and 500 MB. The files were stripped into blocks of 4000 bytes each. The download was repeated ten times and the average time was taken. Figure 5 shows very good performance gain for DDFTP over regular FTP and it increases as the files grow larger. We also verified the correctness of the downloaded file by comparing the downloaded and original files and no differences were found. In addition we used archived and executable files for the download, all of which worked correctly after the download.

In the next set of experiments, the performance gains of DDFTP compared to DADTM and concurrent FTP were measured. To implement the DADTM we developed the model with an initial discovery phase to determine the partition sizes, then periodic monitoring to adjust the load according to changes in the environment. DADTM will first request small equal sized blocks from the servers then based on the response time will partition the file relative to the observed performance. To show the performance differences we made the first server very busy with other loads while the second server was left free. As shown in Fig. 6 the performance of DDFTP is at least eight percent (8 %) better than DADTM and over fifty percent (50 %) better than concurrent FTP for all file sizes. Although the percentage over DADTM is not too high, it offers a good improvement based on the LAN setup and current conditions. In addition, we will show in the experiments with multiple servers that this difference increases significantly as the load on the servers vary and network delays increase especially in WAN.

When measuring the DDFTP performance, since the two DDFTP producers had different loads during the download, we also measured the contribution of each server for DDFTP when downloading a file. Figure 7 shows the percentage of blocks on average that were downloaded by each DDFTP producer. As it clearly shows, $DDFTP_2$ did not have a lot of load on it to begin with so it was able to transfer most of the blocks while $DDFTP_1$ was trying to handle the different loads on it in addition to the file transfer thus it managed to transfer less blocks during the same time. In comparison, the concurrent FTP servers had no choice but to each complete the assigned half of the file (50 % of the file each). Therefore, FTP_1 took much longer than FTP_2 to finish the transfer. During the extra time, FTP_2 was idle. As for DADTM the distribution was relatively similar to DDFTP but with slightly less correspondence to the actual load differences among the FTP servers. We will provide detailed experimental results for DADTM in the multiple-server case. If we consider the case where both servers are free and operating at the same level, then the difference between all three methods would be less obvious and the FTP servers will share the load relatively equally. However, in actual FTP operations FTP servers will mostly have varying loads on each one and their operating conditions will be heterogeneous and dynamically changing during the download. Therefore, our technique provides better load balancing in the download efforts and will result in shorter download times.

Table 2 List of FTP servers used and their operational properties

Server #	RTT (s)	500 MB transfer time (s)	Effective bandwidth (MByte/s)
1	0.240	524.747	0.952840
2	0.200	569.095	0.878588
3	0.150	656.064	0.762121
4	0.240	874.578	0.571704
5	0.240	524.747	0.952840
6	0.200	569.095	0.878588
7	0.150	656.064	0.762121
8	0.240	874.747	0.571594

As these experiments show, the overall performance of DDFTP using two servers has improved significantly compared to regular FTP and also showed significant improvement over the concurrent FTP, when the servers and/or their operating conditions are heterogeneous. We also showed that it performs better compared to DADTM. As we mentioned earlier, several research groups proposed dynamic partitioning of files based on network and server conditions; however, they all impose high coordination overhead. This would be fine when using a LAN and generally stable operating environment, but as soon as this moves to a WAN, the propagation delays and other transmission problems will cause the overhead to grow significantly and reduce any possible gains. Our proposed technique in DDFTP does not require any coordination from the DDFTP producers and the only messages exchanged are one *Start* and one *End* message per producer. Producers do not need to synchronize or coordinate with each other nor wait for intermediate instructions from the consumer. The consumer takes care of the organization of the file components locally thus eliminating the need for any communication overhead.

5.3 Performance measurements (k-producers)

To show the performance gains of DDFTP using multiple servers, we conducted several experiments. We used a wide area network (WAN) emulator between the FTP client and the FTP servers to include the effects of WAN such as high return trip time (RTT), limited bandwidth, packet drops, etc. Eight servers were used with the specifications listed in Table 2. Each numbered server has an RTT (to the client), transfer time of a 500 MB file to the client, and the effective bandwidth achieved in the transfer under the current conditions. In all cases, a maximum of 0.240 second RTT is used, which is relatively close to the time it takes a signal to travel between USA and Europe.

Experiments were conducted using four and eight servers to measure performance of file transfer (file size 500 MB) using DDFTP, DADTM and concurrent FTP. Figure 8 shows the results of all experiments. The figure clearly shows the

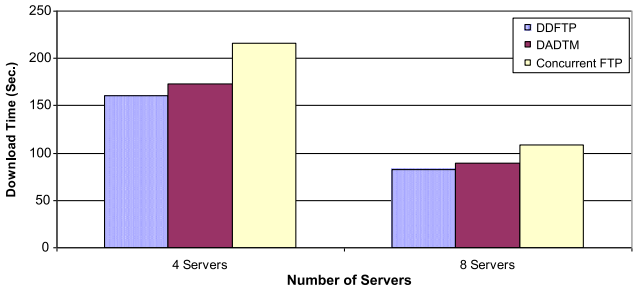

Fig. 8 Download Time (500 MB) with Multiple FTP producers

performance advantage of using DDFTP compared to the other techniques. For example, while it takes concurrent FTP around 216 s and DADTM 173 s to download the file using 4 servers, it takes DDFTP only 160 s to download the same file from the same number of servers. This time drops to 82 s if 4 More servers with similar specifications where added. The speedup ratio between the 8- and 4-server cases using DDFTP is 1.95 which shows significant improvement with minimum loss. Figure 9(a) shows the percentage of downloaded data for DDFTP from each of the producers for 4 and 8 producers. The figures correspond closely to the percentage of performance for each producer used compared to all. It clearly shows how the load balancing happens among the producers based on their current capabilities and loads. As a result, we can achieve a high load balancing without adding coordination overhead. In concurrent FTP the servers are assigned fixed sized partitions; therefore, the slower of the servers will hold up everyone else until it is done, which explains the overall delay. When load balancing is introduced based on periodic measurements as in DADTM (Fig. 9(b)), the percentage of data transfer becomes more proportional to the loads in each server in a similar way as in DDFTP. As a result, we conclude that DDFTP achieves as good or better load balancing on non-dedicated heterogeneous servers. However, the overall performance of DADTM is lower as it involves initial measurements and periodic updates adding more delays.

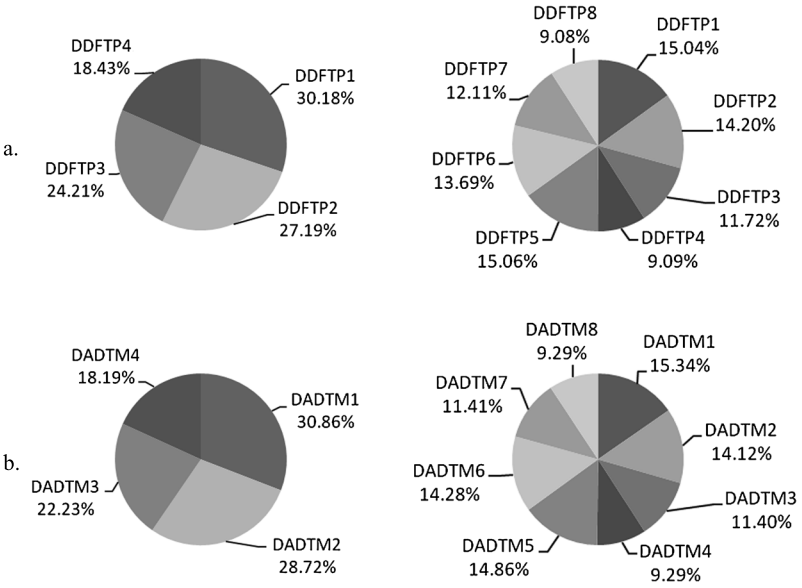

Fig. 9 Percentage of transferred blocks in: (**a**) DDFTP: 4 producers (*left*) and 8 producers (*right*) and (**b**) DADTM: 4 servers (*left*) and 8 servers (*right*)

5.4 Analysis and discussion

Here we introduced an innovative method based on DDOps to parallelize file downloads from distributed FTP servers and achieve good load balancing with minimal coordination overhead. DDFTP uses dual direction operations as described earlier and frees the producers from dealing with coordination or control issues. Furthermore, each producer handles the assigned load as best as its current resources and operational environment allows. Therefore, we do not have to monitor the operational attributes continuously to adjust the load. The producers need minimum information and work completely independently of each other. Therefore, changes in network properties will not severely harm their download operations; on the contrary, each producer will deliver blocks according to its current capabilities and faster producers will automatically compensate for the slower ones.

As the experiments show, with DDFTP we can decrease the download time from multiple producers significantly by allowing each pair of producers download in opposite directions. This basically eliminates the need to monitor and adjust download conditions for each producer and requires no coordination between them. This becomes very suitable for the current situation on the Internet, Cloud and Grid environments where FTP servers are heterogeneous, scattered in different locations and operate under varying network and operational conditions. The overall gain is high mainly because we were able to minimize idle time for all producers. Each producer receives its transfer orders in the *Start* message independently and continues working until it receives the *End* message from the consumer. The consumer, on the other hand, orchestrates the operations and rebuilds the file. However, the overall effort is minimized and localized. The DDFTP approach can be combined with other methods currently used for concurrent file downloads. For example, DDFTP can be easily incorporated within the GridFTP to further enhance download time among different partitions and servers. It could also be enhanced by introducing a proxy server that will take over the DDFTP consumer operations and make the whole download distribution transparent to the user.

6 Dual-direction parallel processing (DDPar)

The general approach for efficient load balancing of parallel applications on non-dedicated heterogeneous environments, where multiple problems execute concurrently, is to use advanced dynamic scheduling and load balancing techniques. This requires continuous monitoring of resources before and during run time. The gathered information is used for task/data assignments and reassignments to achieve effective dynamic load balancing. As an alternative approach to monitoring, the problem can be divided into multiple partitions and processors solve one partition then take the next until all partitions are processed. Thus, the faster processors will solve more partitions than the slower ones. This requires coordination among the processors which causes delay and reduces the expected overall gains. The Grid and the Cloud are heterogeneous environments and usually non-dedicated thus both continuous monitoring and partitioning are very costly due to the high communication delays among the resources.

Using DDOps we contribute in relieving the application from having to handle this issue by changing the way the problem tasks are executed. Using the dual direction approach allows the resources to independently work on different tasks of the problem from opposite directions and not

worry about changes in the environment. However, as we described in Sect. 3, we can apply this technique on specific types of parallel applications that have relatively independent tasks and well defined boundaries. The dual-direction parallel computation (DDPar) technique distributes task processing among the available producers. If necessary, it will also replicate the programs and data needed for the task if it is not already available on the producers' machines. The DDC gives each DDP the instructions to start processing from a specific point in the problem in the direction. In this manner the consumer will form pairs of processing units to handle each partition from two opposite directions. Each producer in the pair will work as slow (or as fast) as its own resources and operating conditions allows, yet they both will complete their work at the same time. The DDC keeps track of the results of these computations and informs the pairs when to stop processing and also reassigns pairs to help others when necessary. The main advantage of using DDPar is that there is no need to monitor resources and reallocate tasks when the environment changes and load balancing is achieved regardless of the changes that may occur.

6.1 Background

Parallelization is used to enhance the performance of compute-intensive problems. Multiple processors are used in parallel to reduce the computation time for large problems such as image and video processing, scientific simulations, data visualization, and design optimizations. Some of these problems are easily dividable and they can be easily parallelized using multiple processors/machines while others have less parallelization degrees and require complex techniques to achieve good parallelization gains. However, in any parallel application achieving good performance also relies on the scheduling and load balancing efforts. These efforts are simple when dealing with a stable homogenous environment, however it is challenging for non-dedicated heterogeneous environments. In this case it is hard keep track of and control the resources available all the time. An example of the importance of proper partitioning to achieve load balancing is dealing with sparse matrices [13]. In such example, processing is not balanced among the different processes due to the varying distribution of the elements; therefore, a model to study the initial distribution and partition the data accordingly is introduced. Using DDPar, it becomes unnecessary to do that since the pair that will receive a less dense section of the matrices, will finish faster and be reassigned to help other pairs quickly. Another way to achieve load balancing is using graph partitioning based on the resources capabilities and operating conditions [8].

As we will use parallel matrix multiplication (PMM) as our example to demonstrate DDPar, we explore here some relevant models and techniques used. Several PMM algorithms are available for parallel machines or homogeneous networks of workstations [2, 12, 23, 37]. In addition there are a few for heterogeneous environments such as [9]. This algorithm was designed with the aim of enhancing load balancing for heterogeneous environments and reducing communication overhead among the heterogeneous machines. The algorithm works well as long as the environment's operational conditions do not change during run time. However, it is not suitable for non-dedicated heterogeneous environments because it cannot change the load distribution after execution begins. The authors in [7] experimented with different implementations of PMM on heterogeneous clusters of multi-core processors to find the best scheduling strategy. The authors in [5] experimented with matrix multiplication using the master/slave model on heterogeneous clusters where matrices are divided into smaller blocks. These blocks are distributed among available slaves based on their dynamic speed. These approaches and several others rely on some knowledge either prior or during execution to adjust the load or they adopt the master/slave model to divide the multiplication task into smaller sub problems. Generally, we may achieve load balancing but at the expense of high overhead. Our technique reduces the problem by limiting the overhead to achieve better load balancing on non-dedicated heterogeneous resources.

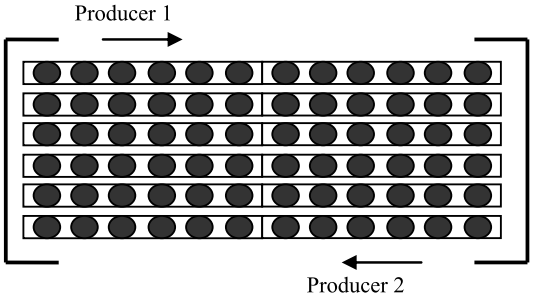

Fig. 10 Result matrix with two producers

6.2 Parallel matrix multiplication with DDPar

In this section we illustrate the benefits of DDPar to enhance load balancing among parallel tasks on non dedicated heterogeneous environments. We will use the PMM problem to demonstrate the technique as it satisfies the requirements described in Sect. 3. It is well defined in a geometric form, has known boundaries and can be easily partitioned. The sub-problems are highly independent therefore the producers can work with minimum interference from each other. DDPar makes the parallelization of matrix multiplication effective on both local environments such as clusters as well as highly distributed environments with high communication delays such as Grid and Cloud environments. DDPar can be applied on the PMM application as shown in Fig. 10. This figure represents the result matrix of the multiplication process. The result matrix can be divided into blocks of equal

Table 3 Machines specifications

Machine #	Specifications
M1	Desktop, Microsoft Windows XP Professional, Intel® Core 2 CPU 6400 @ 2.13 GHz, 3.00 GB of RAM, Gigabit Ethernet NIC
M2	Laptop, Microsoft Windows XP Professional, Intel® Core 2 Duo T7300 @ 2.00 GHz, 0.99 GB of RAM, Gigabit Ethernet NIC
M3	Laptop, Microsoft Windows 7, Intel® Core i5 CPU M450 @ 2.4 GHz, 4.00 GB of RAM, Fast Ethernet NIC
M4	Laptop, Microsoft Windows 7, Intel® Core i7 CPU Q740 @ 1.73 GHz, 6.00 GB of RAM, Fast Ethernet NIC

sizes as shown in the figure. Each computing node in the system is considered as a resource that can be used to solve part of the problem (producer). Multi-core machines have multiple producers. In the dual-producer case, one producer will start calculating the result blocks from the first block (top left corner) while the second will start from the last block (bottom right corner). The consumer will ask the producers to stop processing as soon as it has the full matrix. If the resulting matrix is of size m by p and each block consists of l cells, then there are $(m \times p)/l$ blocks in the result matrix. If there are more than two producers, then the result matrix is divided into multiple partitions. In this case, each pair of producers will work on one partition. For example, with four producers, the result matrix is first divided into two halves and each half is processed by a pair as in dual-producer.

6.3 Performance measurements (dual-producer)

To evaluate DDPar for the PMM problem, four multi-core machines were used with the system specifications listed in Table 3. These machines were connected by a dedicated LAN using Dell 2324 Fast/Gigabit Ethernet switch.

Two sets of experiments were conducted to evaluate the performance of DDPar on PMM with dual producers. In the first set, machine M1 was the consumer while machines M2 and M3 where the producers. Three square matrices of sizes 1000, 1500 and 2000 were used. The results are shown in Fig. 11, where Mat(M2) is for the result when one producer from M2 is used. Mat(M3) is for the result when one producer from M3 is used. Eq-Part(M2, M3) is the result when the problem is divided into two equal parts solved by M2 and M3. Finally, DDPar(M2, M3) is the result when DDPar is used with one producer from M2 and one from M3. As shown in Fig. 11, DDPar achieves better processing times for all matrix sizes and it enhances further with larger matrix sizes.

Another set of experiments was conducted to compare the performance of PMM using DDPar and the M-Blocks approach. In the M-Blocks approach the master divides the matrices into multiple blocks. These blocks are assigned by the master to the producers each time they finish from a block. In this case the faster slaves will solve more blocks.

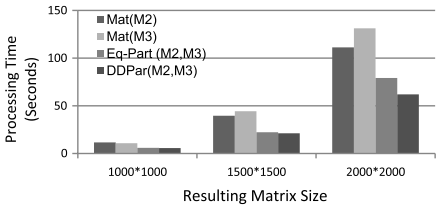

Fig. 11 Performance of PMM

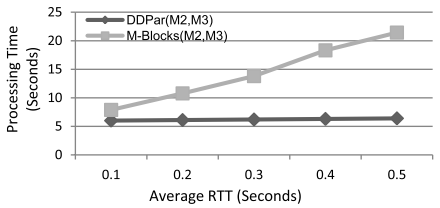

Fig. 12 Effects of long RTT in PMM

The main function of the master is to distribute the blocks and to combine the results. The M-Blocks approach involves some coordination costs for assigning blocks to the slaves several times and involves idle time for the slaves while waiting for new assignments. The experiment was conducted with an emulated WAN embedded between the machines. The main aim of this experiment is to evaluate the effect of long RTT on the performance of both DDPar and M-Blocks. As shown in Fig. 12, the processing times of DDPar slightly increase as RTT increases while the processing time of M-Block increases significantly as RTT increases. This is mainly due to the cost of coordination in M-Block, where slaves need to go back to the master every time they finish a block to get the next one. The cost of coordination in DDPar is kept at minimum since the producer continues its work and it does not need to wait for the consumer during execution.

6.4 Performance measurements (k-producer)

The following experiments were done to evaluate PMM with DDPar for multiple producers. One set of experiments was conducted to evaluate the performance of DDPar in the k-producer case with $k = 4$ and $k = 8$ on a LAN and compare

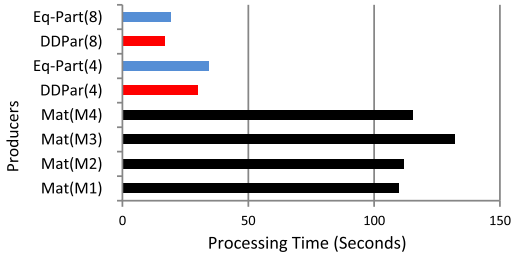

Fig. 13 DDPar performance of PMM, k-procedure case

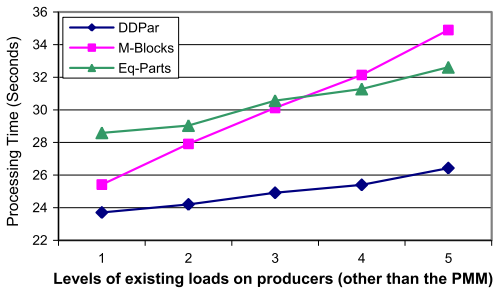

Fig. 14 DDPar performance in a non-dedicated environment

it to the equal partition method (Eq-Part). The results are shown in Fig. 13. The M1 through M4 bars represent the execution time for PMM on a single processor on each machine. For the parallel version M4 was used as the consumer while one producer was used from each machine for the 4-producer case, DDPar(4), and two producers were used from each machine for the 8-producer case, DDPar(8). The speedup for DDPar(4) is 4.38 with respect to the slowest machine M3 and 3.66 with respect to fastest machine M1. The speedup for DDPar(8) is 7.90 with respect to slowest machine M3 and 6.59 with respect to fastest machine M1. Compared to Eq-Part, we achieved better performance since each machine was handling load proportional to its capabilities, while in Eq-Part, each machine has to handle the same amount of load regardless of its capabilities. As a result, the overall performance is always governed by the slowest machine in the group.

In another experiment, we measured the effect of having varying loads on the producers on the overall performance. We compared DDPar with M-Blocks and Eq-Part using eight producers with imposed load on four of them. The load levels as shown in Fig. 14 represent the burstiness of the load on the producers. The amount of load imposed was relatively constant; however, the variations of its distribution have been increased in each level. Assuming for example that the load imposed takes up around 40 % of the resources available on each of the four producers, in level one, this extra load is uniformly distributed among the producers with minimum changes in the levels during run time. On each following level, the load variation is increased such that by level 5, the load level on the four machines varies very quickly and in unpredictable bursts. As Fig. 14 shows, the overall performance of DDPar is better than the others and as the load level variations increase, DDPar becomes more efficient and maintains better performance. In Eq-Part, the load variations cause additional delays, however they do not increase drastically since each producer handles its own fixed part regardless of the others. However, in M-Blocks, the performance decreases drastically as the load variations increase since each producer has to communicate with the consumer with every completed set of blocks. Therefore, as the variations increase, the producers lose more time waiting for the next set of blocks.

6.5 Analysis and discussion

When dealing with parallel applications, one of the major causes of low gains is communications delays and poor load balancing. Although this may be easily manageable in homogeneous environments and relatively easy to handle in dedicated heterogeneous environments, it is extremely difficult to manage in heterogeneous non-dedicated environments. Many dynamic load balancing techniques offer good results if certain conditions are applied. Examples of such conditions are low communication delays, predictable load levels on the nodes and stable operating conditions. With DDPar, we enhance the performance and inherently achieve effective load balancing among the resources without imposing additional overhead. The experiments demonstrated the overall advantage of using DDPar even when high communication delays are involved and varying load levels on the machines were introduced. The simple, yet innovative work assignment from the two ends of the problem offers numerous advantages: (1) It allows each node (producer) to work at its own pace. (2) It eliminates the need for the producers to consult with the consumer to get more work. (3) It allows the consumer to identify and reassign faster producers to help others effectively. (4) Any change in the operating conditions of the node where the producer is working will be automatically reflected on the producer's performance and eventually be controlled by the producer without having to constantly monitor them. (5) Even in homogeneous environments, when the problem is unbalanced (e.g. processing sparse matrices), there is no need to adjust data assignments since producers will automatically work on different parts at different paces and be reassigned to help others when necessary. Therefore, load balancing is always achieved with minimal overhead.

7 Dual-direction message transfer (DDNet)

Here we discuss using the DDOps technique as a middleware-level message striping approach (DDNet) to increase

the communication bandwidth for data transfer in heterogeneous clusters equipped with multiple network interface cards (NICs). Currently network sockets can handle one connection at a time and if the machine has more than one NIC, only one is used per application. In our earlier work [41], we introduced a new technique to allow an application to use multiple NICs to increase bandwidth by stripping the messages and distributing the stripes (blocks) among the available NICs. Here we enhance the method to reduce the overhead. Using TCP we are assured of the correctness and order of received packets. Therefore, the sender does not need to number the blocks (no additional header needed) and the receiver will not have problems managing the arriving blocks. In addition, by applying DDNet to send blocks from opposite directions, we reduce contention on resources.

In the Cloud and Grid environments, the infrastructure usually consists of multiple highly distributed high performance equipment connected either locally on high speed LANs or globally over high bandwidth WANs. In addition, most computers are now equipped with multiple NICs. Yet, the communication bandwidth is mostly governed by the available bandwidth in the network and the maximum bandwidth the NIC can achieve. As a result, it is feasible to find ways to utilize multiple NICs on the machines to increase the bandwidth. Using the DDNet approach we can divide the messages into blocks and assign them to different NICs on the machine and the receiving end receives the blocks over one or more NICs then combine them to reconstruct the original message. Since block transmission is done from opposite directions, we do not worry about the NICs differences in performance and we do not need remote coordination and synchronization between them. Using DDNet at the middleware-level, where the middleware acts as the consumer, provides a scalable, portable, and flexible solution to increase bandwidth among heterogeneous systems. This technique is devised to balance the load so that slower networks will still contribute in the message transfer, yet at a slower pace. DDNet relies on the middleware (DDC) to distribute the blocks, ensure correct transmission and stop it when all blocks arrive at the destination. The sending and receiving ends are the producers as together they complete the transmission of the messages.

7.1 Background

Efforts are being made to utilize existing multiple NICs on computers to enhance the communication performance of distributed applications. One important approach is the striping technique [52]. Striping is well known and understood in the context of storage systems, for example, the redundant arrays of inexpensive disk (RAID) architecture [32]. Striping at lower network levels has been implemented in a number of research projects, such as the IBM project for striping PTM packets [50], and the Stripe, which is an IP packets scalable striping protocol [1]. However, these techniques are network and/or system dependent. In addition, GridFTP [26] uses multiple TCP streams to improve aggregate bandwidth over a single TCP stream in wide area networks. In addition, GridFTP can transfer data from multiple servers where data is partitioned to improve performance. Some work as in [1] and [50] has also been done to minimize the overhead of striping at lower network layers. This work uses queues at the receiving end of the channels to maintain synchronization between sender and receiver to reorder the packets. Another example of using striping in clusters is channel bonding [10], which is implemented at the kernel level and it provides high bandwidth and high throughput for the applications, thus reducing transfer time for medium to large messages [14]. Since channel bonding is implemented in the lower network layers, it requires specific hardware configurations and additional addressing scheme that restrict its utilization and impose several limitations [17]. Moreover, most techniques introduced do not adequately address load balancing in heterogeneous environments. Our model, MuniSocket [42], solves the problem with minimum interference.

7.2 Performance measurements (dual-producers)

To evaluate the performance gains of DDNet, a number of experiments were conducted using MuniSocket that allows an application to simultaneously use the available multiple NICs [41] and using Enhanced MuniSocket that includes the DDNet technique. To measure the achievable round-trip transfer time we used the ping-pong benchmark [21]. All experiments were conducted in sets done in multiple repetitions and the average results for each set were taken. Then, the sets were repeated and the peak averages of all sets were reported. This method allows us to avoid considering the results that are impacted by other load on the networks when we need not consider it. All experiments were conducted on a 24-node cluster, where each node is equipped with two Fast Ethernet (FE) NICs connected to two FE networks. The experiments measure the round trip time (RTT) and the effective bandwidth is derived as:

$$\textit{Effective Bandwidth}\ (\text{Mbps}) = \left(8 \times \textit{Message size}/10^6\right)/(\textit{RTT}/2) \tag{9}$$

In Enhanced MuniSocket using DDNet the senders will each keep a separate counter for the blocks and will increase or decrease it for the next block depending on the direction it was assigned. An array of status values (one per block) is used to synchronize the point where the senders stop processing more blocks. Each entry in the array represents a block status. The value 0 represents unsent block while value 1 represents in-transit or sent block. Each sender

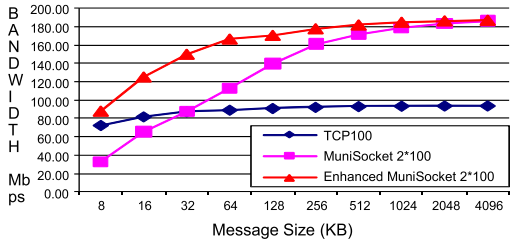

Fig. 15 Effective bandwidth of TCP, MuniSocket, and Enhanced MuniSocket

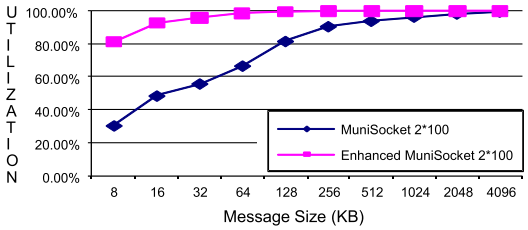

Fig. 16 Bandwidth utilization of MuniSocket and Enhanced MuniSocket

tries to get and change the corresponding block status value from the array. As soon as the entry is granted to a sender the value is changed to 1. Since senders start from opposite directions, they both can stop as soon as they encounter a sent block.

Experiments were conducted to evaluate the performance of Enhanced MuniSocket over two Fast Ethernet (FE) networks. With Enhanced MuniSocket, the overall performance improved noticeably with most message sizes (see Fig. 15). While MuniSocket (MuniSocket 2*100) only achieved speedup with messages of size 32 KB and larger, Enhanced MuniSocket achieved speedup for messages of size 8 KB and beyond. The speedup of Enhanced MuniSocket for a 64 KB message is around 1.86 with respect to TCP100 and 1.47 with respect to MuniSocket. The speedup of Enhanced MuniSocket for a 2 MB message is around 1.98 with respect to TCP100 and 1.02 with respect to MuniSocket. This shows that the original MuniSocket imposes overhead due to the use of block headers and the senders contention on the blocks. While Enhanced MuniSocket eliminates this overhead and increases load balancing. Figure 16 shows the bandwidth utilization of both MuniSocket and Enhanced MuniSocket, normalized to the maximum achievable peak. While the effective bandwidth of MuniSocket approaches the maximum achievable peak with a 0.5 MB message, Enhanced MuniSocket achieves this with a much smaller message (16 KB).

Another set of experiments was conducted to measure the load balancing effect achieved by Enhanced MuniSocket using two FE networks. The first network had no load, while the second had some load generated by messages of size 32 KB being exchanged frequently between the

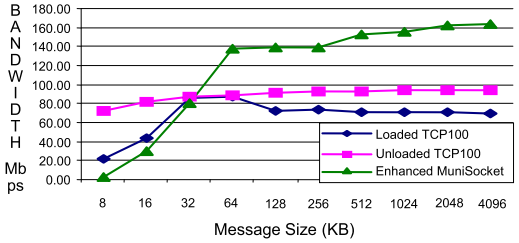

Fig. 17 Effective bandwidth of TCP and enhanced MuniSocket on loaded network

two nodes. Figure 17 shows the average effective bandwidth achieved with a TCP socket, using the first (Unloaded TCP100) and the second (Loaded TCP100) network, respectively. In addition, it shows the average effective bandwidth achieved with Enhanced MuniSocket. Enhanced MuniSocket achieved speedups with messages of size 40 KB and larger and offered dynamic load balancing with the varying load on the networks. The results also show that while the loaded network provides less than 75 percent of its peak bandwidth, MuniSocket still achieved high bandwidth gain with dynamic load balancing. That is, with a 2 MB message, for example, an average bandwidth of 82.123 Mbps was obtained with either network interface individually. However, Enhanced MuniSocket achieved an average bandwidth of 161.84 Mbps.

7.3 Performance measurements (k-producers)

DDNet provides an efficient solution for load balancing network traffic across multiple NICs as well as the dual-NIC case demonstrated in the previous section. Here we show the performance results for DDNet on more than two NICs. As described in the general approach, DDOps, when multiple NICs are available, a matching number of senders/receivers pairs are created. The message is divided into the same number of partitions and each pair handles one partition in dual directions as described for the dual NICs case. Load balancing is achieved by assigning free pairs to help others when the load varies. We used the same cluster and benchmark described earlier to evaluate the performance of Enhanced MuniSocket over four FE networks. Two nodes were equipped with two NICs in addition to the original two NICs each had. Both MuniSocket (MuniSocket 4*100) and Enhanced MuniSocket (Enhanced MuniSocket) with four channels were evaluated. The results, in terms of peek effective bandwidth, are shown in Fig. 18. Enhanced MuniSocket provides better performance than MuniSocket by a margin of 35 % for messages of size 128 Kbyte and of 7.6 % for 2 MB messages. These experiments also show that DDNet provides good scalability and flexibility to increase message transfer effective bandwidth.

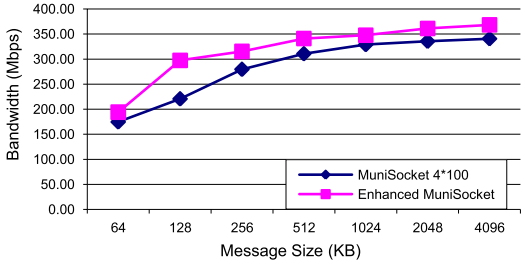

Fig. 18 Effective bandwidth of MuniSocket and Enhanced MuniSocket on four Fast Ethernet networks

7.4 Analysis and discussion

As DDNet allows us to increase the effective bandwidth of transmission by utilizing the available multiple NICs, it also results in a good balancing among used NICs and networks. As shown in the experiments, we were able to achieve high bandwidth and good load balancing across dual and multiple NICs. When the network attributes and operating conditions are similar across the distributed environment, the DDNet technique (Enhanced MuniSocket) performs slightly better. However, as soon as the network attributes and operating conditions vary, we see higher gains with DDNet. The main reason is that with varying conditions some NICs are able to transmit several blocks within a time period, while others have other loads and are able to transmit fewer blocks within the same time frame. However, the overall performance will improve relative to the average capacity currently available. All this is achieved without having to monitor the networks or keep track of the variations in loads and operating conditions. Therefore, DDNet offers an efficient mechanism to enhance network performance and at the same time ensure good load balancing without adding any significant overhead. In addition, DDNet does not require coordination among the different threads; therefore, there will be no contention delays or coordination overhead.

8 DDOps analysis

After discussing the DDOps technique and demonstrating its benefits in three different domains (File download, parallel computation and multiple networks message transfer), we discuss additional characteristics and evaluate the general performance. As we described earlier, DDOps imposes minimal overhead while achieving very good load balancing. To evaluate the overhead imposed, let us consider the dual-producer case first. In the dual-producer case, we need one *Start* message with starting location (counter's initial block number) and counter mode to be sent to each producer. In addition, we need only one *End* message to each producer to stop processing. Thus only four control messages are needed regardless of the problem size or the number and size of the blocks used. For the k-producer scenario, where $k > 2$ and $k \ll n$, and n is number of blocks in the problem, we consider the best and average case scenarios. The best case is where all producer pairs finish processing their blocks at the same time. Thus the consumer needs to send k *Start* messages at the beginning and k *End* messages when all the blocks are completed. The complexity of the best case scenario is $O(k)$, Thus, for small k, which is the usual operational case, the complexity is constant.

The average case analysis provides a good projection of the behavior of the proposed approach in real-life where the communication and processing workload on the networks and producers are dynamic. A simple approximation of the average case scenario can be considered when processing the partitions consists of j load-balancing rounds. In each round multiple recursive steps are performed by different producer pairs. In addition, in each round, we assume that on average half of the producer pairs finished processing their blocks while the rest only processed an average of half of their blocks. This implies that about a quarter of the total blocks is not processed at the end of the current round and need to be processed in the following rounds. Therefore, j will be equal to $\log_4(n)$ rounds. In the first round, the consumer will send k *Start* messages to the k producers, while during the rest of the rounds only $k/2$ *Start* message need to be sent by the consumer to the producers. Thus the number of *Start* messages in the average case is:

$$\textit{Number of Start Messages} = k + \frac{k}{2}\big((\log_4 n) - 1\big) \qquad (10)$$

This means that $O(k \times \log_4(n))$ *Start* messages need to be sent in the average case. Similarly the same number of *End* messages will need to be sent, which keeps the total number of messages within the same order calculated in Eq. (10). To verify our analysis, we experimentally counted the number of *Start* messages needed in the k-producer case. In all experiments, we used DDFTP with two files of sizes 100 MB and 500 MB to be downloaded using four servers and eight servers. Three block sizes were used 500 Bytes, 2500 Bytes, and 4000 Bytes. The results are shown in Table 4.

The results show that as the number of blocks increases, the number of the *Start* messages increases slightly. In addition, as the number of producers increases, the numbers of *Start* messages increases too. However, for different file sizes, the number of *Start* messages stays as it is if the number of servers and blocks are the same. Figure 19 shows the average number of *Start* messages needed in the average case for different scenarios. Another issue to consider is how DDOps performance is affected by high communication delays and variations in load levels on the machines. Experiments were done to measure these effects using DDFTP as an example. In one set we measured the file download time using DDFTP, DADTM and concurrent FTP (described

Table 4 Number of start control messages under different scenarios

File size (MB)	# Servers	Block size (Bytes)	#*Blocks*	Actual range of start msgs	# Start msgs calculated from (9)
100	4	4000	26215	8–16	16.7
100	8	4000	26215	22–30	33.4
100	4	500	209716	14–18	19.7
100	8	500	209716	26–36	39.4
500	4	2500	209716	14–18	19.7
500	8	2500	209716	32–36	39.4
500	4	500	1048576	16–19	22
500	8	500	1048576	29–36	44

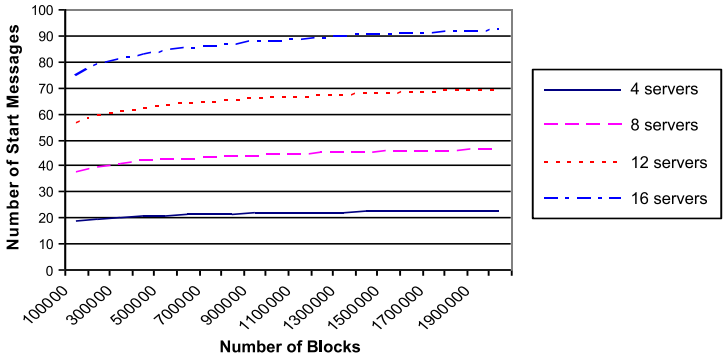

Fig. 19 Load balancing steps under different scenarios

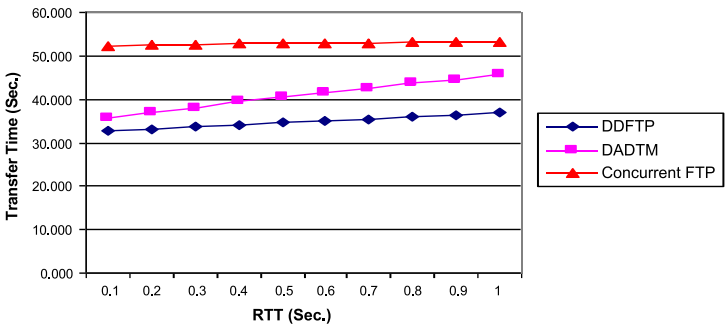

Fig. 20 The effect of increased RTT on File download time using a 100 MB file and 4 producers

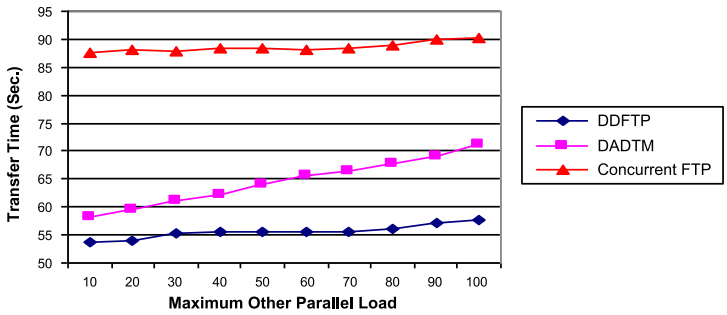

Fig. 21 Effect of varying load on producers on file download time using a 100 MB file and 4 servers

in Sect. 5) for a fixed file size, with increasing RTT (see Fig. 20).

Experiments were also done to evaluate the performance when each of the producers and its associated network is subjected to varying levels of load throughout the download time. This shows how non-dedicated machines would behave in a Grid or Cloud environment. Here we introduced several concurrent tasks distributed randomly on the machines where our producers operate to occupy some part of the resources including the network bandwidth. Figure 21 shows the overall results again showing the least effect with DDFTP. The number of parallel load in the figure represents

Table 5 Average client CPU utilizations

Servers types & transfer approach	Transfer time (s)	Average CPU utilization
Homogenous servers & DDFTP	66	6.27 %
Homogenous servers & DADTM	72	5.89 %
Homogenous servers & concurrent FTP	65	5.42 %
Heterogeneous servers & DDFTP	54	6.67 %
Heterogeneous servers & DADTM	59	6.15 %

the maximum number of tasks randomly introduced across all machines. For example 10 parallel loads mean that there may be 5 tasks on one server and its network connections, three on another, two on a third and nothing on the remaining machines or it could be that all 10 tasks are on one machine and everyone else is free. This imposes a random dynamic load on each machine similar to what they would experience on normal day to day operations. The higher the number of parallel tasks imposed, the busier the machines would be. In this case for concurrent FTP, the overall download time increased with the existing load since we cannot really change the fact that the slowest of the machines will hold everyone else. Therefore, as soon as one machine experiences high load during the download, the whole process will be slowed even if the other machines are not busy. As for DADTM, the performance is good; however, it still cannot match DDFTP since the periodic monitoring to change the load usually cannot fully keep up with the dynamic changes. This extra time for monitoring and adjustments reflects in the overall performance. In addition, the effect increases even more as the load increases on the machines since more adjustments will be required. In DDFTP, each producer works independently without having to adjust download parameters. Thus, each producer will use whatever resources available to it to achieve the best possible performance. Therefore, the overall performance remains at the same level with increasing load on the machines.

Since the DDFTP client (DDC) has a lot of work to do controlling the producers and organizing the results, we need to measure how much overhead this imposes on the DDC. An experiment was conducted to compare the overhead of different approaches used on the DDC machine. In this case we used DDFTP for parallel file download as an example, thus the DDC will request a download operation. In this experiment, a file of size 500 MB was used. The experiment has two parts to compare the overhead with and without using load balancing. The first part was conducted using homogenous servers and network capabilities. Each server can transfer 1 MB/s. In this part the load balancing mechanism will not be used as the servers and connecting networks are homogenous. The second part was conducted using eight servers with heterogeneous capabilities. The servers can transfer between 0.6 MB/s to 2 MB/s. In this experiment, load balancing is activated as the servers are heterogeneous. In the first part, the DDC CPU usage values during the data transfer were recorded for three download approaches: dual-direction (DDFTP), adaptive (DADTM) and concurrent (Concurrent FTP) approaches. In the second part, the DDC CPU usage during the data transfer for DDFTP and DADTM were recorded as in the case of concurrent FTP the overhead is always constant regardless of the environment's conditions. The results are shown in Table 5 and show that there is little overhead on the client's CPU usage for the three approaches. In addition, there are little differences between DDFTP and DADTM; and the overhead slightly increases in the heterogeneous cases. This is due to the involvement of the load balancing mechanism overheads. Although, DDFTP has slightly more overhead on the client compared to the other approaches, it still provides better load balancing that results on better transfer times. As the heterogeneity of the environment increases and its operating conditions change dynamically, we achieve better performance with DDFTP and overhead is less significant.

DDOps offers a simple, yet an elegant approach to achieve load balancing among parallel tasks while minimizing the control overhead and dodging the effects of high communication delays. For environments like the Grid and Cloud, this is particularly important since the resources are heterogeneous, non-dedicated and highly distributed. The analysis of control overhead and the experimental evaluations show that we can achieve excellent results in different settings and various environments (local clusters on LAN and distributed resources over WAN). This is achievable due to the low overhead which make the technique more stable even when high delays and variable loads exist in the environment.

9 Conclusion

In this paper we presented a new approach for performance enhancement and dynamic load balancing among distributed cooperative processes that work on a single dividable problem on non-dedicated heterogeneous environments. This approach, based on dual direction operations (DDOps), minimizes the number of messages needed for coordinating the

processing nodes (producers). In addition, it does not require any direct communication or coordination among the producers. This is done by dividing the work among the producers which are grouped in pairs such that each pair will work on its part from opposite directions. This allows each producer to continue its work without having to consult with the consumer or the other producers. In addition, as each producer in a pair works towards the other, the load will be automatically evenly distributed among the two based on their speed and they will both finish at the same time. This makes DDOps suitable for load balancing among parallel processes in any type of distributed environments, yet it shows its merits best in heterogeneous non-dedicated environments. This approach works perfectly for dual as well as for multiple producers. As we have demonstrated, DDOps can be used in various application domains; however, there are certain requirements where it will offer the best results. These requirements include having easily dividable problems, well defined problem boundaries, and multiple producers with replicated resources (e.g. multiple processors for parallel computations or replicated file servers for parallel file download). We used the parallel file download from replicated file servers as one example, demonstrated the concept as it supports utilizing multiple NICs to transfer messages in parallel as another example, and described it in parallel computations using PMM as a third example. The experimental evaluation has shown DDOps's efficiency and low overhead in several scenarios. DDOps proved to be useful in parallelizing large tasks and it solves load balancing issues facing non-dedicated heterogeneous environments such as highly distributed systems and the Grid and Cloud environments.

References

1. Adiseshu, H., Parulkar, G., Vargese, G.: A reliable and scalable striping protocol. In: Computer Communication Review, October 1996 vol. 26, pp. 131–141. ACM, New York (1996)
2. Agarwal, R., Gustavson, F., Zubair, M.: A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. IBM J. Res. Dev. **38**(6), 673–681 (1994)
3. Al-Azzoni, I., Down, D.G.: Decentralized load balancing for heterogeneous grids. In: Proc. of the International Conference on Future Computing, pp. 545–550 (2009)
4. Al-Jaroodi, J., Mohamed, N.: DDFTP: dual-direction FTP. In: Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid), Newport Beach, CA, USA, May 2011
5. Al-Jaroodi, J., Mohamed, N., Jiang, H., Swanson, D.: Middleware infrastructure for parallel and distributed programming models on heterogeneous systems. IEEE Trans. Parallel Distrib. Syst. **14**(11), 1100–1111 (2003)
6. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S., Foster, I.: Secure, efficient data transport and replica management for high-performance data-intensive computing. In: Proc. of the IEEE 18th Symposium on Mass Storage Systems and Technologies, San Diego, CA, USA, April 2001
7. Alonso, P., Reddy, R., Lastovetsky, A.: Experimental study of six different implementations of parallel matrix multiplication on heterogeneous computational clusters of multicore processors. In: Proc. of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (2010)
8. Aubanel, E.: Resource-aware load balancing of parallel applications. In: Handbook of Research on Grid Technologies and Utility Computing: Concepts for Managing Large-Scale Applications, May 2009, pp. 12–21. IGI Global, Hershey (2009)
9. Beaumont, O., Boudet, V.: Matrix multiplication on heterogeneous platforms. In: IEEE Transaction on Parallel and Distributed Systems, vol. 12, pp. 1033–1051 (2001)
10. Beowulf Ethernet channel bonding. http://www.phy.duke.edu/~rgb/brahma/Resources/beowulf/software/bonding.html. Accessed Apr 2013
11. Bhardwaj, D., Kumar, R.: A parallel file transfer protocol for clusters and grid systems. In: Proc. of the 1st International Conference on e-Science and Grid Computing (e-Science), pp. 248–254 (2005)
12. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: a linear algebra library for message-passing computers. In: Proc. of the 8th SIAM Conference on Parallel Processing for Scientific Computing, (PPSC), Minneapolis, Minnesota, USA, March 1997, pp. 14–17 (1997)
13. Boman, E.G., Catalyurek, U.V., Chevalier, C., Devine, K.D., Safro, I., Wolf, M.M.: Advances in parallel partitioning, load balancing and matrix ordering for scientific computing. J. Phys. Conf. Ser. **180**, 1 (2009)
14. Channel bonding performance. http://www.scl.ameslab.gov/Projects/MP_Lite/perf_cb_fe.html. Accesssed Apr 2013
15. Chiasson, J., Tang, Z., Ghanem, J., Abdallah, C.T., Birdwell, J.D., Hayat, M.M., Jérez, H.: The effect of time delays on the stability of load balancing algorithms for parallel computations. IEEE Trans. Control Syst. Technol. **13**(6), 932–942 (2005)
16. Chung, I., Bae, Y.: The design of an efficient load balancing algorithm employing block design. J. Appl. Math. Comput. **14**(1–2), 343–351 (2004)
17. Configuring Ethernet channel bonding. http://www.linuxtopia.org/online_books/linux_system_administration/redhat_cluster_configuration_and_management/generated-index.html. Accessed Apr 2013
18. De Giusti, A.E., Naiouf, M.R., De Giusti, L.C., Chichizola, F.: Dynamic load balancing in parallel processing on non-homogeneous clusters. J. Comput. Sci. Technol. **5**(4), 272–278 (2005)
19. Dhakal, S., Hayat, M., Pezoa, J.E., Yang, C., Bader, D.A.: Dynamic load balancing in distributed systems in the presence of delays: a regeneration-theory approach. IEEE Trans. Parallel Distrib. Syst. **18**(4), 485–497 (2007)
20. Eggen, M., Franklin, N., Eggen, R.: Load balancing on a non-dedicated heterogeneous network of workstations. In: Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 856–862 (2002)
21. Figueira, S., Berman, F.: A slowdown model for applications executing on time-shared clusters of workstations. IEEE Trans. Parallel Distrib. Syst. **12**(6), 653–670 (2001)
22. File transfer protocol, RFC 959. http://www.faqs.org/rfcs/rfc959.html. Accessed Apr 2013
23. Fox, G., Otto, S., Hey, A.: Matrix algorithms on a hypercube I: matrix multiplication. Parallel Comput. **3**, 17–31 (1987)
24. FTP client software and FTP server software. http://en.wikipedia.org/wiki/Comparison_of_FTP_client_software, http://en.wikipedia.org/wiki/Comparison_of_FTP_server_software. Accessed Apr 2013
25. Funasaka, J., Kawano, A., Ishida, K.: Implementation issues of parallel downloading methods for a proxy system. In: Proc. of the

4th International Workshop on Assurance in Distributed Systems and Networks (ICDCSW), vol. 1, pp. 58–64 (2005)
26. GridFTP: universal data transfer for the grid, The Globus Project, documentation. The University of Chicago and The University of Southern CA, USA. http://www.globus.org/alliance/publications/papers.php. Accessed Apr 2013
27. Hargrove, W., Hoffman, F.: Optimizing master/slave dynamic load balancing in heterogeneous parallel environments. White paper, Oak Ridge National Laboratory, Oak, Ridge, TN (1999)
28. HPSS user's guide—high performance storage system, release 7.1, 2009 International Business Machines Corporation, USA, Feb. 2009
29. Hsu, C.-H., Chu, C.-W., Chou, C.-H.: Bandwidth sensitive co-allocation scheme for parallel downloading in data grid. In: Proc. of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 34–39 (2009)
30. Johnsen, E.S., Anshus, O.J., Bjørndalen, J.M., Bongo, L.A.: Survey of optimizing techniques for parallel programs running on computer clusters. Technical report, Department of Computer Science, University of Tromsø, September (2003). http://www.idi.ntnu.no/~elster/notur-cluster03/NOTUR_ET_Cluster_A12b_ifi_uit_survey.pdf. Accessed Apr 2013
31. Kathiresan, N.: Adaptive distributed load balancing for heterogeneous computing system using global scheduler policy. In: Proc. of the 4th International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks, UK, September 2006
32. Katz, R., Gibson, G., Patterson, D.: Disk system architectures for high performance computing. Proc. IEEE **77**(12), 1842–1858 (1989)
33. Kawano, A., Funasaka, J., Ishida, K.: Parallel downloading using variable length blocks for proxy servers. In: Proc. of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW) (2007)
34. Khan, S.U., Ahmad, I.: A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids. IEEE Trans. Parallel Distrib. Syst. **20**(3), 346–360 (2009)
35. Kliazovich, D., Bouvry, P., Khan, S.U.: DENS: data center energy-efficient network-aware scheduling. Clust. Comput. **16**(1), 65–75 (2013)
36. Kołodziej, J., Khan, S.U., Wang, L., Byrski, A., Min-Allah, N., Madani, S.A.: Hierarchical genetic-based grid scheduling with energy optimization. Clust. Comput. (1994). doi:10.1007/s10586-012-0226-7
37. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Benjamin-Cummings, Redwood City (1994)
38. Li, Y., Lan, Z.: A survey of load balancing in grid computing. In: Computational and Information Science. Lecture Notes in Computer Science, vol. 3314, pp. 280–285 (2005)
39. Lim, S.B., Fox, G., Kaplan, A., Pallickara, S., Pierce, M.: GridFTP and parallel TCP support in NaradaBrokering. In: Distributed and Parallel Computing. Lecture Notes in Computer Science, vol. 3719, pp. 93–102 (2005)
40. Mateescu, G.: Parallel sorting on heterogeneous platforms. In: Proc. of the 16th International Symposium on High Performance Computing Systems and Applications, pp. 116–117, June 2002
41. Mohamed, N., Al-Jaroodi, J.: Self-configured multiple-network-interface socket. J. Netw. Comput. Appl. **33**(1), 35–42 (2010)
42. Mohamed, N., Al-Jaroodi, J., Jiang, H., Swanson, D.: A middleware-level parallel transfer technique over multiple network interfaces. In: Proc. of the ClusterWorld Conference & Expo, San Jose, CA, June 2003
43. Pinar, A., Hendrickson, B.: Improving load balance with flexibly assignable tasks. IEEE Trans. Parallel Distrib. Syst. **16**(10), 956–965 (2005)
44. Pınar, A., Tabak, E.K., Aykanat, C.: One-dimensional partitioning for heterogeneous systems: theory and practice. J. Parallel Distrib. Comput. **68**, 1473–1486 (2008)
45. Pinel, F., Pecero, J.E., Bouvry, P., Khan, S.U.: A two-phase heuristic for the scheduling of independent tasks on computational grids. In: Proc. of the ACM/IEEE/IFIP International Conference on High Performance Computing and Simulation (HPCS), Istanbul, Turkey, July 2011, pp. 471–477 (2011)
46. Pinel, F., Pecero, J.E., Khan, S.U., Bouvry, P.: Utilizing GPUs to solve large instances of the tasks mapping problem. In: Proc. of the International Research Workshop on Advanced High Performance Computing Systems, Cetraro, Italy, June 2011
47. Pinel, F., Dorronsoro, B., Pecero, J.E., Bouvry, P., Khan, S.U.: A two-phase heuristic for the energy-efficient scheduling of independent tasks on computational grids. Clust. Comput.. doi:10.1007/s10586-012-0207-x (2012).
48. Qureshi, K., Rehman, A., Manuel, P.: Enhanced GridSim architecture with load balancing. J. Supercomput. (2010). doi:10.1007/s11227-010-0402-6
49. Rus, P., Stok, B., Mole, N.: Parallel computing with load balancing on heterogeneous distributed systems. Adv. Eng. Softw. **34**(4), 185–201 (2003)
50. Theoharakis, V., Guerin, R.: SONET OD-12 interface for variable length packets. In: Proc. of the 2nd International Conference on Computer Communication and Networks (1993)
51. Transmission control protocol, RFC 793. http://www.faqs.org/rfcs/rfc793.html. Accessed Apr 2013
52. Traw, C.B.S., Smith, J.: Striping within the network subsystem. IEEE Netw. **9**(4), 22–32 (1995)
53. Tudruj, M., Borkowski, J., Koponski, D.J.: Load balancing with migration based on synchronizers in PS-GRADE graphical tool. In: Proc. of the 4th International Symposium on Parallel and Distributed Computing, July 2005, pp. 105–112 (2005)
54. Valentini, G.L., Lassonde, W., Khan, S.U., Min-Allah, N., Madani, S.A., Li, J., Zhang, L., Wang, L., Ghani, N., Kolodziej, J., Li, H., Zomaya, A.Y., Xu, C.-Z., Balaji, P., Vishnu, A., Pinel, F., Pecero, J.E., Kliazovich, D., Bouvry, P.: An overview of energy efficiency techniques in cluster computing systems. Clust. Comput. **16**(1), 3–15 (2013)
55. Yagoubi, B., Medebber, M.: A load balancing model for grid environment. In: Proc. of the 22nd International Symposium on Computer and Information Sciences, Turkey, pp. 1–7 (2007)
56. Zamanifar, K., Nematbakhsh, N., Sadjady, R.S.: A new load balancing algorithm in parallel computing. In: Proc. of the 2nd International Conference on Communication Software and Networks (ICCSN), pp. 449–453 (2010)
57. Zhang, Q., Li, Z.: Data transfer based on multiple replicas in the grid environment. In: Proc. of the 5th Annual ChinaGrid Conference, Guangzhou, China, July 2010, pp. 240–244 (2010)

Nader Mohamed is an associate professor at The College of Information Technology, United Arab Emirates University, Al-Ain, UAE. He obtained his Ph.D. in Computer Science from University of Nebraska-Lincoln, Nebraska, USA in 2004. He was an assistant professor of Computer Engineering at Stevens Institute of Technology in New Jersey, USA. His current professional interest focuses on middleware, Internet computing, sensor networks, and cluster, Grid, and Cloud computing. He published

more than 100 refereed articles in these fields. His research was published in prestigious international journals such as IEEE Transactions on Parallel and Distributed Systems (TPDS), Journal of Parallel and Distributed Computing (JPDC), International Journal of High Performance Computing Applications, and Concurrency and Computation: Practice and Experience.

Jameela Al-Jaroodi received her Doctor of Philosophy degree in Computer Science from the University of Nebraska-Lincoln, USA in 2004. She then joined the Stevens Institute of technology in New Jersey, USA as a research assistant professor where she embarked on several research projects. In 2006 she moved to the Faculty of Information Technology, at the United Arab Emirates University, UAE, where she took the position as an assistant professor. Currently, her research interests involve middleware, distributed collaborative systems, information systems, distributed systems, Cloud computing and pervasive computing. Her research generated over 80 refereed articles in international Journals and conferences such as Concurrency and Computation: Practice and Experience, IEEE Transactions on Distributed Systems, The Journal of Network and Computer Application (JNCA), Cluster, Cloud and Grid Computing (CCGrid) and IEEE International Conference on Cluster Computing. Dr. Al-Jaroodi served as guest editor for several special issues in middleware in Concurrency and Computation, JNCA and Journal of Software. In addition she organized and co-chaired the First and Second International Symposiums on Middleware and Network Applications.

Hong Jiang received the B.Sc. degree in Computer Engineering in 1982 from Huazhong University of Science and Technology, Wuhan, China; the M.A.Sc. degree in Computer Engineering in 1987 from the University of Toronto, Toronto, Canada; and the Ph.D. degree in Computer Science in 1991 from the Texas A&M University, College Station, Texas, USA. Since August 1991 he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, USA, where he is Willa Cather Professor of Computer Science and Engineering. At UNL, he has graduated 12 Ph.D. students who upon their graduations either landed academic tenure-track positions in Ph.D.-granting US institutions or were employed by major US IT corporations. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, performance evaluation. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACM-TACO, JPDC, ISCA, MICRO, USENIX ATC, FAST, LISA, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD and the State of Nebraska. Dr. Jiang is a Senior Member of IEEE, and Member of ACM.