Use Only What You Need: Judicious Parallelism For File Transfers in High Performance Networks

Md Arifuzzaman
University of Nevada, Reno
Reno, Nevada, USA
marifuzzaman@unr.edu

Engin Arslan
University of Nevada, Reno
Reno, Nevada, USA
earslan@unr.edu

ABSTRACT

Parallelism is key to efficiently utilizing high-speed research networks when transferring large volumes of data. However, the monolithic design of existing transfer applications requires the same level of parallelism to be used for read, write, and network operations for file transfers. This, in turn, overburdens system resources since setting the parallelism level for the slowest component results in unnecessarily high parallelism for other components. Using more than necessary parallelism lead to increased overhead on system resources and unfair resource allocation among competing transfers. In this paper, we introduce modular file transfer architecture, `Marlin`, to separate I/O and network operations for file transfers so that parallelism can be independently adjusted for each component. `Marlin` adopts online gradient descent algorithm to swiftly search the solution space and find the optimal level of parallelism for read, transfer, and write operations. Experimental results collected under various network settings show that `Marlin` can identify and use a minimum parallelism level for each component, improving fairness among competing transfers and CPU utilization. Finally, separating network transfers from write operations allows `Marlin` to outperform the state-of-the-art solutions by more than $2x$ when transferring small datasets.

ACM Reference Format:
Md Arifuzzaman and Engin Arslan. 2023. Use Only What You Need: Judicious Parallelism For File Transfers in High Performance Networks. In *Proceedings of ACM Conference (ICS'23).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Distributed science projects such as Large Hadron Collider [6] and Vera Rubin Observatory [5] require high-performance data transfers in the orders to tens of gigabits-per-second to move data between geographically distant locations in timely manner. Research networks (e.g., Internet-2 and ESnet) provide high-speed connectivity between research and education institutions with up to 100*Gbps* bandwidth to separate scientific data transfers from internet traffic, thereby facilitating large-scale data movements. However, legacy file transfer applications (e.g., SCP and FTP) fail to reach high utilization in these networks due to employing one file-at-a-time approach, which limits their I/O and network throughput significantly.

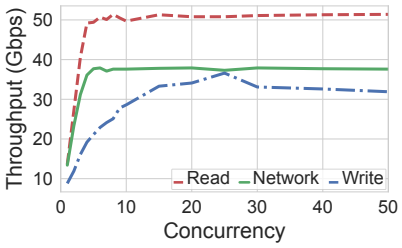


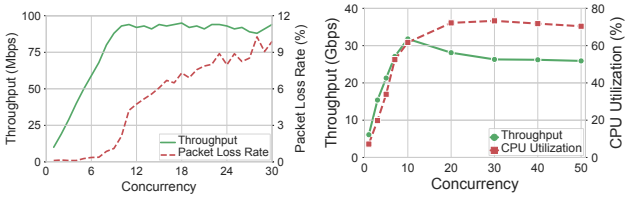
Figure 1: While transfer parallelism (aka concurrency) is necessary to achieve high transfer throughput, its optimal level is not the same for all components of file transfers.

A typical solution to overcome the limitations of legacy transfer applications is transferring multiple files simultaneously (henceforth concurrency) as it can improve aggregate I/O throughput by reading and writing multiple files and network throughput by creating multiple network connections [11, 12, 20, 22, 24]. However, the monolithic architecture of existing file transfer applications requires the same level of concurrency to be used for I/O and network operations, incurring unnecessary overhead and causing unfair resource allocation. Figure 1 presents the throughput of read, write, and network operations for increasing concurrency in a network with 40Gbps bandwidth and 1 ms delay. For read and write measurements, we create multiple processes to read or write from/to a local file system (consisting of a RAID array of NVMe SSDs). For network operation, on the other hand, we transfer dummy data from the memory of the source node (`/dev/zero`) to the memory of the destination node (`/dev/null`). When the concurrency level is set to 1, network transfer obtains around 13Gbps, the read process attains 16Gbps, and the write operation achieves 8Gbps. The throughput of the network transfer and read operations reach to $40 - 50$ Gbps when concurrency is set to 4. On the other hand, write operation yields can reach 36 Gbps when the concurrency is set to 25. As a result, while $3 - 4$ concurrency is needed for read and network operations to reach maximum speed, as much as 25 concurrency is necessary to achieve the maximum performance for the write operation.

Despite the speed mismatch between read, write, and network operations, the implementation of concurrency in existing file transfer applications causes the same level of parallelism for all three operations. In the above example, setting the concurrency value to around 25 to maximize write operation performance will create 25 transfer threads/processes in the source node to read 25

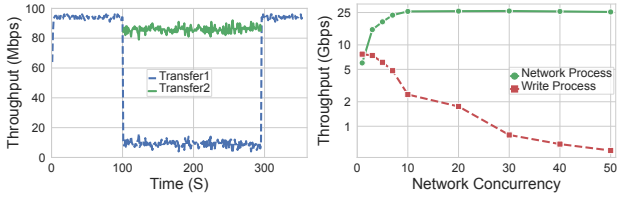
ICS'23, June 21-23 2023, Orlando, Florida, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Increased Packet Loss (b) High CPU Utilization

Figure 2: While concurrency is needed to increase transfer throughput, choosing an arbitrarily high value increases network congestion (a) and end host CPU utilization (b).



(a) Unfairness Among Transfers (b) Adverse Impact on I/O Workload

Figure 3: Setting concurrency value based on the slowest transfer operation leads to unfair resource sharing between transfers (a). It also adversely affects other processes running on the same end hosts due to increasing I/O contention (b).

separate files from the file system and transfer them using 25 separate network connections. Similarly, 25 threads/processes will be created on the destination node to receive and write data to the file system. Although using a high level of concurrency is mostly harmless to the performance of read and network operations, it can have adverse impacts on resource usage, increase I/O contention, and cause unfairness among competing flows with different I/O characteristics.

Resource Overhead: To demonstrate this, we evaluate the performance of a file transfer when concurrency is set to values between 1 and 32 in a simple network where sender and receiver nodes are connected via two switches. While sender and receiver nodes are connected to switches with 1*Gbps* links, the two switches are connected with a 100 Mbps link, limiting end-to-end network bandwidth to 100 Mbps. We throttle disk read throughput to 10 Mbps per process to emulate the behavior of parallel file systems in which concurrent I/O access (using multiple threads) is necessary to achieve high I/O performance. Since the network bandwidth is limited to 100 Mbps, ten concurrent transfers are needed to obtain 100 Mbps aggregate I/O throughput, thereby reaching to maximum possible transfer speed. Although creating more than ten concurrent transfers does not degrade the transfer throughput considerably, it significantly (from 2% to 10%) increases packet loss due to network congestion at the bottleneck link, as presented in Figure 2(a).

Moreover, high concurrency overburdens end hosts and storage systems due to creating too many threads/processes. Figure 2(b) shows the relationship between transfer throughput and sender host CPU utilization in a 40 Gbps bandwidth network. When the concurrency is set to the optimal value of 10, the transfer yields 31 Gbps throughput, and CPU utilization is around 60%. On the other hand, setting the concurrency to larger values increases CPU utilization by around 10%.

Unfair Resource Sharing: The monolithic design of existing transfer applications also leads to unfair resource sharing when concurrency is used. Figure 3(a) shows the throughput of two transfers between two separate server pairs. The transfers share a network link with a capacity of 100 Mbps. We throttled the read I/O throughput of each thread to 10Mbps for Transfer-2 to simulate increased I/O performance by means of concurrency. Transfer-1, on the other hand, does not have any I/O limitations and can attain close to 1Gbps read/write I/O throughput using a single transfer thread. Assume that users are aware of existing bottlenecks and set the concurrency level to optimal values, which is 1 for Transfer-1 (because single I/O and network thread is sufficient to attain reach 100Mbps throughput) and 10 for Transfer-2 (since it needs 10 threads to increase read I/O throughput to 100Mbps). When the first transfer starts, it obtains 100Mbps throughput by transferring one file at-a-time (i.e., concurrency=1). When Transfer-2 joins, it sets its concurrency level to 10 to overcome the I/O limitation and attain maximum throughput. However, a concurrency value of 10 requires 10 connections to be created due to the monolithic architecture of the transfer applications. This, in turn, causes unfair bandwidth allocation since Transfer-2 creates more network connections and thus yields nearly 90% of the available bandwidth.

We also tested the impact of using an unnecessarily high concurrency value on the performance of other applications on the same node. To do so, we run a process on the receiver host that writes 100GiB to a file while the transfer application is running. We then measure the transfer throughput and the execution time of the write process. Figure 3(b) shows that the transfer throughput reaches the maximum at a concurrency level of 10, at which point the write process attains 2.4Gbps throughput. Increasing concurrency significantly degrades the performance of the write process due to increasing I/O contention. Specifically, the throughput of the write process drops to less than 1Gbps when the concurrency is set to 30 and 0.5 Gbps when the concurrency of the transfer is set to 50. *Therefore, while concurrency is necessary for speeding up file transfers in high-speed networks, its current implementation results in increased resource usage and unfair resource sharing.*

Although researchers proposed solutions to separate I/O and network operations for file transfers (e.g., mdtmFTP [26] and FDT [3]) to overcome the limitations of monolithic designs, these solutions require manual tuning for concurrency to perform well. In addition, they solely focus on increasing the throughput of transfers without considering system overhead (e.g., memory footprint and network congestion); hence they fall short of offering fully automated, low-overhead alternatives to existing solutions. Thus, in this work, we introduce a modular file transfer architecture, `Marlin`, to tune concurrency for read, transfer, and write operations independently. `Marlin` utilizes a game-theory-inspired utility function with an online optimization algorithm to discover the optimal concurrency level for each component. The utility function is used to evaluate the fitness of different concurrency values in terms of increasing throughput and decreasing resource consumption (i.e., minimal number of threads/processes and low network packet loss). Since

it is crucial to swiftly scan the solutions space for concurrency levels for I/O and network operations in real-time, we implemented Gradient Descent and Bayesian Optimization algorithms that can converge to the optimal in $10 - 15$ search intervals.

Our extensive evaluations using both isolated and production systems show that `Marlin` mostly obtains similar throughput compared to the state-of-the-art solutions that use the same concurrency level for both I/O and network operations despite significantly minimizing system overhead. We also show that it addresses the fairness issue between competing transfers with different I/O performance behavior. Finally, we show that `Marlin` can speed up the transfer of small transfers by more than $2x$ when write I/O is the bottleneck as it can take advantage of high network performance to transfer files to the cache space (e.g., main memory or NVMe buffer) on the receiver end. In summary, the contributions of this paper are as follows:

- We introduce a modular file transfer framework, `Marlin`, to separate read, transfer, and write operations of file transfers to overcome the limitations of the legacy monolithic design of state-of-the-art solutions. We show that the modular architecture allows transfers to use "just enough" concurrency to keep the system overhead low and ensure fairness while achieving high transfer throughput.
- We develop a game theory-inspired utility function to evaluate the performance of different concurrency values for read, transfer, and write operations. We implement an online gradient descent algorithm to quickly and accurately discover component-specific concurrency values in real-time.
- Through extensive experiments, we show that `Marlin` can automatically discover the slowest operation of file transfers and tunes its concurrency in real-time to maximize the performance while keeping the concurrency for other operations at a minimum. We also show that `Marlin` provides fair resource sharing between competing transfers, which is critical for production systems to ensure shared network resources are allocated to users/jobs fairly.
- Finally, we show that `Marlin` can speed up the throughput of small transfers (i.e., less than stage-in area capacity) by more than $2x$ when transfers are write-limited by quickly transferring data to cache space on receiver ends. Since more than 80% of transfer jobs in research networks transfer less than 100GiB, optimizing them is crucial in offering performance enhancements to most transfers [21].

2 RELATED WORK

Most existing work on optimizing wide-area data transfer focused on designing new congestion control algorithms such as BBR [16] and Vivace [17]. BBR achieves higher performance than legacy TCP variants such as TCP Cubic in the presence of random packet losses. However, since file transfers in high-speed networks often face I/O performance limitations, improving the performance of congestion control algorithms is not sufficient to overcome the performance issues in today's high-performance networks.

Researchers proposed application-layer optimization solutions such as pipelining transfer commands [15], creating parallel network connections [18], transferring multiple files concurrently [9], and distributing transfer load to multiple DTNs [7] to address the performance problems of file transfers. However, finding the optimal transfer setting in a timely manner has risen as a challenging problem due to having a large search space and the slow nature of evaluating various settings in real-time. Previous work proposed heuristic [8, 13], historical analysis [10, 11, 22], and real-time optimization [9, 24, 25] approaches to discover the optimal configuration for some of the transfer settings. As an example, Prasanna et al. proposed direct search optimization to dynamically tune transfer parameters on the fly based on measured throughput for each transferred chunk [14].

Globus [4] is a widely-adopted data transfer service that schedules, maintains, and optimizes large data transfers in high-speed networks. It either relies on system administrators to configure transfer settings or uses a heuristic method to estimate the values of some application-layer transfer parameters such as command pipelining, network connections, and concurrent file transfers. To avoid overwhelming end system and network resources, it typically underestimates the value of some critical settings, such as the number of concurrent file transfers, and thus falls short of achieving high performance for most transfers. Yun et al. proposed ProbData [25] to tune the number of parallel streams and buffer size for memory-to-memory TCP transfers using stochastic approximation. ProbData can identify near-optimal configurations, but it takes several hours to find a solution, which makes it impractical to use as most transfers in high-speed networks only last for a few minutes [21].

Yildirim et al. proposed PCP [24] to tune the values of command pipelining, network connections, and concurrent file transfers using a simple hill-climbing method. Since PCP explores the optimal values of parameters sequentially (i.e., one parameter at-a-time), it is neither fast nor precise. Arslan et al. proposed heuristic [12, 13] and historical data-based (HARP [11]) models to determine the transfer settings for file transfers that can maximize the throughput. While heuristic models fail to guarantee high performance, the performance of historical data-based solutions is bound to the availability of large-scale, up-to-date historical data collected under various background loads and transfer settings. Gathering rich training datasets in a periodic manner is, however, a daunting task for isolated networks and nearly impossible for production systems. Arifuzzaman et al. developed an online learning model, Falcon, to discover the optimal concurrency for file transfers that can maximize the transfer throughput while ensuring fairness among competing flows [9]. While Falcon addresses fairness issues when competing transfers have similar file system configurations (i.e., the same level of I/O parallelism is needed for competing transfers), it fails to do so when transfers have different I/O characteristics. Furthermore, Falcon adopts a monolithic transfer architecture and hence fails to address the wasteful use of concurrency. Fast Data Transfer (FDT) [3] and Multicore-aware Data Transfer Middleware (mdtmFTP) implemented the idea of separating I/O and network operations; however, they both require users to tune transfer settings such as the number of concurrent I/O and network threads and memory size. This is a challenging task even for domain experts, as the optimal settings change over time.

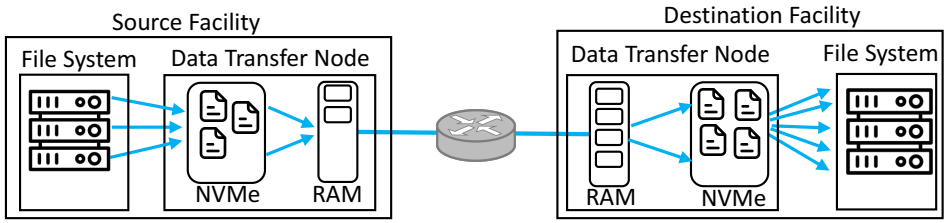


Figure 4: `Marlin` introduces modular file transfer architecture to enable fine-granular, adaptive parallelism for end-to-end transfers.

3 MARLIN: MODULAR FILE TRANSFER APPLICATION

To scale file transfers to high speeds while avoiding to overload system resources, we build a modular file transfer application, `Marlin`, as illustrated in Figure 4. Specifically, `Marlin` separates I/O and network operations at source and destination servers to be able to tune their parallelism independently. This allows it to take advantage of I/O and network parallelism to increase transfer throughput while avoiding unnecessary parallelism on well-performing components.

When selecting a concurrency level for read, transfer, and write operations, `Marlin` uses two criteria: low overhead and high performance. In other words, it searches for "just enough" concurrency values for each operation that leads to close-to-maximum performance using as minimal concurrency as possible. Hence, we adopted *utility function* proposed in [9] that rewards high throughput and penalizes high concurrency. While [9] searches for one concurrency value for read, transfer, and write operations due to using a monolithic transfer architecture, `Marlin` tunes each component separately. Thus, `Marlin` extends the utility function to meet the unique design of the proposed modular transfer architecture. Since identifying the optimal concurrency level for read, transfer, and write operations quickly is key to reaching maximum and stable transfer speed, `Marlin` uses online gradient descent algorithm as it can converge to the optimal solution with only a few sample transfers. We next discuss the details of the utility function and online optimization algorithms.

3.1 Utility Function

Utility functions are used to quantify the fitness of a configuration in terms of maximizing the benefit and minimizing the cost. In the context of file transfers, we aim to maximize the throughput and minimize the number of concurrent read/write threads and network connections. Including a punishment term into a utility function does not only help to lower the resource overhead, but also play an important role to converge to a fair and optimal solution in the presence of competition [19, 23, 27]. Hence, we adopted the following utility function as proposed in [9]

$$u(n_i, t_i, L_i) = \frac{n_i t_i}{K^{n_i}} \tag{1}$$

where n_i is the number of concurrent files to transfer (i.e., concurrency) and t_i is the average throughput of each file transfer, and K is a constant coefficient that is used to determine the severity of punishment for the concurrency level. Although previous studies show that the utility functions that incorporate monotonically increasing penalty terms in linear form guarantee high performance for single transfer and optimal and fair convergence for competing transfers (i.e., Nash Equilibrium) [19, 27], it is challenging to achieve both high-performance and fair and optimal convergence when the penalty for concurrency is incorporated in a linear form as shown in [9]. Thus, we adopted a nonlinear form for concurrency penalty that experimentally satisfies both higher performance and fairness between competing agents. As the throughput improvement ratio is not directly proportional to increased concurrency (i.e., the ratio of gain starts to lower at higher concurrency values), the value of K can be tuned to require small but non-negligible gain (e.g., 1%) for increasing concurrency values. By doing so, we ensure that the utility will increase as long as a non-negligible amount of throughput gain is observed and decrease upon exceeding the optimal concurrency value.

While the utility function given in Equation 1 is sufficient to lower resource overhead for read and write I/O operations, it is not sufficient for network transfers since it does not capture the impact on the network adequately. More specifically, one can attain higher network throughput with increased concurrency at the expense of causing or exacerbating congestion in the network. Hence, we incorporated the packet loss ratio as an additional cost for the transfer operations to keep the network congestion at a minimum. Please note that while congestion control algorithms already take packet loss into account while determining a sending rate (i.e., congestion window) for transfers, they do it on individual connection levels. This in turn does not capture the full impact of concurrent file transfers (that are part of the same transfer job) on the network. As an example, a file transfer that uses a concurrency level of 10 for network connections can lead to a high (e.g., 1%) packet loss rate despite individual network connections experiencing a relatively small packet loss rate (e.g., 0.1%). Consequently, we calculate a total packet loss rate for all network connections and add it to the utility function as a penalty term to lower the severity of network congestion as

$$u(n_i, t_i, L_i) = \frac{n_i t_i}{K^{n_i}} - n_i t_i L_i \times B \tag{2}$$

where B is a constant coefficient that is used to determine the severity of punishment for packet loss penalty. We observe that $B = 10$ works well for the most commonly used TCP variants (i.e., TCP Cubic and Reno, and HSTCP) by keeping packet loss rate below $1-2\%$ while achieving over 95% network utilization [17]. As a result, the utility function in the form of Equation 2 can be used to prevent high packet losses caused by suboptimal concurrency settings.

As Marlin is designed to tune the concurrency level for read, transfer, and network operations to different values, we have two main options in designing a search algorithm. In the first approach, we can come up with a single utility function that combines the performance of each operation to produce a single value and utilize multi-parameter optimization algorithms (e.g., conjugate gradient descent and Bayesian optimization) to search for the optimal concurrency for all operations simultaneously. In the second approach, we can use a separate utility function and search algorithm to independently tune the concurrency level for read, network, and write operations. For the first approach, the utility function needs to reward increased throughput for read throughput, network throughput, and write throughput while penalizing increased concurrency level for each operation as well as increased packet loss rate. Hence, we can calculate the utility of each operation using Equation 1 and 2 as

$$u(n_i, t_i, L_i) = \frac{t_r}{K^{n_r}} + \frac{t_n}{K^{n_n}} + \frac{t_w}{K^{n_w}} - t_n L \times B \tag{3}$$

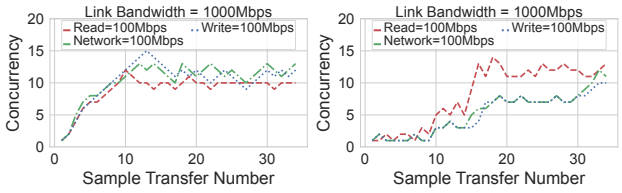
where t_r, t_n, t_w are the throughput of read, transfer, and write operations; n_r, n_n, n_w are the concurrency level of read, transfer, and write operations, and L_{n_n} is the packet loss rate. We show in the evaluations that the convergence speed of optimization algorithms is very slow when using a combined utility function. This is mainly because of the dependence between the concurrency levels, which prevents evaluating some settings. As an example, if the network speed is faster than the read I/O throughput, it may not be possible to evaluate a setting with a large network concurrency level and a small read I/O concurrency level due to a lack of data in the stage-in area. Hence, Marlin uses a separate optimization approach to tune the concurrency level for each operation independently. It uses Equation 1 for read and write I/O operations and Equation 2 for network operation.

Utility functions in the form of Equation 1 and Equation 2 converge to a fair and optimal state in the presence of multiple competing transfers due to being in the concave form. The term $1 - L_i \times B$ in Equation 2 follows a monotonically decreasing pattern for the increasing number of concurrent transfers since packet loss either stays the same or increases as the number of concurrent connection increase. Thus, both Equations 1 and 2 are guaranteed to be concave as long as $\frac{n_i t_i}{K^{n_i}}$ is concave. It is proved in [9] that for a value of $K = 0.02$, $\frac{n_i t_i}{K^{n_i}}$ is guaranteed to be strictly concave as long as n_i less than 100, which we find to be sufficient in all production networks.

3.2 Online Search Algorithm

Naive algorithms such as brute force search may be feasible for scanning small search spaces or when the cost of evaluating a setting is minimal. However, neither of these conditions is valid for file transfers as the search space is very large, and it takes several seconds to test a concurrency setting accurately. Hence, it is essential to devise a search algorithm that can quickly converge to the optimal solution.

Online Gradient Descent (OGD) is known to accelerate optimization processing significantly with the help of adaptive step size. OGD works by testing two close settings and calculating the gradient of the utility of these settings. As an example, assume that we test the concurrency values of 1 and 2 in two consecutive intervals.



(a) Univariate Gradient Descent (b) Conjugate Gradient Descent

Figure 5: Comparison of univariate and conjugate gradient descent algorithms in terms of convergence speed. It takes more than two times longer for the conjugate gradient to find a solution.

We then calculate the utility value for these two concurrency levels, say u_1 and u_2 using the utility function, and calculate the gradient using $\frac{u_2 - u_1}{2-1}$. The gradient value is then used to decide which direction to continue the search as well as how big of a jump to make. As an example, if the gradient value is large, OGD can jump to a concurrency value of 10 instead of testing 4 or 5. By doing this, it can take large steps when the current values are far from optimal.

Moreover, OGD can be easily extended to keep searching continuously in case the optimal solution changes over time. This is especially relevant for long-running transfers since network and I/O congestion may change over time so does the optimal solution. Since OGD does not have memory, it can avoid being stuck in earlier solutions and respond to changing conditions by converging to the new optimal, a key requirement to ensure fairness and high performance in shared environments. Yet, we observe that OGD can still be stuck in suboptimal regions due to a lack of differentiable differences when comparing concurrency values in suboptimal regions. For example, if the optimal concurrency is 12 but OGD ended up testing concurrency values around 20 (say it is testing 19 and 20 to calculate the gradient) and then it may not be able to learn that a lower concurrency value is better because both evaluated concurrency values return similar utility value. To overcome this limitation, we extended the base OGD implementation to keep track of the optimal concurrency value with the highest utility value. By doing so, the OGD can avoid being stuck in the suboptimal region and continue the search around the optimal. For example, if we observed the maximum utility value at a concurrency value of 10 in the last 20 intervals (interval duration is equal to the duration of testing a setting, which is three seconds, by default.), but OGD is currently stuck at around concurrency 20, then it will come back to 10 as its utility value is highest among all concurrency values tested in last 20 intervals.

The conjugate gradient can be used to find a solution for multivariate problems. Hence, we implemented both conjugate gradient descent and independent univariate gradient descent algorithms. The conjugate gradient uses Equation 3 as a utility function to evaluate the fitness of concurrency combination n_r, n_n, and n_w for read, transfer, and network operations, respectively. The univariate gradient descent, on the other hand, uses a separate OGD to identify the optimal concurrency value for read, network, and write operations independently but concurrently. Figure 5 compares the convergence time for univariate gradient and conjugate gradient algorithms. We set up the experiment in a way that the optimal

Source	Destination	Storage	Bandwidth	RTT
Emulab	Emulab	RAID-0 SSD	1G	2ms
HPCLab	HPCLab	RAID-0 SSD	20G	0.1ms
HPCLab	Expanse (SDSC)	Lustre	10G	15ms
Bridges2 (PSC)	Expanse (SDSC)	Lustre	10G	58ms

Table 1: Specifications of experimental networks. Emulab is an emulated testbed, HPCLab is an isolated lab cluster, and Expanse [2] and Bridges-2 [1] are production supercomputers that are connected via high-speed research networks.

concurrency is 10 for all three operations. It takes nearly 35 sample transfers for the conjugate gradient and 15 sample transfers for the univariate gradient to find the optimal solution. Section 4.1 presents further evaluations for the performance of `Marlin` when using different online search algorithms including univariate gradient descent, conjugate gradient descent, and Bayesian optimization. In short, we find that both conjugate gradient and Bayesian optimization require extensive tuning to perform well, thus we settled on univariate gradient descent as a search algorithm.

4 EVALUATION

We assess the performance of `Marlin` in four networks as listed in Table 1 out of which Expanse [2] and Bridges-2 [1] are production HPC clusters, HPCLab is an isolated lab cluster, and Emulab is an emulated network testbed. Except for Emulab, all clusters have parallel file systems or RAID arrays as storage due to which the use of concurrency is required to maximize I/O performance. Since Emulab nodes have direct-attached single disk storage volumes, we throttle per process disk read throughput to necessitate concurrent I/O accesses to reach maximum performance, similar to parallel file systems. We use Bridges-2 and Expanse clusters for real-world wide-area experiments. Unless otherwise stated, we used datasets containing multiple 1 GiB files. The number of files is adjusted based on achievable throughput in each network. We compare `Marlin` against the state-of-the-art monolithic transfer application Falcon [9]. Similar to `Marlin`, Falcon uses the online gradient descent algorithm to search for the optimal transfer concurrency. Falcon is shown to outperform other file transfer applications (e.g., HARP [10] and Globus [8]) by up to 2×, thus we believe that comparison to Falcon would be sufficient to evaluate the efficiency of `Marlin`.

4.1 Evaluation of Optimization Algorithms

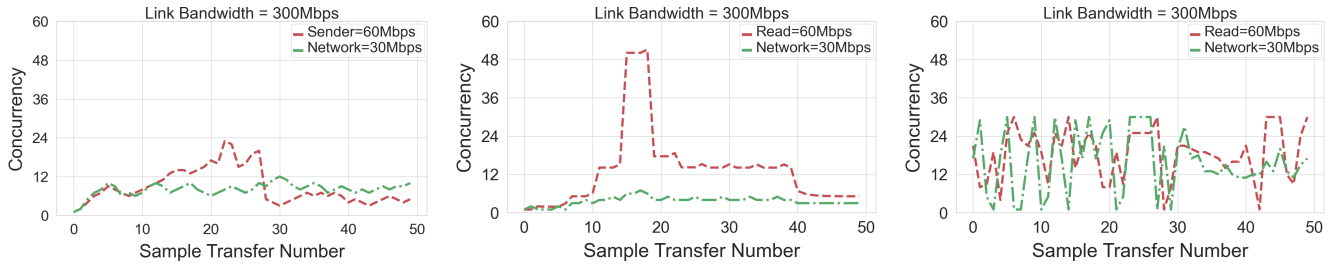
We created a testbed with 300Mbps link bandwidth, 60Mbps read I/O limitation per thread, and 30Mbps network limitations per network connection to compare the performance of different optimization algorithms. Hence, the optimal concurrency is 5 for the read operation, 10 for the network operation, and 1 for the write operation. Figure 6 presents convergence behavior when using univariate gradient descent, conjugate gradient descent, and Bayesian optimization. Gradient Descent quickly discovers that the optimal concurrency for the network is around 10. However, it keeps increasing read concurrency until it hits the memory limit because increased read concurrency increases read throughput until the staging area becomes full. Then, it reduces the number of read threads to around 5 to match the network transfer speed. Conjugate gradient descent, on the other hand, chooses a particular search direction and keeps exploring that direction until it converges. This behavior leads to undesired behavior when increasing read concurrency temporarily increases read throughput despite slower network speed. As can be seen in Figure 6(b), the conjugate gradient descent increases read concurrency to almost 50 because of misleading information collected in the first 20 intervals. It eventually lowers the number of read threads but falls short of finding the optimal concurrency for the network.

Similar to conjugate gradient, Bayesian optimization can tune all three concurrency values simultaneously using Equation 3 as a utility function. It starts with a few random concurrency combinations to begin building a Gaussian surrogate model. It updates the model after each new observation (i.e., new sample transfer with a different concurrency setting) and predicts new values to test in the next interval. The performance of the Bayesian optimization is highly dependent on the accuracy of observations. Throughput fluctuations and a temporary increase in read throughput cause it to build an incorrect model, which then affects its ability to make accurate predictions. Similar to conjugate gradient, Bayesian optimization is also unable to converge to the optimal quickly due to misleading information collected at the beginning of transfers, during which increasing read I/O concurrency leads to higher utility value. As a result, we leave the optimization of conjugate gradient and Bayesian optimization algorithms as future work and use univariate gradient descent in the rest of the experiments.

4.2 Fine-Tuning Concurrency

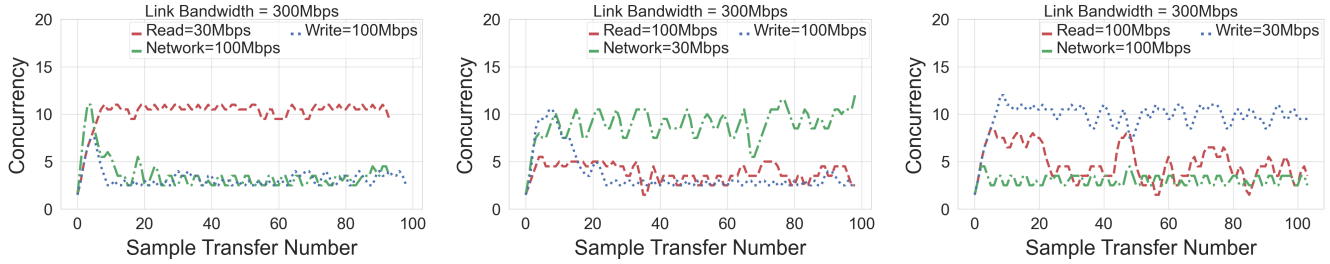
We next evaluate the performance of `Marlin` in terms of its ability to detect the bottleneck operation in a transfer and find the optimal concurrency to reach maximum utilization. Figure 7 presents the `Marlin`'s performance when concurrency is needed only for one of the read, transfer, and write operations. We used Emulab to manually restrict the throughput for I/O and network operations per thread and connection. For example, in Figure 7(a), read threads are limited to 30Mbps, network connections are limited to 100Mbps, and write threads are throttled to 100Mbps. We also limited bandwidth from the source node to the destination node to 300Mbps. Since the maximum total I/O speed is 1Gbps, the transfer task can attain 300Mbps at most (limited by network capacity) if the right concurrency values are configured. In the read bottleneck scenario (Figure 7(a)), the optimal concurrency levels are 10, 3, and 3, for read, transfer, and write operations, respectively. We observe that although `Marlin` initially increases the concurrency for all three operations, it lowers them for transfer and write operations after a few iterations while keeping it between $9-11$ for the read operation.

In the case of network bottleneck (Figure 7(b)), the optimal concurrency for read and write operations is 3, and transfer is 10. We can see that the concurrency level for both read and write operations settle at around $3-4$ while network concurrency changes around $8-11$. The main reason for the fluctuations in concurrency value is the continuous search functionality of the OGD. While it is possible to run OGD once and keep using the selected values, it is not desired in shared environments as the optimal concurrency



(a) Univariate Gradient Descent (b) Conjugate Gradient Descent (c) Bayesian Optimization

Figure 6: Comparisons of different search algorithms. While conjugate gradient descent and Bayesian optimization algorithms can tune multiple parameters at the same time, their long convergence time as well as poor prediction accuracy make them hard to adapt. Univariate gradient descent, on the other hand, works well as it can converge to optimal quickly and does not require extensive tuning.



(a) Read I/O Bottleneck (b) Network Bottleneck (c) Write I/O Bottleneck

Figure 7: `Marlin` can identify the bottleneck operation in file transfers and tune the concurrency accordingly to maximize transfer performance while keeping the overhead on other components at a minimum.

depends on the congestion level. Hence, OGD keeps searching around higher and lower values even after finding the optimal to be able to react to changing conditions quickly. Finally, `Marlin` is again performs well in the write bottleneck scenario(Figure 7(c)) by increasing write concurrency to around 10 while keeping the network and transfer concurrency at around $2-4$.

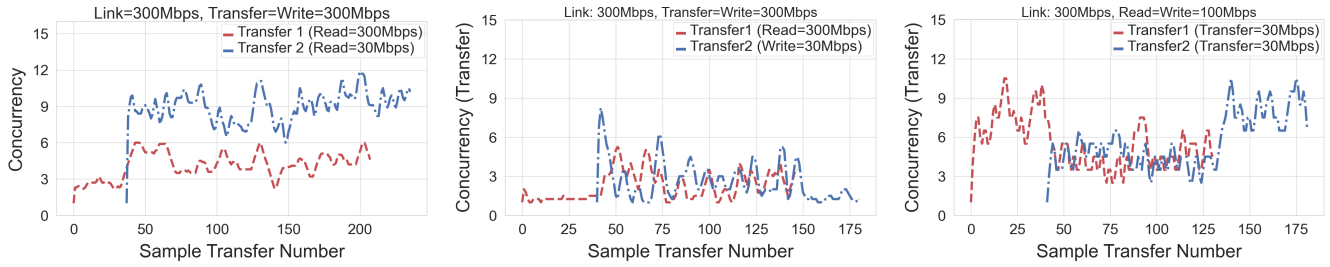
4.3 System Overhead

We next evaluate the CPU usage and I/O contention created by `Falcon` and `Marlin`. We use HPCLab and limit per thread read I/O to 1Gbps to simulate a slow sender scenario. As the HPCLab network has 20Gbps bandwidth, `Falcon` uses around 20 concurrency for all three operations. On the other hand `Marlin` achieves a similar throughput using 20 read threads, $4-5$ network threads, and $5-6$ write threads. While transfers are running, we also executed several `dd` processes on the receiver node, each writing 100GiB data to the file system to evaluate the fairness of transfer applications to non-transfer processes.

Table2 shows CPU usage, the execution time of write (i.e., `dd`) processes, and the throughput of transfer operations. When there is only one competing write process, `Falcon` consumes $45\%-50\%$ CPU utilization while `Marlin` consumes nearly 10% less. As the number of write processes increases, their runtime increases as well; however, the rate of increase is nearly double when they are competing against `Falcon` compared to `Marlin`. This is mainly because `Falcon` does not only create a high number of write threads when there is no competition but also increases the concurrency level even further in the presence of competing I/O processes to increase its throughput. On the other hand, `Marlin` allows competing non-transfer processes to attain their fair share by creating a minimum number of I/O threads. Therefore, using the same level

# of write operations	CPU (F)	CPU (M)	Write time with F	Write time with M	Thr (F)	Thr (M)
1	46.1%	37.8%	158s	131s	19.2G	19.1G
2	54.7%	44.1%	209s	149s	19.1G	17.9G
3	62.6%	54.3%	266s	197s	19.1G	16.1G
4	66.9%	58.7%	373s	245s	19.1G	15.3G
5	70.4%	64.8%	533s	271s	19.1G	14.8G

Table 2: System overhead and fairness analysis for `Marlin` (M) and `Falcon` (F). We execute up to five concurrent write operations while running transfers to see the impact of transfers on competing I/O processes. `Marlin` results in around 10% less CPU usage in addition to being fairer to competing write operations.



(a) `Falcon` (Different Read I/O Performance) (b) `Marlin` (Different Read I/O Performance) (c) `Marlin` (Similar Read I/O Performance)

Figure 8: Fairness analysis for Falcon and `Marlin`. Falcon causes unfair network bandwidth allocation between competing transfers due to using the same concurrency for network and I/O operations. On the other hand, `Marlin` is able to identify I/O bottlenecks and tune the concurrency for transfer and I/O operations separately, which helps it to ensure fairness between transfers regardless of I/O characteristics.

of concurrency for all operations has an adverse impact on other processes running alongside the transfers.

4.4 Fairness Between Competing Transfers

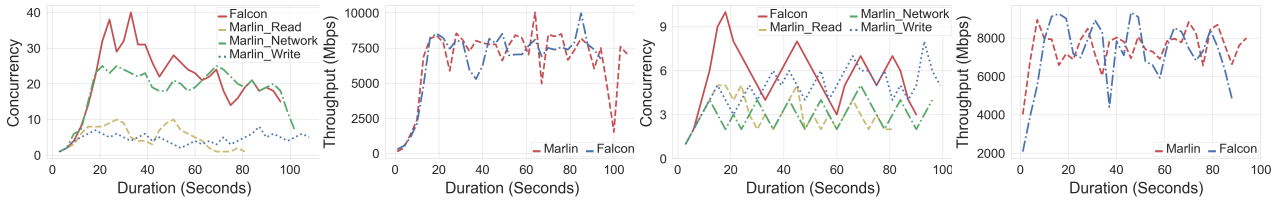
We next compare the performance of independent transfers competing for the same bottleneck link in Emulab as shown in Figure 8. We run two transfers between two different source-destination pairs with different read concurrency requirements. Specifically, read I/O throughput per thread is limited to 300Mbps for the first (Transfer-1), whereas it is limited to 30Mbps for the second transfer (Transfer-2). The transfers share a network link whose capacity is limited to 300Mbps. Thus, single write and transfer threads are sufficient for both transfers to attain the maximum possible throughput (i.e., 300Mbps) in this network as no limitations are injected for write and transfer operations. On the other hand, Transfer-2 requires 10 concurrent read threads to attain 300Mbps read I/O throughput.

Figure 8(a) presents the results for two competing `Falcon` transfers. When Transfer-1 starts, it settles on a concurrency value between 1−3 as small concurrency is sufficient for it to reach 300Mbps throughput. When Transfer 2 joins, it tests higher concurrency values than 3 and observes a considerable increase in the utility function due to an increase in read I/O throughput. As the same concurrency level is used for read, transfer, and write operations in Falcon, this in turn causes Transfer-2 to attain a higher share in the network. That is, the capacity of the bottleneck link is shared between Transfer 1 and Transfer 2 in proportion to the number of concurrent network connections they create. Since Transfer-2 chooses a higher concurrency value than Transfer-1, it initially obtains almost three times higher network throughput than Transfer-1. Although Transfer-1 responds to this by increasing its concurrency value to around 5, it does not observe a sufficient increase in utility to increase it even further. As a result, bottleneck link capacity is shared in a 2-1 ratio between Transfer-2 and Transfer-1.

Figure 8(b) shows results for `Marlin` for the same configuration as Falcon transfers are executed. Unlike Falcon, both transfers choose to create 1 − 3 network connections (on average 2.25 for Transfer 1 and 2.53 for Transfer-2) and share the network bandwidth almost equally (47% to 53%) despite having different read I/O concurrency requirement. Figure 8(c) shows two competing `Marlin` transfers with network limitations of 30Mbps, thus 10 concurrent transfer threads are needed to fully utilize the network capacity of 300Mbps. Transfer-1 converges to concurrency level 10 for the transfer operation when it is the only transfer in the network. When Transfer-2 joins, Transfer-1 lowers its network concurrency as it realizes that concurrency value 5 − 6 is sufficient to attain its fair share in the network (i.e., around 150Mbps) while minimizing the packet loss rate. Transfer-2 also converges to a concurrency level of 5 − 6 and obtains its fair share. When Transfer-1 completes, Transfer-2 is able to claim the free network bandwidth by increasing its network concurrency with the help of the continuous search functionality of the OGD.

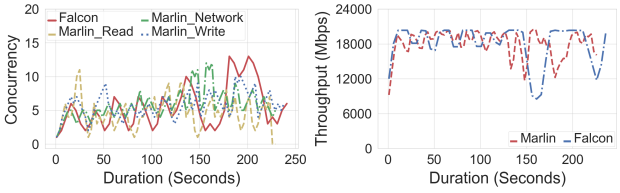
4.5 Evaluations in Production Systems

Figure 9 compares the performance of Falcon and `Marlin` in production high-performance networks. While both Expanse and Bridges-2 supercomputers are equipped with high-performance Lustre file systems, have high-speed connectivity to research networks, and utilize dedicated data transfer nodes, transfers need parallelism to unleash the available capacity. In particular, Bandwidth Delay Product (BDP) for Bridges-2 to Expanse communication is 69 MiB (10Gbps×58ms); thus, TCP requires nearly 70MiB buffer space to reach 10Gbps throughput using a single connection. However, the maximum TCP buffer size is limited to 5.8 MiB in Bridges-2 nodes, which is twelve times smaller than the requirement. Since end users cannot change the TCP buffer size, the use of multiple concurrency network connections is the only way to mitigate TCP buffer size limitation as each connection can attain a separate TCP buffer space equal to a maximum value (i.e., 5.8MiB). Figure 9(a) and 9(b) show the concurrency and throughput values using Falcon and `Marlin`. As TCP buffer size is the main limitation for the performance, both Falcon and `Marlin` chooses a high concurrency value (around 20) for the network. On the other hand, `Marlin` realizes 3 − 5 read and write threads are sufficient to read and write files at maximum performance. As a result, while achieving similar throughput to Falcon,



(a) Bridges2-Expanse (Concurrency) (b) Bridges2-Expanse (Throughput) (c) HPCLab-Expanse (Concurrency) (d) HPCLab-Expanse (Throughput)

Figure 9: Performance comparison for `Falcon` and `Marlin` in real-world networks. `Marlin` attains competitive results in transfer throughput while lowering the concurrency value significantly.



(a) HPCLab (Concurrency) (b) HPCLab (Throughput)

Figure 10: Performance comparison for `Falcon` and `Marlin` in HPCLab network with 20Gbps bandwidth.

`Marlin` is able to lower the number of I/O processes significantly, thereby lowering CPU utilization and I/O contention as discussed in Section 4.3.

TCP buffer size limitation affects HPCLab-Expanse transfers (Figure 9(c) and 9(d)) since Expanse nodes are configured with 8MiB maximum TCP buffer size while BDP is around 17MiB ((10Gbps×15ms). In addition to TCP buffer size limitation, both read and write I/O operations require parallelism to overcome the I/O limitations. Specifically, a concurrency value of 6 is needed to reach close to 8Gbps write I/O throughput on the receiver end, Expanse. While the read operation also needs parallelism, it can reach 8Gbps throughput with a slightly smaller concurrency value. Similar to Bridges2-Expanse transfers, `Marlin` attains comparable throughput to Falcon despite using a lower read and transfer threads. Finally, Figure 10 presents `Marlin`'s performance in a local-area network (i.e., HPCLab) with 20Gbps bandwidth. The optimal concurrency level for read, transfer, and write operations is almost the same, around 5. Hence, Falcon and `Marlin` perform similarly in terms of concurrency values and throughput.

4.6 Performance Enhancements for Short Transfers

A previous analysis of the data transfers in the research networks showed that the median dataset size is 10GiB and more than 80% all transfers move less than 128GiB data [21]. Thus, it is essential to improve the performance of small transfers that last a few seconds to minutes. In most cases, I/O performance is the bottleneck for small transfers, hence, optimizing I/O performance is critical to enhancing the throughput of short transfers. One possible solution is placing high-performance storage caches (e.g., NVMe SSD, nonvolatile memory) on data transfer nodes such that files can be staged at a faster speed than directly reading/writing from parallel file systems (as illustrated in Figure 4) similar to burst buffers

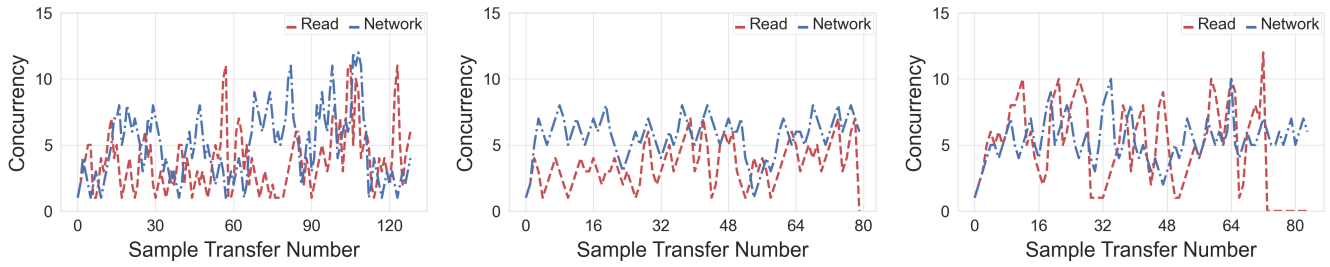
Data Size	`Falcon` (Sec)	`Marlin` (Sec)	Improvements (%)
HPCLab1 to HPCLab2 (Memory Buffer)			
10 GB	9.8	7.4	24.5
25 GB	21.7	14.1	35.1
50 GB	43.8	24.3	44.5
75 GB	65.6	34.2	47.8
100 GB	86.9	42.1	51.6
HPCLab to Campus Cluster (NVMe SSD Buffer)			
500 GB	465.4	202.7	56.5
2000 GB	1839.7	833.6	54.7
5000 GB	4494.4	2116.3	52.9

Table 3: Performance comparisons of Falcon and `Marlin` for the transfer of small datasets when write I/O is the bottleneck of the transfers. Since `Marlin` is able to cache files in the staging area (RAM or NVMe SSD), it can attain higher read and network throughput and move the entire dataset to the destination node quicker.

in HPC clusters. Please note that small transfers are more likely to observe a significant gain through this approach because large transfers are likely to hit the capacity limit of the staging area and lower their speed to the speed of the file system.

`Marlin` lends itself to this idea as its modular architecture allows it to stage-in files to temporary space before transferring to the network and writing to file systems. Hence, we demonstrate the benefit of using a staging area for the transfer of small datasets in Table 3. The staging area can utilize main memory or NVMe SSDs (or both) to cache the data transferred by the network operations based on the availability of the hardware.

We first evaluate the impact of using main memory as a staging area for small datasets when the transfers are write-limited. To simulate a scenario in which the write speed of staging space for `Marlin` is significantly (more than 2×) higher than the write speed of a file system, we used HPCLab servers (HPCLab1-HPCLab2 in Table 3) and limited the write speed to 10 Gbps while the read I/O speed is 30 Gbps, network bandwidth is 20Gbps. Thus, it is possible to transfer files to the staging area at around 20 Gbps speed. On the other hand, only 10Gbps throughput can be attained if a monolithic transfer application is used which will write data directly to the file system. Clearly, `Marlin` can move the files to the staging area of the destination node faster by more than $2x$, reducing the transfer time by up to 51.6% compared to `Falcon`.



(a) 3GB Buffer (b) 10GB Buffer (c) 100GB Buffer

Figure 11: Impact of memory space on the performance of `Marlin`. 10GiB is sufficient for `Marlin` to perform normally and attain high performance in a 20 Gbps network.

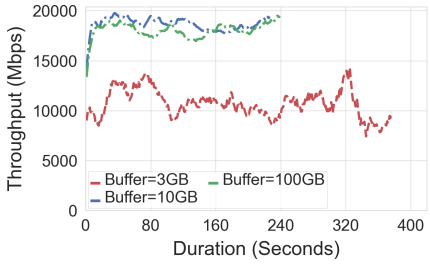


Figure 12: Throughput of `Marlin` with different memory limits. It requires at least 10GiB memory space to achieve 20 Gbps throughput in HPCLab network.

We next present a performance comparison when using NVMe SSDs as a staging area. To do so, we conduct transfers between HPCLab nodes where the receiver node saves the files to a parallel file system located at a nearby campus cluster. The write limit to the campus cluster is limited to 10Gbps whereas the transfers between HPCLab nodes can reach 20 Gbps. Since NVMe SSD-based staging area has higher capacity than the main memory-based staging area, larger datasets can be cached in the staging before hitting the limits. Hence, we used 500 GiB to 5 TiB transfer sizes and attained up to 56% reduction in transfer times by means of mitigating write limitations. Consequently, `Marlin` is able to reduce the duration for small dataset transfers significantly by means of mitigating write I/O limitations.

4.7 Impact of Memory Limit

`Marlin` uses main memory (`tmpfs`) on sender and receiver nodes as a staging area between network and I/O operations. Hence, it is important to limit memory usage to avoid saturating the whole memory space for a single transfer application. We, therefore, define a hard limit for memory usage on both the sender and receiver sides. By default, we set the limit to be 30% of free memory space. Figure 11 evaluates the impact of memory limit for HPCLab transfers. We varied the buffer limit on the source node between 3GiB and 100GiB. Since the speed of transfers is around 20Gbps, 3GiB allows `Marlin` to store around 0.6 seconds worth of data on memory before hitting the limit. This value becomes 4 seconds when using the buffer size limit of 10GiB and 40 seconds when setting the memory limit to 100GiB. The figures show that using a very small memory limit causes `Marlin` to experience significant fluctuations in the concurrency value of read and transfer operations. The number of read threads often hit 1 since it cannot create and test multiple read threads accurately due to lack of memory space. As a result, the transfers take 60% longer compared to using 10GiB or 100GiB memory space as shown in Figure 12. Therefore, `Marlin` necessitates a memory space that is at least as big as to hold a couple of seconds worth of data. We believe this is a reasonable expectation as the memory capacity of data transfer nodes in production systems is high enough to accommodate this. As an example, the Expanse transfer node has a 64GiB memory and the Bridges-2 transfer node has a 128GiB memory.

5 CONCLUSION

Similar to compute jobs, data transfers in wide-area high-performance networks require parallelism to overcome I/O and network limitations. However, the implementation of transfer parallelism (aka concurrency) in existing transfer applications has two main issues. First, it creates the same level of parallelism for read, transfer, and write operations when transferring files. This, in turn, overburdens system resources (e.g., CPU) since not all operations require the same level of parallelism to achieve a similar throughput. Second, it causes unfair resource allocation when multiple transfers with different I/O characteristics share the bottleneck network link. Instead of trying to overcome these problems by manipulating existing monolithic transfer applications, we propose a modular file transfer application, `Marlin`, to separate read, transfer, and write operations. `Marlin` combines game theory-inspired utility functions with a univariate online gradient descent algorithm to swiftly discover the fair and optimal parallelism levels for each operation.

We evaluated the performance of `Marlin` both in emulated and real-world networks to show that it is able to identify the minimum concurrency level for read, transfer, and write operations to maximize transfer throughput while minimizing system overhead and ensuring fairness among competing transfers. We also show that the modular architecture of `Marlin` lends itself to the implementation of burst buffer design in data transfer nodes to expedite the transfer of small datasets by caching data on high-performance storage spaces (e.g., NVMe SSD and PMEM). Specifically, we find that `Marlin` can speed up the transfer performance of short transfers by more than 2×.

ACKNOWLEDGEMENT

The work in this study was supported in part by the NSF grants 2145742, 2007789, and 2209955.

REFERENCES

[1] 2023. Bridges-2. https://www.psc.edu/resources/bridges-2/.
[2] 2023. Expanse. https://www.sdsc.edu/services/hpc/expanse/.
[3] 2023. Fast Data Transfer. http://monalisa.cern.ch/FDT/.
[4] 2023. Globus. https://www.globus.org.
[5] 2023. Lighting up the LSST Fiber Optic Network: From Summit to Base to Archive. lsst.org/news/lighting-lsst-fiber-optic-network-summit-base-archive.
[6] 2023. The network challenge. https://home.cern/science/computing/network.
[7] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. 2005. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 54.
[8] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke. 2012. Software as a Service for Data Scientists. *Commun. ACM* 55:2 (2012), 81–88.
[9] Md Arifuzzaman and Engin Arslan. 2021. Online Optimization of File Transfers in High-Speed Networks. In *High Performance Computing, Networking, Storage and Analysis, SC21: International Conference for*. IEEE.
[10] Engin Arslan, Kemal Guner, and Tevfik Kosar. 2016. HARP: predictive transfer optimization based on historical analysis and real-time probing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 288–299.
[11] Engin Arslan and Tevfik Kosar. 2018. High-Speed Transfer Optimization Based on Historical Analysis and Real-Time Tuning. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (2018), 1303–1316.
[12] Engin Arslan, Bahadir A Pehlivan, and Tevfik Kosar. 2018. Big data transfer optimization through adaptive parameter tuning. *J. Parallel and Distrib. Comput.* 120 (2018), 89–100.
[13] Engin Arslan, Brandon Ross, and Tevfik Kosar. 2013. Dynamic protocol tuning algorithms for high performance data transfers. In *European Conference on Parallel Processing*. Springer, 725–736.
[14] P. Balaprakash, V. Morozov, R. Kettimuthu, K. Kumaran, and I. Foster. 2016. Improving Data Transfer Throughput with Direct Search Optimization. In *2016 45th International Conference on Parallel Processing (ICPP)*. 248–257. https://doi.org/10.1109/ICPP.2016.36
[15] John Bresnahan, Michael Link, Rajkumar Kettimuthu, Dan Fraser, Ian Foster, et al. 2007. Gridftp pipelining. In *Proceedings of the 2007 TeraGrid Conference*.
[16] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (2016), 50.
[17] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. {PCC} Vivace: Online-Learning Congestion Control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 343–356.
[18] T. J. Hacker, B. D. Noble, and B. D. Atley. 2005. Adaptive Data Block Scheduling for Parallel Streams. In *Proceedings of HPDC '05*. ACM/IEEE, 265–275.
[19] Elad Hazan. 2016. Introduction to online convex optimization. *Foundations and Trends® in Optimization* 2, 3-4 (2016), 157–325.
[20] Yuanlai Liu, Zhengchun Liu, Rajkumar Kettimuthu, Nageswara Rao, Zizhong Chen, and Ian Foster. 2019. Data transfer between scientific facilities–bottleneck analysis, insights and optimizations. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 122–131.
[21] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Nageswara SV Rao. 2018. Cross-geography scientific data transferring trends and behavior. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 267–278.
[22] MD SQ Zulkar Nine and Tevfik Kosar. 2020. A Two-Phase Dynamic Throughput Optimization Model for Big Data Transfers. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 269–280.
[23] Pratiksha Thaker, Matei Zaharia, and Tatsunori Hashimoto. [n.d.]. Learning and utility in multi-agent congestion control. *optimization* 24, 10 ([n. d.]), 11–18.
[24] Esma Yildirim, Engin Arslan, Jangyoung Kim, and Tevfik Kosar. 2016. Application-level optimization of big data transfers through pipelining, parallelism and concurrency. *IEEE Transactions on Cloud Computing* 4, 1 (2016), 63–75.
[25] Daqing Yun, Chase Q Wu, Nageswara SV Rao, Qiang Liu, Rajkumar Kettimuthu, and Eun-Sung Jung. 2017. Data Transfer Advisor with Transport Profiling Optimization. In *Local Computer Networks (LCN), 2017 IEEE 42nd Conference on*. IEEE, 269–277.
[26] Liang Zhang, Phil Demar, Bockjoo Kim, and Wenji Wu. 2017. MDTM: Optimizing data transfer using multicore-aware I/O scheduling. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*. IEEE, 104–111.
[27] Martin Zinkevich. 2003. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 928–936.