Improving Data Transfer Throughput with Direct Search Optimization

Prasanna Balaprakash*†, Vitali Morozov†, Rajkumar Kettimuthu*, Kalyan Kumaran†, and Ian Foster*
* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
† Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA
{pbalapra, morozov, kettimut, kumaran, foster}@anl.gov

***Abstract*—Improving data transfer throughput over high-speed long-distance networks has become increasingly difficult. Numerous factors such as nondeterministic congestion, dynamics of the transfer protocol, and multiuser and multitask source and destination endpoints, as well as interactions among these factors, contribute to this difficulty. A promising approach to improving throughput consists in using parallel streams at the application layer. We formulate and solve the problem of choosing the number of such streams from a mathematical optimization perspective. We propose the use of direct search methods, a class of easy-to-implement and light-weight mathematical optimization algorithms, to improve the performance of data transfers by dynamically adapting the number of parallel streams in a manner that does not require domain expertise, instrumentation, analytical models, or historic data. We apply our method to transfers performed with the GridFTP protocol, and illustrate the effectiveness of the proposed algorithm when used within Globus, a state-of-the-art data transfer tool, on production WAN links and servers. We show that when compared to user default settings our direct search methods can achieve up to 10x performance improvement under certain conditions. We also show that our method can overcome performance degradation due to external compute and network load on source end points, a common scenario at high performance computing facilities.**

I. Introduction

Advances in supercomputers, scientific instruments, and sensor technologies require high-speed file transfer, for example to distribute data to remote users and/or computing facilities. Yet achieved data transfer performance is often only a small fraction of the capacity of the high-speed networks that have been increasingly widely deployed to support data-driven science. Thus improving data transfer performance has become a vitally important problem.

Many approaches have been proposed to improve the throughput of data transfers. We can classify them into two broad groups: low-level protocol and high-level application tuning. Protocol-level work has produced new protocols such as Hamilton TCP [18], CUBIC TCP [8], Scalable TCP [14], and UDT [7] and methods for tuning their parameters. Application-level tuning consists in optimizing high-level data transfer parameters such as scheduling, socket buffer size, pipelining, parallelism, and concurrency to maximize bandwidth utilization [11, 12, 16, 27, 28]. We focus here on improving the throughput of TCP-based transfers at the application level by tuning the number of parallel streams. Although this problem has received much attention from the research community [25], previous approaches have used only ad hoc heuristics. We formulate the task of tuning the number of parallel streams as a mathematical optimization problem, and show this optimization problem can be solved effectively with direct search methods.

The context for our work is thus as follows. An application splits large data into chunks that it sends simultaneously via multiple TCP streams. Multiple streams can make better use of parallel file systems, scale more rapidly to peak bandwidth, and respond less aggressively to packet losses. However, multiple streams also introduce overheads and can compete with other flows. Thus, finding the optimal number of parallel streams for a given transfer is a difficult task, depending not only on network characteristics but also on other external conditions at the endpoints. Moreover, both network characteristics and external conditions can change over time because of other activities on shared networks and endpoints [25]. Existing approaches for tuning parallel streams can be classified into three broad classes: analytical, empirical, and dynamic tuning approaches.

In **analytical approaches**, first-principles analytical models are developed to capture the relationship between network characteristics and throughput. Hacker et al. [9] developed analytical model for throughput as a function of round trip time (RTT), packet loss rate, maximum segment size, and number of parallel streams. Lu et al. [19] extended Hacker's model for congestion networks by computing the relationship between packet loss rate, RTT, and number of parallel streams. Altman et al. [3] developed analytical model that establishes the relation between bottleneck link capacity and the number of streams required for saturating it. Analytical models have enjoyed significant success in the network protocol community. However, they often fail to capture all of the complex interactions between input parameters and dynamic external load in the network infrastructure. Moreover, developing a complex analytical model is time consuming and requires expertise in several fields such as memory and compute subsystems, protocols, and parallel file systems.

When analytical models become too restrictive, **empirical approaches** are an effective alternative. For a given pair of endpoints, experimental and/or historical transfer information is used to build a predictive model using model-fitting approaches; the resulting model is used to find the optimal number of parallel streams. Yildirim et al. [27] developed a curve-fitting approach using Newton method to find the

optimal number of streams. Yin et al. [28] developed analytical models that require calibration data from Iperf and GridFTP to identify the optimal number of parallel streams. Kettimuthu et al. [16] use historical information and adjust parallel streams in response to external load. The key drawbacks of these methods are that they lack generality—a model obtained for one pair of endpoints cannot be generalized to others. Moreover, collected data may become obsolete when source, destination, or network is modified. Furthermore, the data may not cover all possible external traffic and load conditions.

Dynamic approaches are model free: they transfer data chunks with varying numbers of streams, measure the aggregate throughput as a function of stream count, and modify the number of parallel streams with respect to throughput gain according to some fixed scheme. Ito et al. [11, 12] proposed additive increase, multiplicative decrease, and multiplicative increase adaptation schemes. Their approach requires determining the values of round-trip time and TCP buffer sizes obtained from profiling tools. Balman et al. [5] proposed a simple adaptive scheme that compares the two consecutive throughputs (current and previous) and additively increases the number of parallel streams by a constant factor. Yildirim et al. [25] analyzed the effects of parallelism, concurrency, and pipeline parameters on throughput of large dataset transfers with heterogenous file sizes. They used rules derived from the observed relationships to develop a heuristic that exponentially increases parallelism and concurrency values until the maximum achievable throughput is reached.

Previously proposed approaches for dynamic tuning involve ad hoc heuristics or the encoding of substantial expert knowledge. The approach that we present here uses systematic mathematical optimization to tune parallel streams at the application level, an approach that has received little attention. As with the dynamic tuning approaches referenced above [5, 25], our method does not require any external measurements or profiling results, relying instead on the measured throughput of each transferred data chunk. The principal innovation in our approach is the use of direct search methods, instead of heuristic and ad hoc schemes, to modify the number of streams. The direct search methods systematically explore the search space defined by the tunable parameters.

The contributions of the paper are as follows.

- For the first time, we formalize and solve the problem of optimizing data transfer throughput from a systematic and general mathematical optimization perspective.
- We present an experimental study of direct search methods, a class of mathematical optimization approaches, for dynamically tuning the number of parallel TCP streams during the data transfer.
- We demonstrate that the proposed direct search methods can result in data transfer throughput improvement of up to 10x and they can reach the performance of the expert-knowledge-based-heuristic.
- We show that compute load and network traffic at the source endpoint can significantly affect the throughput and dynamic tuning can be used to improve the throughput. This issue has received little attention from the research community.

II. Problem

Given source src, destination dst, and data of size s to be transferred from src to dst, the problem of throughput optimization can be formulated as maximizing the integral of throughput over time:

$$\underset{x\in\mathcal{D}}{\operatorname{argmax}} \int_{T_{start}}^{T_{end}^{f}} f_t(x, s_t, \delta_t, \theta_t^{src}, \theta_t^{dst})dt, \tag{1}$$

where $x \in \mathcal{D} \subset \mathbb{R}^m$ is a vector of m tuning parameters that control the transfer; $\mathcal{D}$ is a domain of possible values for the tuning parameters; δ_t is a hyperparameter capturing the network condition at time t; and θ_t^{src} and θ_t^{dst} are hyperparameters describing external loads on the source and destination at time t, respectively. The non-negative function f_t is the observed throughput from time t' to t'' ($t'' - t' = dt$) for transferring data of size s_t. The integral limits T_{start} and T_{end}^{f} denote the transfer start and end time, respectively, where the T_{end}^{f} depends on the throughput f_t. The network condition hyperparameter δ_t can include the transfer protocol parameters. External load hyperparameters θ_t^{src} and θ_t^{dst} can include the state of the various components involved in the transfer, such as CPU, memory, and (in the case of disk-to-disk transfers) file system.

In this paper, we focus on transferring data from src memory to dst memory with the TCP protocol. Parallel TCP streams are often required to achieve transfer rates close to network speeds [13, 26]. But blindly using a large number of TCP streams can be counterproductive. On a loss-free dedicated network connection with a smaller RTT ($<$ 20ms), even a single TCP stream can saturate the network; and in that case multiple TCP streams decrease the throughput [22]. For dedicated connections with larger RTT and for shared connections, parallel TCP streams help improve transfer rates but only up to a certain point [15]. The point of negative return varies based on external conditions such as other traffic originating from or destined to the same source and/or destination endpoint, other traffic in the network, and other compute load exerted on the source and/or destination endpoints.

From a methodological viewpoint, solving Eq. (1) is a difficult task because of two primary dynamic factors. First, network conditions (δ_t) such as available bandwidth, RTT, packet loss rate, TCP congestion control dynamics, and bottleneck link capacity can change over the transfer duration. Second, the external loads (θ_t^{src}, θ_t^{dst}) on the source and destination can start and end at any time, further changing the network conditions and the corresponding hyperparameters. Consequently, the best parameter configuration x^* for one external condition might not be the best for a different condition.

III. Proposed approach

We solve the problem of finding the appropriate number of parallel TCP streams using direct search. First, we present a

simple case study that illustrates the effect of external conditions on the number of parallel streams. We then introduce the proposed direct search methods.

A. Impact of external conditions

We begin our discussion with empirical observations from a controlled experiment. We use a Nehalem machine (a dual socket quad-core Intel Xeon CPU E5530 running at 2.40 GHz with 48 GB main memory) at Argonne's Joint Laboratory for System Evaluation as the source and a Sandybridge machine (a dual socket eight-core Intel Xeon CPU E5-2670 running at 2.60 GHz with 32 GB main memory) at the University of Chicago as the destination. Both the Nehalem and the Sandybridge machines are connected to the network through 40 Gb/s NIC. Consequently, the theoretical peak throughput is 40 Gb/s (5 GB/s).

We use the Globus Toolkit's command line utility `globus-url-copy` to transfer data from `/dev/zero` to `/dev/null` under various controlled external conditions and for varying numbers of TCP streams. For external network traffic, we run a second transfer, with `ext.tfr` streams, from source to destination. For external CPU load, we run `ext.cmp` copies of the Intel Math Kernel Library's `dgemm` routine on the source machine. We configure each copy of this routine, which calculates the product of two double precision matrices, to consume all available CPU on all available cores. In the rest of the paper, we use the term *external load* to refer to this combination of external traffic originating from the source and computation running on the source endpoint.

The number of TCP streams used by Globus GridFTP [1] is the product of two user-defined parameters, concurrency and parallelism. While the former exploit multiple CPU cores, the latter does not. For example, when concurrency and parallelism are set to 2 and 4, respectively, 2 CPU cores each with 4 streams run a total of 8 parallel streams. In this study, we fix parallelism at 1 and vary concurrency.

Figure 1 shows the observed throughput with and without external load. We note that: 1) In both graphs, observed throughput increases monotonically with the number of parallel streams up to a critical point, after which it decreases monotonically. For our purposes here, *critical* refers to the number that yields the highest throughput under a given load. 2) The critical point increases with external load. 3) External load decreases observed peak throughputs.

We attribute the first observation, the monotonic increase in the observed throughput with the number of parallel streams, to the additive increase and multiplicative decrease (AIMD) strategy adopted for determining the window size—the number of packets that can be sent without acknowledgment—in the TCP protocol during the steady state (or congestion avoidance phase). The window size is increased gradually with an additive factor; as soon as a packet loss is detected, it is decreased drastically with a multiplicative factor. This slow increase and fast decrease tends to result in unused bandwidth. The newer TCP congestion control modules such as Hamilton TCP [18], CUBIC TCP [8] (default in Linux), and Scalable

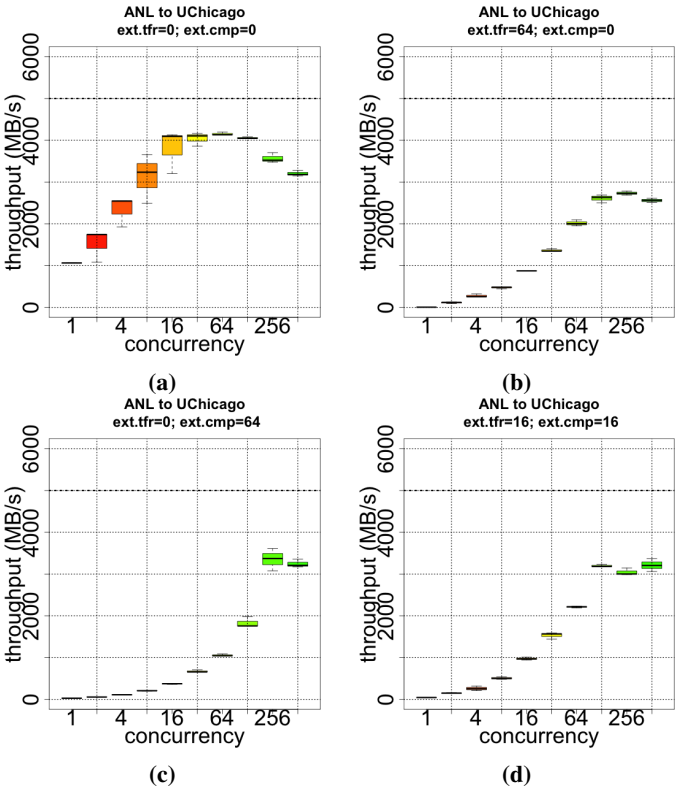


Figure 1: Boxplot statistics showing the impact of parallel TCP streams (concurrency) on throughput from ANL to UChicago in two scenarios: (a) no external load and (b) high external load, i.e., both `ext.tfr` and `ext.cmp` set to 16. We repeat the transfer for each concurrency value 5 times, and run each transfer for 10 mins.

TCP [14] use a smaller multiplicative factor and/or a more aggressive additive factor but still result in unused bandwidth (the endpoints used for experiments shown in Figure 1 use Hamilton TCP.) The adoption of multiple streams consumes otherwise wasted bandwidth and thus increases the achievable throughput. After the critical point, however, the benefit of multiple streams is dominated by processing overhead due to context switching and related book-keeping required for running many streams. These costs eventually reduce the overall observed throughput.

With reference to the second observation, in the absence of any external transfer, 64 streams are required to achieve the peak, but when the external traffic rises to 64 streams, the critical point increases to 256. As noted in previous work [6, 21, 4, 10], the creation of many streams allows our flow to claim the majority of available bandwidth. On the other hand, increasing `ext.cmp` decreases the CPU time available for each active data transfer stream. Increasing the number of streams—up to the critical point—increases the overall share of CPU time spent on the active data transfer stream and consequently the aggregate throughput.

In summary, these results show that the critical number of parallel streams for a particular transfer depends on the external load. We conclude that one should be able to improve

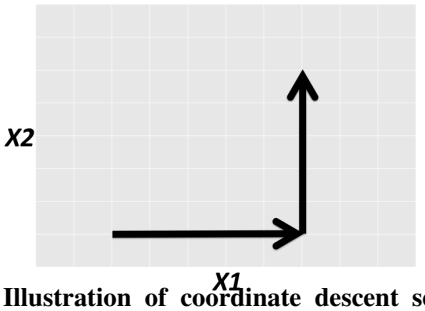


Figure 2: Illustration of coordinate descent search with two parameters $X1$ and $X2$. The method first finds the best value for $X1$ by keeping $X2$ at constant value and then finds the best value for $X2$ with the best-found $X1$.

Algorithm 1 Coordinate descent tuner (`cd-tuner`)

Input: starting point x_0, control epoch time e, data size s, tolerance $\epsilon\%$

1 `global` $s' \leftarrow s$
2 **procedure** RUNTRANSFER(x')
3 $f_{x'}, s'' \leftarrow$ `transfer`(x', src, dst, s')
4 $s' \leftarrow s' - s''$
5 **Output:** $f_{x'}$
6 **end procedure**
7 $c \leftarrow 0$
8 $f_{x_0} \leftarrow$ `runTransfer`(x_0)
9 $c \leftarrow c + 1$
10 $f_{x_1} \leftarrow$ `runTransfer`(x_1)
11 $c \leftarrow c + 1$
12 **while** $s' > 0$ **do**
13 $\Delta_c \leftarrow 100 \times \frac{f_{x_{c-1}} - f_{x_{c-2}}}{f_{x_{c-2}}}$
14 **if** $x_{c-1} \neq x_{c-2}$ **then**
15 $\delta_c \leftarrow \frac{\Delta_c}{x_{c-1} - x_{c-2}}$
16

$$x_c \leftarrow \begin{cases} x_{c-1} + 1, & \text{if } x_{c-1} = x_{c-2} \text{ and } |\Delta_c| > \epsilon \\ x_{c-1} + 1, & \text{if } x_{c-1} \neq x_{c-2} \text{ and } \delta_c > \epsilon \\ x_{c-1} - 1, & \text{if } x_{c-1} \neq x_{c-2} \text{ and } \delta_c < -\epsilon \\ x_{c-1}, & \text{otherwise} \end{cases}$$

17 $f_{x_c} \leftarrow$ `runTransfer`(x_c)
18 $c \leftarrow c + 1$

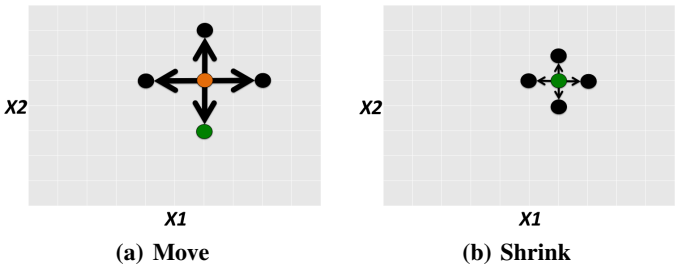
the performance of a given transfer by implementing an adaptive approach in which changes to the external state are observed by monitoring transfer throughputs periodically and then an adaptive method is used to tune the number of parallel streams for the given transfer. We implement this adaptive capability using direct search methods.

B. Direct search

To solve the optimization problem, we focus on direct search methods [17], a class of mathematical optimization techniques. We investigate three such methods: coordinate descent [24], compass [23], and Nelder-Mead [20].

Coordinate descent search: The main idea behind coordinate descent search consists in searching the best value for one parameter at a time (see Figure 2 for an illustration). To find the best value for each parameter, we modify the coordinate descent search to increase the number of streams whenever additional bandwidth is available and to decrease it as soon as the src has become the bottleneck.

Algorithm 1 shows the pseudo code of the customized



(a) Move **(b) Shrink**

Figure 3: Illustration of compass search with two parameters $X1$ and $X2$. The method tries to find the best values for $X1$ and $X2$ in the coordinate directions of the incumbent point, either by moving to the improving point (Fig. 3a) or by examining points closer to the incumbent point (Fig. 3b).

coordinate descent search (`cd-tuner`). At each control epoch c of `cd-tuner`, the data is transferred for e seconds with x_c parallel streams. The function `runTransfer`(x') starts data transfer from src to dst for e seconds with x' parallel streams (lines 2–6). It results in s'' data transferred with throughput of $f_{x'}$; the function also updates the global variable s', which keeps track of remaining amount of data that needs to be transferred. The number x_c streams for the control epoch c is determined by observed throughputs f_{c-1} and f_{c-2} with x_{c-1} and x_{c-2} parallel streams at previous control epochs $c-1$ and $c-2$, respectively. The number of streams is increased from x_{c-1} when there is a new congestion or additional bandwidth. This condition is detected when x_{c-1} and x_{c-2} are the same but $f_{x_{c-1}}$ and $f_{x_{c-2}}$ are significantly different, or when $x_{c-1} > x_{c-2}$ and $f_{x_{c-1}}$ is significantly greater than $f_{x_{c-2}}$. The user-defined parameter $\epsilon\%$ determines whether the observed difference in throughput is significant. The number of streams is decreased when the x_{c-1} value is too high so that it creates bottleneck and/or overhead. This condition is detected when $x_{c-1} > x_{c-2}$ but $f_{x_{c-1}}$ is significantly smaller than $f_{x_{c-2}}$. The algorithm does not change the number of streams when there is no significant change in throughputs between two consecutive control epochs.

The pseudo code in Algorithm 1 optimizes only one parameter, but it can easily be extended without loss of generality for cases when the number of streams is determined by more than one parameter. In such cases, `cd-tuner` iterates through all the parameters, one at a time; `cd-tuner` will move to tune the subsequent parameter when the observed throughputs do not vary over several consecutive control epochs. Note that the heuristic proposed in [5] can be seen as a simplified version of `cd-tuner` in which the number of streams is incremented by one as long as there is a significant throughput improvement.

Compass search: At each iteration, the method tries to move in the coordinate directions of the incumbent point x'. If one of these steps yields improvement, the new point becomes the incumbent. The step size λ determines the search radius from the incumbent point. If none of these steps yields improvement, the search continues with the reduced radius. See Figure 3 for an illustration.

Algorithm 2 gives the pseudo code for the customized

Algorithm 2 Compass search tuner (`cs-tuner`)

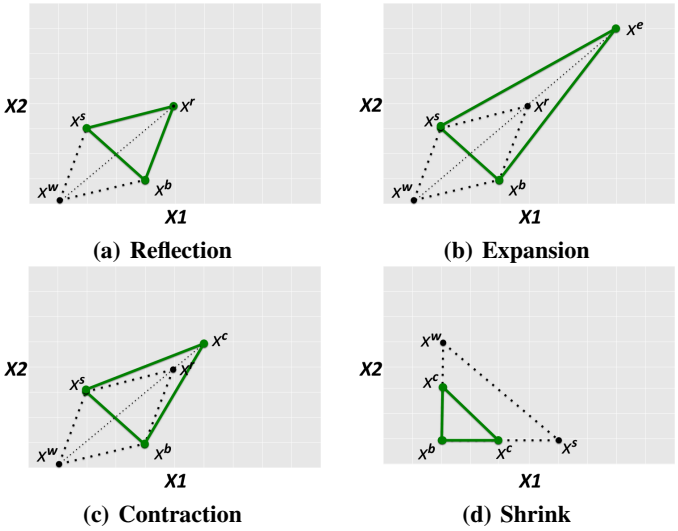
Input: starting point x_0, control epoch time e, data size s, tolerance $\epsilon\%$, step size λ

```
1  globals s' ← s, c, e
2  procedure COMPASS-SEARCH(x')
3      f_x' ← runTransfer(x')
   /* generate coordinate directions */
4      Q ← {±e_j | j = 1...m}, where e_j is the j^th unit coordinate vector such that |C| = 2^m
5      while λ > 0.5 do
6          for each q ∈ Q do
7              x_r ← x' + λ × c_j
8              x_r ← fBnd(x_r)
9              f_x_r ← runTransfer(x_r)
10             if f_x_r > f_x' then
11                 x' ← x_r; break;
12         if x_r ≠ x' then
13             λ ← λ × 0.5
14     Output: x', f_x'
15 end procedure

16 c ← 0; Δ_c ← ∞
17 x_c, f_x_c ← COMPASS-SEARCH(x_c)
18 while s' > 0 do
19     if c > 1 then
20         Δ_c ← 100 × (f_x_c-1 − f_x_c-2) / f_x_c-2
21     if |Δ_c| > ε then
22         x_c, f_x_c ← COMPASS-SEARCH(x_0)
23     else
24         x_c ← x_c-1
25     f_x_c ← runTransfer(x_c)
```

compass search tuner (`cs-tuner`). The algorithm first runs the compass search routine (lines 2–15) with initial point x_0. At each iteration, it randomly samples a coordinate direction q from the coordinate set $\mathcal{Q}$. It then generates a new point x_r from q and λ, and evaluates the throughput of x_r (lines 7–9). If an improved throughput is obtained, then the search continues from x_r. If no new point yields an improvement, the search continues from x' but with reduced step size (line 13). The search stops when λ becomes less than 0.5, in which case the coordinates degenerate to single point. In the main loop (lines 16–25), the algorithm monitors the throughput at each control epoch c; whenever there is a significant difference in consecutive observed throughputs, it invokes the compass search routine.

Compass search was originally proposed for unbounded continuous optimization problems involving real valued parameters that can take values from $-\infty$ to ∞. However, the parameters that determine the number of parallel streams take only integer values and have specific limits because of hardware/software limitations. Therefore, to handle bounded integer parameters, `cs-tuner` uses the `fBnd` function that performs two modifications. First, it forces newly generated coordinate points to have integer values by rounding off their values; for example $(3.8, 9.2)$ is rounded off to $(4, 9)$. Second, when the coordinate value is outside its bounds, it projects the point to the bound; for example $(12, -1)$ is projected to $(12, 1)$. The function `fBnd` is applied immediately whenever a new point is generated (line 8).



(a) Reflection (b) Expansion

(c) Contraction (d) Shrink

Figure 4: Illustration of Nelder-Mead search for two parameters. For a simplex with 3 vertices, x^b, x^s, and x^w are the best, second best, and worst points, respectively. The method tries to replace the worst point x^w by reflecting it through centroid to obtain x^r (Fig 4a). When the function value of x^r is better than that of x^b, x^r is expanded further to obtain x^e (Fig 4b). If expansion point does not result in improvement, x^e is contracted to obtain x^c (Fig 4c). When no improvement is found by the three operations, a smaller simplex is generated by shrinking the current simplex that results in the replacement of all vertices except x^b (Fig 4d).

Nelder-Mead search: The Nelder-Mead (NM) method navigates the search space of m tunable parameters using a geometric shape with $m + 1$ vertices called a simplex. For example, for m = 1, 2, and 3, the simplexes are a line, triangle, and tetrahedron, respectively. At each iteration, NM tries to replace the worst vertex point (lowest throughput) in the simplex via four geometric operations, namely, reflection, expansion, contraction, and shrink, parameterized by R, E, C, and S, respectively. See Figure 4 for an illustration.

Algorithm 3 shows the pseudo code of the proposed NM-based stream tuner (`nm-tuner`). The main loop of `nm-tuner` is similar to `cs-tuner`. It starts by running a Nelder-Mead procedure (lines 2–36) that begins with an initial simplex of $m+1$ points with the starting point x_0. The throughput $f_{x_\cdot}$ for each vertex $x_\cdot$ is obtained by using the `runTransfer` function. This is followed by search space navigation using reflection, expansion, contraction, and shrink operations. To handle integer parameters and their bounds, `nm-tuner` uses `fBnd`(). It forces the simplex to move only on integer values by rounded versions of the reflection, expansion, contraction, and shrink, operations (lines 11, 19, 27, 33). The search stops when the simplex degenerates to a single point.

The key advantage of these three methods over other sophisticated numerical optimization approaches is that they are computationally simple and therefore can be implemented with minimal overhead. Moreover, they do not exploit the

Algorithm 3 Nelder-Mead tuner (`nm-tuner`)

Input: starting point x_0, control epoch time e, data size s, tolerance $\epsilon\%$

```
1  globals s' ← s, c, e
2  procedure NELDER-MEAD(x')
3      Generate a simplex of m + 1 points including x'
4      for j : 1 ... m + 1 do
5          f_{x_j} ← runTransfer(x_j)
6      while simplex is not a single point do
7          /* Step1, Order and centroid: */
8          Relabel m + 1 vertices so that f_{x_0} ≥ ··· ≥ f_{x_m}
9          x̄ ← (1/m) Σ_{j≠m} x_j
10         /* Step 2, Reflect: */
11         x_r ← x̄ + R(x̄ − x_n); x_r ← fBnd(x_r)
12         f_{x_r} ← runTransfer(x_r)
13         if f_{x_0} ≥ f_{x_r} > f_{x_m} then
14             replace x_m with x_r; continue;
15         else
16             if f_{x_r} < f_{x_0} then
17                 go to step 4
18         /* Step 3, Expand: */
19         x_e ← x̄ + E(x_r − x̄); x_e ← fBnd(x_e)
20         f_{x_e} ← runTransfer(x_e)
21         if f_{x_e} ≥ f_{x_r} then
22             replace x_m with x_e; continue;
23         /* Step 4, Contract: */
24         x_t ← x_n
25         if f_{x_r} ≥ f_{x_t} then
26             x_t ← x_r
27         x_c ← x̄ + C(x_t − x̄); x_c ← fBnd(x_c)
28         f_{x_c} ← runTransfer(x_c)
29         if f_{x_c} ≥ f_{x_n} then
30             replace x_n with x_c; continue;
31         /* Step 5, Shrink: */
32         for j : 1 ... n do
33             x_j ← x_0 + S(x_j − x_0); x_j ← fBnd(x_j)
34             f_{x_j} ← runTransfer(x_j)
35     Output: x_0, f_{x_0}
36 end procedure
```

37 See lines [16—24] in Algorithm 2; Whenever Δ_c is significant, Nelder-Mead procedure is applied to find the best number of parallel streams.

full history to prune the search space, and therefore can revisit regions based on changing function values. This scheme is particularly attractive for throughput optimization because regions that are not promising may become promising as external load evolves.

The effectiveness of `cd-tuner` depends on the starting point x_0. It should be close to the critical value x^* because it requires $|x_0 - x^*|$ control epochs to reach x^*. Consequently, large difference between x_0 and x_c result in wasted bandwidth. Moreover, `cd-tuner` will be less effective when the external load changes rapidly, which will also result in wasted bandwidth. The main advantage of `cs-tuner` over `cd-tuner` is that, given a sufficiently large λ value, it can make rapid progress toward the critical point. Nevertheless, once it reaches the critical point, it will start evaluating the neighbors of the critical point until λ value becomes less than 0.5, which will result in wasted bandwidth. Therefore λ should be chosen neither too large nor too small. Similar to `cs-tuner`, `nm-tuner` can rapidly move to the critical using reflection and expansion operations. The convergence to the local solution is faster than `cs-tuner` because it uses two operations (shrinking and contraction).

IV. Experimental Analysis

In addition to the ANL and UChicago endpoints described in Section III, we consider data transfer between ANL and the Texas Advanced Computing Center (TACC). At ANL, we use the same Nehalem machine as the source; at TACC we use a Sandy Bridge node (a dual socket 16-core Intel Xeon CPU E5-2680 running at 2.70 GHz with 32 GB main memory) from the Stampede cluster. Data is transferred using `globus-url-copy` from `/dev/zero` at the source to `/dev/null` at the destination. RTT between the nodes is 33ms and the link capacity is 20 Gb/s.

We use the performance obtained for the data transfer with the tuning settings used by Globus transfer [2] as the baseline. Globus transfer is a hosted service that orchestrates and manages GridFTP transfers for the users. It selects transfer protocol parameters; monitors and retries transfers when there are faults; and allows the user to monitor status. The number of parallel streams used by Globus transfer is a product of concurrency (nc) and parallelism (np). For large files, Globus transfer uses default values of 2 and 8, respectively for nc and np. For baseline, we run transfers with these default values; we refer to this approach as `default` in our analysis.

We set the parameters for the tuners as follows. In `cs-tuner`, the step size λ is set to 8; in `nm-tuner`, R, E, C, and S are set to the customary values 1, 2, 0.5, and 0.5, respectively. In all tuners, we set the tolerance $\epsilon\%$ for significant difference to 5% and the control epoch e to 30 s.

All the discussed methods are implemented in Python. The function `runTransfer` in Algorithms 1, 2, and 3 is a Python wrapper for running the data transfer using `globus-url-copy`, where only the parallelism (np) option is supported as a command line parameter. We implement concurrency (nc) by creating nc copies of `globus-url-copy` using the `joblib.Parallel` library with the `backend='threading'` option and pin them on alternate sockets using the `taskset` system call.

We control the external load on the source by changing the values of the parameters `ext.tfr` and `ext.cmp` from the set $\{0, 16, 32, 64\}$. Note that external load on the destination endpoint and third party traffic on the network will also affect the throughput of the data transfer. However, for this study, we did not control them explicitly.

A. Detailed analysis on tuning concurrency

Figure 5 shows the results of a first experimental study in which we tune concurrency (nc) only, fixing parallelism (np) to a default value of 8.

Adaptive concurrency improves throughput: Figure 5a shows the observed throughputs of all tuners and `default` in the absence of external load. We observe that `cd-tuner`, `cs-tuner`, and `nm-tuner` obtain throughputs (∼3500 MB/s) that are 1.4x higher than that of `default` (∼2500 MB/s). We attribute the higher throughput to the dynamically tuned nc values. Figure 6a shows the values of nc found by different methods over time. We note that the default $nc = 2$ (16 parallel TCP streams) cannot saturate the network,

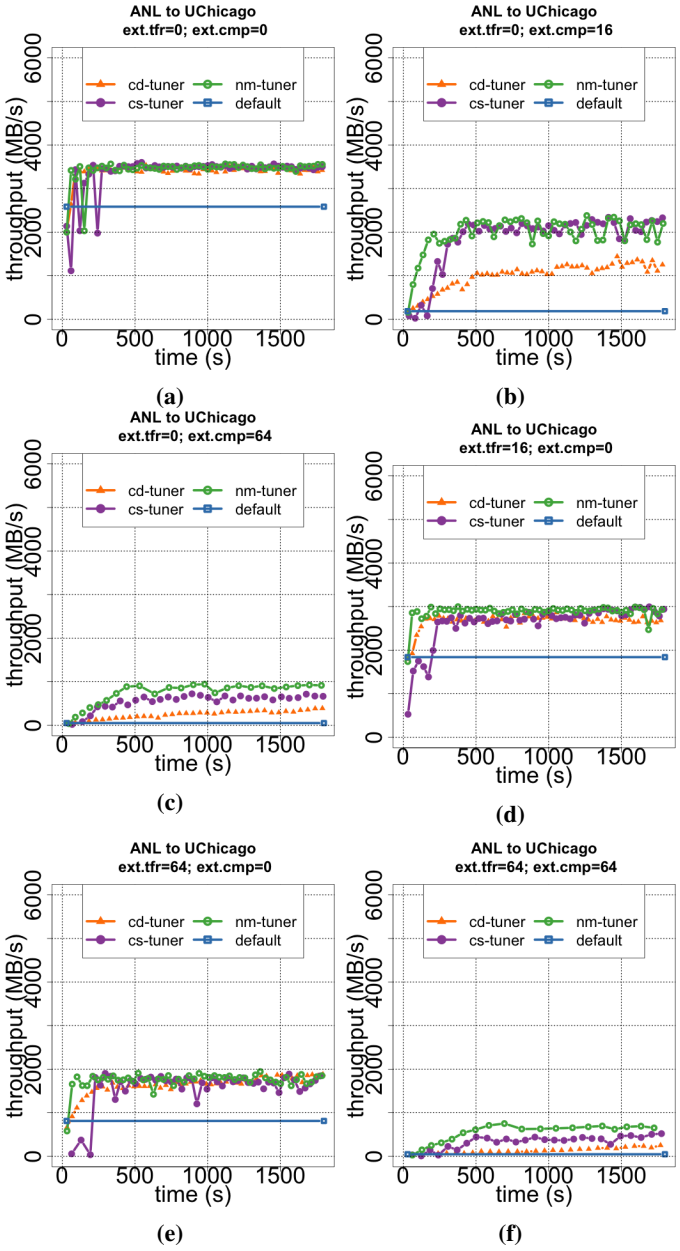


Figure 5: ANL to UChicago: Observed throughputs (with overhead) obtained by tuners under various external loads

but $nc = 5$ (40 parallel TCP streams) does and results in throughput improvement.

External compute load affects the throughput in a significant way: From Figures 5b and 5c, we note that introducing the external compute load on the source affects the observed throughput much more than the external transfers do. When `ext.cmp` is set to 16 and 64, throughputs obtained by `default` are reduced to 200 MB/s and 100 MB/s, respectively; however, `cs-tuner` and `nm-tuner` reach 1500 MB/s and 1000 MB/s, resulting in 7x and 10x improvement over `default`. Figures 6b and 6c show that `cs-tuner` and `nm-tuner` obtain higher throughputs by setting the value of nc to 50 to 80, respectively. Although `cd-tuner` does not reach the throughputs of `cs-tuner` and `nm-tuner`, it provides 2x improvement over `default`.

External transfer load reduces the throughput: Figures 5d and 5e show observed throughputs when the external traffic is introduced at the source. By increasing `ext.tfr` from 0 to 16 and 64, throughputs obtained by `default` reduce from 2500 MB/s to 1400 MB/s and 900 MB/s, respectively. In contrast, `cd-tuner`, `cs-tuner`, and `nm-tuner` achieve throughputs of 3000 MB/s (`ext.tfr`=16) and 1800 MB/s (`ext.tfr`=64), respectively, each 2x higher than that of `default`. The nc values are adapted to 25 (`ext.tfr`=16) and 35 (`ext.tfr`=64; see Figure 6c), respectively.

`cd-tuner` is sensitive to the starting point, but `cs-tuner` and `nm-tuner` are robust: In the absence of external load, `cs-tuner` and `nm-tuner` take 500 s to reach steady-state throughput but `cd-tuner` takes only 100 s (see Figure 6a). Since the starting value for nc is 2, `cd-tuner` can reach the best value within three control epochs, whereas `cs-tuner` and `nm-tuner` perform large steps in the beginning, resulting in relatively slower convergence to steady state. When the external compute load and traffic are introduced (Figures 6b, 6d) `cd-tuner` becomes less effective because the best nc values are not close to 2. Nevertheless, the large step sizes of `cs-tuner` and `nm-tuner` allow them to find an appropriate value for nc in about the same amount of time (500–600 s; 16–20 control epochs).

Overhead under external compute load is significant: Each call to the `globus-url-copy` must load the executable into memory, allocate the buffer and required data structures, create the required number of threads, transfer the data, and free the memory. In `cd-tuner`, `cs-tuner`, and `nm-tuner`, the `globus-url-copy` is restarted for every control epoch. Consequently, the restart overhead in these methods will be significantly higher than that of the `default`, where the `globus-url-copy` is stopped only when the transfer is over. In an ideal scenario, `globus-url-copy` will allocate a sufficient number of threads and adapt the value of nc without requiring restart. We call the throughput without overhead the *best-case throughput*; we determine this value by aggregating the throughput values reported by nc copies of `globus-url-copy`. The best-case throughputs obtained by the tuners, shown in Figure 7, are significantly higher than those reported in Figure 5.

In the absence of external compute load and traffic, `cd-tuner`, `cs-tuner`, and `nm-tuner` obtain steady-state throughput of 4000 MB/s, compared with 3500 MB/s shown in Figure 5a; the overhead corresponds to 17% reduction in throughput. When `ext.cmp` is set to 16 and 64, throughput reduction increases to 33% and 50%, respectively. Under these conditions, the direct search methods achieve higher throughput by increasing the value of nc but at the expense of significant overhead. When `ext.cmp` is set to 16 and 64, throughput does not reduce significantly and the reduction remains at ∼15%. The nc values selected in response to external network load are small relative to those selected in response to external compute load; clearly, running 64 copies of `globus-url-copy` is computationally less intensive than

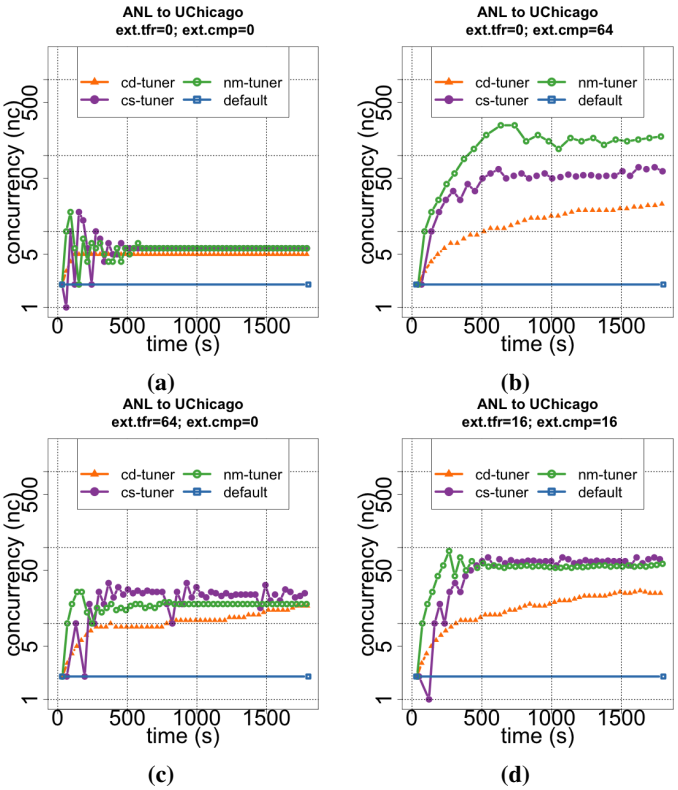


Figure 6: ANL to UChicago: Concurrency values adopted by tuners under various external loads.

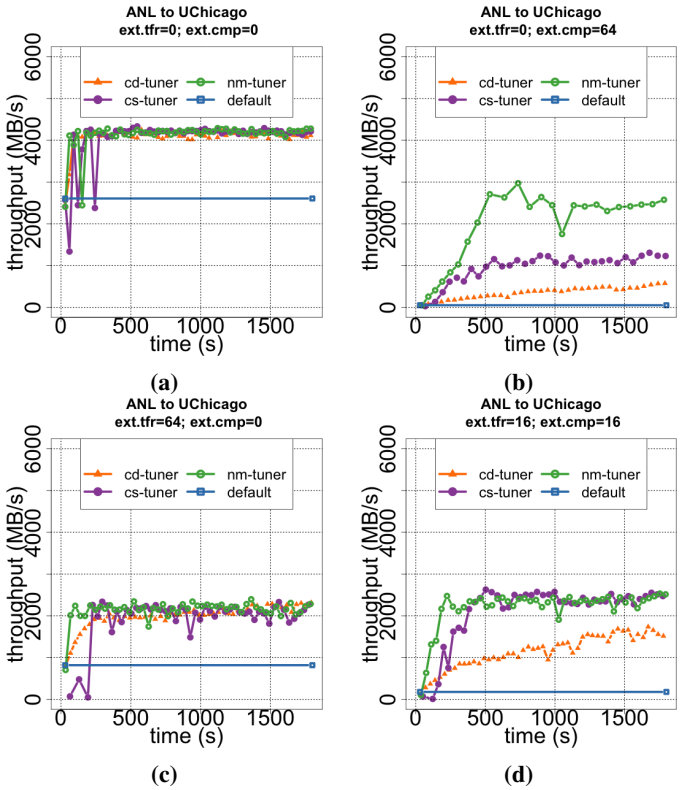


Figure 7: ANL to UChicago: Best-case throughput under various external loads.

64 `dgemm` copies.

Trend is similar on ANL to TACC: The throughputs obtained by various tuners on ANL to TACC show a trend that is similar to that of ANL to UChicago. An exception is that without any external load, the `default` and direct search tuners achieve 1900 MB/s. Although the achievable throughput without overhead is 2200 MB/s in the direct search tuners, because of the restart overhead, they achieve the same throughput as `default`. `cs-tuner` and `nm-tuner` search and adapt large values for nc (between 45 and 50) even in the absence of external load. Consequently, the convergence of `cd-tuner` to the best nc is slower. For all other external load cases, `cs-tuner` and `nm-tuner` obtain throughput improvements between 1.5x and 10x.

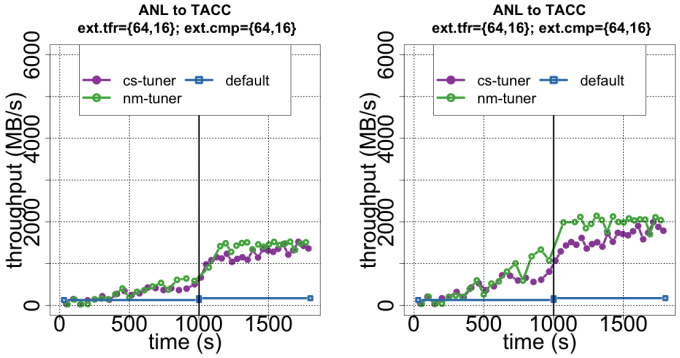
B. Tuning concurrency and parallelism and adaptation to external load

Now we focus on tuning concurrency and parallelism simultaneously. In addition, we assess the adaptiveness of `cs-tuner` and `nm-tuner` by changing the external load during the transfer. For each tuner, we allow 1800 s of data transfer. From 0 to 1000 s, we set `ext.tfr` and `ext.cmp` to 64 and 16, respectively; after 1000 s, both `ext.tfr` and `ext.cmp` are set to 16. We compare throughputs of `cs-tuner` and `nm-tuner` to that of `default`, which is run under the same type of external load. Since `cd-tuner` is less effective under changing external load conditions, we did not include it in the analysis.

Concurrency is important: Figure 8 shows throughputs obtained for the ANL to TACC transfers. We observe that both `cs-tuner` and `nm-tuner` obtain throughputs that are significantly better than that of `default`; the tuners obtain 1.3x (up to 1000 s) and 10x (after 1000 s) improvements. Figures 8c and 8d show the trajectories of the concurrency and parallelism adopted by `cs-tuner` and `nm-tuner` over time. The throughputs in Figures 8a and 8b follow the trajectory of the associated concurrency values. On the other hand, varying parallelism values have only a minor impact on the throughput. We observe that `nm-tuner` and `cs-tuner` follow different search strategies as they maximize the observed throughput; `nm-tuner` adopts large nc values to reach higher throughput but the overhead associated with creating nc copies results in reduced observed throughput. `cs-tuner` achieves a similar throughput with relatively smaller nc values but with lower overhead. We observed a similar trend for ANL to UChicago transfers (see Figure 9).

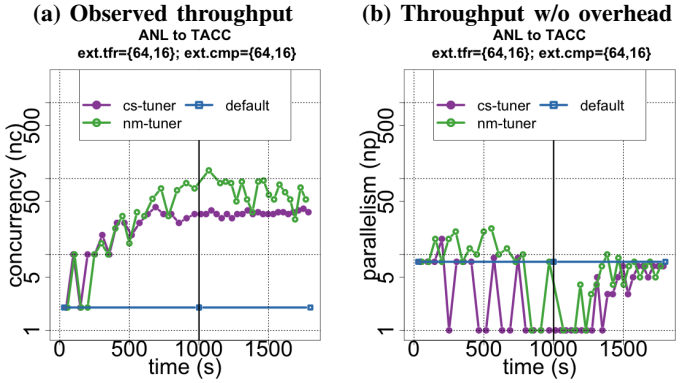
C. Comparison with existing heuristics

We compare `nm-tuner` with the simple heuristic proposed by Balman et al. [5] and to the expert-knowledge-based heuristic proposed by Yildirim et al. [25], which we label `heur1` and `heur2`, respectively. We modified `heur1` in a similar way to `cd-tuner` so that it can be used to tune more than one parameter. Figure 10 shows representative results for ANL to TACC transfers. For various external conditions, `nm-tuner` and `heur2` obtain throughputs that are significantly better



(a) Observed throughput

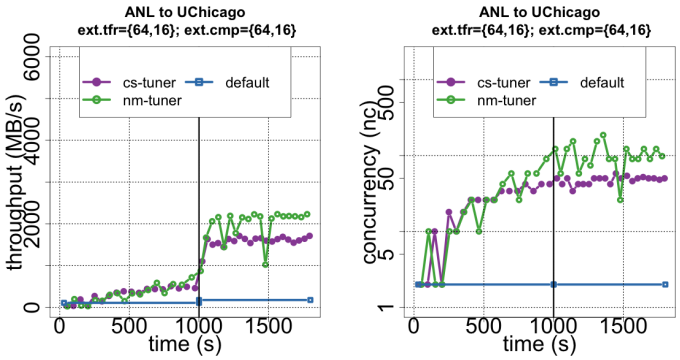
(b) Throughput w/o overhead



(c) Concurrency

(d) Parallelism

Figure 8: ANL to TACC. Tuning concurrency and parallelism under varying external load.



(a) Observed throughput

(b) Concurrency

Figure 9: ANL to UChicago. Tuning concurrency and parallelism under varying external load.

than those of `heur1`. Both `nm-tuner` and `heur2` reach the maximum achievable throughput in a few control epochs. The additive increment strategy means that `heur1` requires a larger number of control epochs to reach throughputs comparable to those obtained by `nm-tuner` and `heur2`. Although the effectiveness of `heur2` stems from the fact that it adopts an aggressive exponential increment strategy, it is sensitive to the starting values. For example, when the starting values of np and nc are larger than the critical values, then `heur2` (unlike `nm-tuner`) has no decrement mechanism to reduce these values. Consequently, it will terminate with poor parallel stream settings that will eventually affect the observed throughput.

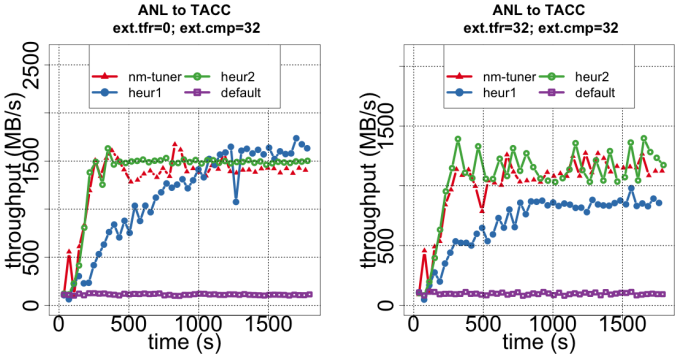


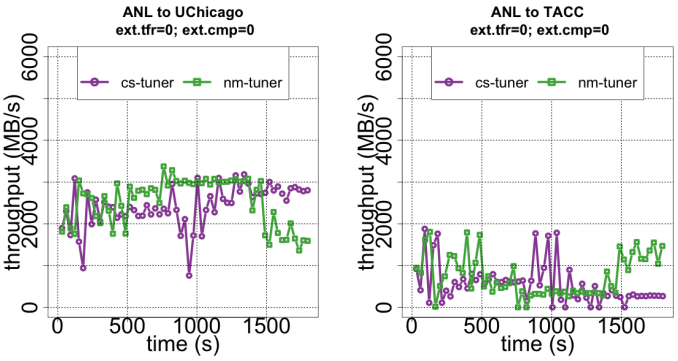
Figure 10: Comparison with existing heuristics: Tuning concurrency and parallelism under varying external load.

D. Simultaneous tuning

We examine what happens when two dynamically tuned transfers share the same endpoint. We run two data transfers simultaneously, one from ANL to UChicago and one from ANL to TACC, and use `nm-tuner` (or `cs-tuner`) to set the concurrency and parallelism parameters for each transfer. Note that the tuning performed for one transfer is not aware of the tuning performed for the other; thus each transfer treats the other as external load.

Figures 11a and 11b show the results when `nm-tuner` (or `cs-tuner`) is used to tune nc and np dynamically for each of the two transfers, without any other load (i.e., `ext.tfr=ext.cmp`=0). We observe that the throughput of the ANL to UChicago transfer is improved at the expense of the ANL to TACC transfer. The reason is that the ANL to UChicago network can support up to 5000 MB/s throughput. Consequently the tuner for the ANL to UChicago transfer adopts a large number of parallel streams to inject data into the network. Doing so reduces the ANL to TACC transfer's throughput because the outgoing traffic shares the same NIC at ANL. In response, the tuner for the ANL to TACC transfer increases the number of streams that it uses, in order to gain additional bandwidth (see `nm-tuner` from 1500 s to 1800 s). We see evidence of complex interactions between the two flows, which result for example in the ANL to UChicago transfer often getting a much larger fraction of the available outgoing bandwidth at ANL. Further study is required to determine the reasons for these behaviors, which may be due to different RTTs or loss rates, or to the temporal ordering of control epochs, for example.

Recall that we formulated the problem 1 in Sec II as optimizing the throughput of individual transfers. Consequently, the proposed tuners operate at the individual transfer level but not at the endpoint level. To handle multiple transfers, we may need to aggregate transfers involving a common endpoint and then optimize the parallelism and concurrency values for all such transfers simultaneously using `nm-tuner` or `cs-tuner`. In so doing, we may be able to apply the methods proposed by Kettimuthu et al. [16] to prioritize transfers.



(a) **(b)**

Figure 11: Simultaneous data transfer from ANL to UChicago and TACC. The transfers are tuned by `nm-tuner` (`cs-tuner`) at the same time.

V. Conclusion

We have presented a new approach to the problem of improving data transfer throughput via the use of parallel TCP streams. We proposed a model-free approach, a significant departure from the analytical/empirical model-based approaches that dominate the literature. Specifically, we formalized the problem of optimizing data transfer throughput as a model-free dynamic optimization problem. We customized direct search methods, a class of mathematical optimization algorithms, to adapt the number of parallel TCP streams based on the external load. We evaluated the feasibility and effectiveness of the proposed approach by applying it to a widely used file transfer tool, `globus-url-copy`, under various load conditions, and identified under what conditions the adaptive strategy is effective. We showed that the direct search methods can greatly improve achieved transfer throughput when significant external load is exerted on the source endpoint.

Our future work includes (1) broadening the approach to enable disk-to-disk optimization over sets of transfers with different file sizes; (2) investigating ways to reduce the restart overhead to increase the responsiveness of the proposed methods; (3) developing an end-to-end performance modeling and tuning framework for high performance computing facilities; and (4) extending the tuning approach to take into account the destination endpoint.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory.

References

[1] B. Allcock et al. Globus striped GridFTP framework and server. In *SC*, 2005.

[2] B. Allen et al. Software as a service for data scientists. *Commun. ACM*, 2012.

[3] E. Altman, D. Barman, B. Tuffin, and M. Vojnovic. Parallel TCP sockets: Simple model, throughput and validation. In *IEEE INFOCOM*, 2006.

[4] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *17th Ann. Joint Conf. of the IEEE Comp. and Comm. Soc.*, volume 1, pages 252–262. IEEE, 1998.

[5] M. Balman and T. Kosar. Dynamic adaptation of parallelism level in data transfer scheduling. In *Intl. Conf. on Complex, Intelligent and Software Intensive Systems*, pages 872–877. IEEE, 2009.

[6] L. Eggert, J. Heidemann, and J. Touch. Effects of ensemble TCP. *ACM SIGCOMM Comp. Comm. Rev.*, 30(1):15–29, 2000.

[7] Y. Gu and R. L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[8] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[9] T. J. Hacker, B. D. Athey, and B. Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *IEEE Int. Par. and Dist. Proc. Sym.*, Washington, DC, USA, 2002.

[10] T. J. Hacker, B. D. Noble, and B. D. Athey. Adaptive data block scheduling for parallel TCP streams. In *14th IEEE Int. Sym. on High Perf. Dist. Comp.*, pages 265–275. IEEE, 2005.

[11] T. Ito, H. Ohsaki, and M. Imase. On parameter tuning of data transfer protocol GridFTP for wide-area grid computing. In *2nd Intl. Conf. on Broadband Networks*, pages 1338–1344. IEEE, 2005.

[12] T. Ito, H. Ohsaki, and M. Imase. GridFTP-APT: Automatic parallelism tuning mechanism for data transfer protocol GridFTP. In *6th IEEE Int. Sym. on Clus. Comp. and the Grid*, pages 454–461, Washington, DC, USA, 2006. IEEE Computer Society.

[13] W. E. Johnston, E. Dart, M. Ernst, and B. Tierney. Enabling high throughput in widely distributed data management and analysis systems: Lessons from the LHC. In *TERENA Net. Conf. (TNC)*, 2013.

[14] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comp. Comm. Rev.*, 33(2):83–91, 2003.

[15] R. Kettimuthu. Type- and workload-aware scheduling of large-scale wide-area data transfers. https://etd.ohiolink.edu/, 2015.

[16] R. Kettimuthu et al. An elegant sufficiency: Load-aware differentiated scheduling of data transfers. In *SC*, 2015.

[17] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM review*, 45(3):385–482, 2003.

[18] D. Leith and R. Shorten. H-TCP: TCP for high-speed and long-distance networks. In *2nd Intl. Workshop on Protocols for Fast Long-Distance Networks*, 2004.

[19] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante. Modeling and taming parallel TCP on the wide area network. In *19th IEEE Int. Par. and Dist. Proc. Sym.*, IPDPS '05, Washington, DC, USA, 2005.

[20] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[21] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *ACM/IEEE Conference on Supercomputing*, page 37. IEEE Computer Society, 2000.

[22] B. Tierney. Experiences with 40G/100G applications. http://meetings.internet2.edu/speakers/4067/, 2014.

[23] V. Torczon. On the convergence of pattern search algorithms. *SIAM Journal on optimization*, 7(1):1–25, 1997.

[24] S. J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[25] E. Yildirim, E. Arslan, J. Kim, and T. Kosar. Application-level optimization of big data transfers through pipelining, parallelism and concurrency. *IEEE Transactions on Cloud Computing*, in press, 2016.

[26] E. Yildirim and T. Kosar. End-to-end data-flow parallelism for throughput optimization in high-speed networks. *Journal of Grid Computing*, 10(3):395–418, 2012.

[27] E. Yildirim, D. Yin, and T. Kosar. Prediction of optimal parallelism level in wide area data transfers. *IEEE Trans. Parallel Distrib. Syst.*, 22(12):2033–2045, Dec. 2011.

[28] D. Yin, E. Yildirim, S. Kulasekaran, B. Ross, and T. Kosar. A data throughput prediction and optimization service for widely distributed many-task computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):899–909, 2011.