v020826.1830

Efficient Construction of Liquid Schedules for Collective Communication

Emin Gabrielyan, Roger D Hersch

École Polytechnique Fédérale de Lausanne
{Emin.Gabrielyan,RD.Hersch}@epfl.ch

Abstract

We propose a method for the optimal scheduling of collective data exchanges relying on the knowledge of the underlying network topology. We introduce the concept of liquid schedules. Liquid schedules ensure the maximal utilization of a network's bottleneck links and offer an aggregate throughput as high as the flow capacity of a liquid in a network of pipes. The collective communication throughput offered by liquid schedules in highly loaded networks may be several times higher in comparison with the throughput of traditional topology-unaware techniques such as round-robin or random schedules. This paper presents an efficient technique for building of liquid schedules. To create a liquid schedule we need to find the smallest partition of a set of transfers into subsets of mutually non-congesting transfers. Increasing the traffic size results in an exponential explosion of the possible combinations of non-overlapping subsets of mutually non-congesting transfers. By first limiting the search space to transfers using bottleneck links, we strongly reduce the search space without affecting the solution space. Measured liquid throughputs on a real computer cluster are very close to the theoretical predictions.

Keywords: Optimal network utilization, collective data exchange, liquid schedules, network topology, topology-aware scheduling.

1. Introduction

The interconnection topology is one of the key factors influencing the performance of parallel applications [1], [2], [3], [4]. The aggregate throughput is, among other factors, influenced by the transfer block size. Due to protocol overhead, point to point communication throughput tends to decrease with the decrease of the message size. However, smaller messages allow a more progressive utilization of network links and reduce network congestions. Intuitively, the data flow becomes liquid when the packet size tends to zero [5], [6], [7], [8]. The aggregate throughput of a collective data exchange depends on the application's underlying network topology. The amount of data that has to pass across the most loaded links gives the utilization time of the most loaded links. The total amount of data divided by the utilization time of the most loaded links gives an estimation of the *liquid throughput* of the network for the considered collective data exchange. The liquid throughput corresponds to the flow capacity of a non-compressible fluid in a network of pipes [6]. Since data streams are broken into packets, congestions may occur. Therefore the aggregate throughput of a collective data exchange may be lower than its liquid throughput. The rate of congestions for a given data exchange depends on the sequence of transfers forming the data exchange (i.e. the schedule). Previous research efforts were focused on the optimization and

scheduling of collective communications over wavelength division multiplexing optical networks [9] and over satellite-switch time division multiplexing networks [10].

Unlike flow control based congestion avoidance mechanisms [11] [12], we establish schedules for the data transfers without trying to regulate the sending processors' data rate. We specifically address the problem of reaching the flow capacity of a fluid in a network by trying to optimally schedule the set of transfers of a collective data exchange.

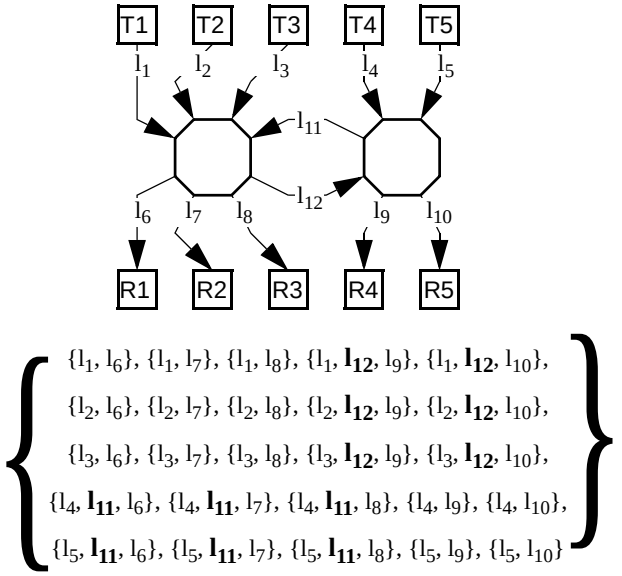


Fig. 1. Example of a data exchange composed of 25 transfers

For example, consider the all-to-all collective data exchange shown in Fig. 1. There are 5 transmitting processors (T1,... T5), each of them sending a packet to each of the receiving processors (R1... R5). The network consists of 12 links. Links l_{11} and l_{12} are the most loaded links, since each of them will be used by 6 transfers. The most loaded links are the bottlenecks of the collective data exchange. They have the longest active time. In the best case, the duration of a collective data exchange is as long as the active time of the bottleneck links, i.e. the collective data exchange reaches its liquid throughput. A *round-robin* schedule is carried out within 5 logical phases: *(1)* {T1→R1, T2→R2 ... T5→R5}, *(2)* {T1→R2, T2→R3 ... T5→R1}, etc. The round-robin schedule's throughput is however lower than the liquid throughput, since bottleneck links l_{11} and l_{12} are idle in step 1 (Fig. 2). Phases 3 and 4 are carried out at a lower rate, since they contain congesting transfers.

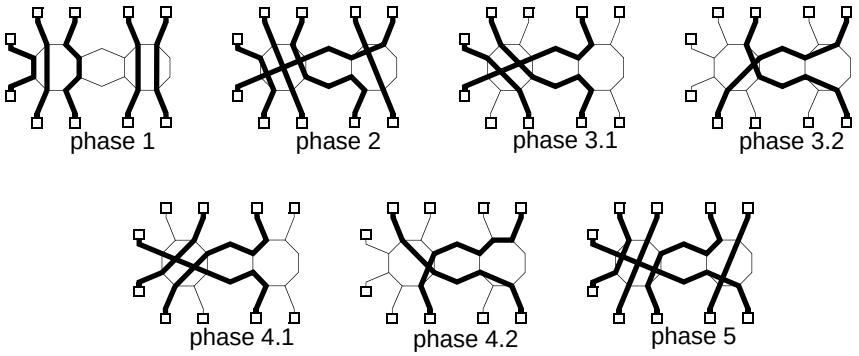


Fig. 2. Round-robin schedule of transfers.

Fig. 3 shows that a schedule achieving the liquid throughput for the considered collective data exchange exists.

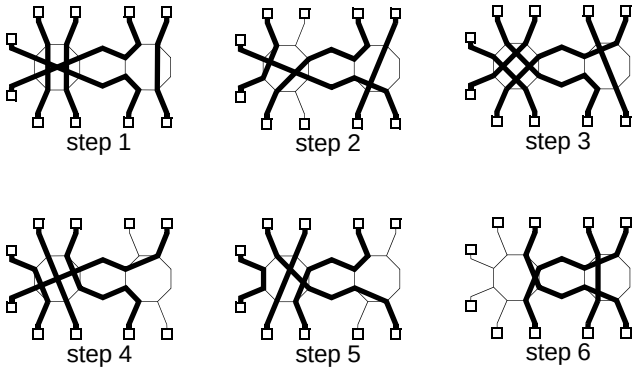


Fig. 3. An optimal schedule.

The method we propose allows to compute optimal schedules for complex network topologies and considerably increases collective data exchange throughputs compared with traditional topology-unaware techniques such as round-robin, random or asynchronous transfer schedules.

Let us introduce the definitions that are required for computing optimal schedules. A single "point-to-point" transfer is represented by the set of communication links forming the network path between a transmitting and a receiving processor according to a given static routing scheme.

A *transfer* is a set of links (i.e. the path between a sending processor and a receiving processor). A *traffic* is a set of transfers (i.e. the collective data exchange). Fig. 1 shows the traffic for an all-to-all data exchange. The all-to-all data exchange is just a particular case of a traffic. Any collective exchange comprising transfers between a set of emitting and a set of receiving processors is a traffic[1]. A link l is *utilized* by a transfer x if $l \in x$. A link l is utilized by a traffic X if l is utilized by a transfer of X. Two transfers are in *congestion* if they share a common link. If they don't use a

1. A processor may in the general case be at the same time an emitting and a receiving unit.

common link they are *simultaneous*. We limit ourselves to data exchanges consisting of identical packet sizes.

A *simultaneity* of a traffic X is a subset of X consisting of mutually non-congesting transfers. A transfer is in congestion with a simultaneity if the transfer is in congestion with at least one element of the simultaneity. A simultaneity of a traffic is *full* if all transfers in the complement of the simultaneity in the traffic are in congestion with the simultaneity. A simultaneity of a traffic is processed in the timeframe of a single transfer. $\lambda(l, X)$, the *load* of link l in the traffic X is the number of transfers in X using l. The *duration* $\Lambda(X)$ of a traffic X is the maximal value of the load among all links involved in the traffic. The links having maximal load values are called *bottlenecks.* The *liquid throughput* of a traffic X is the ratio $\#(X)/\Lambda(X)$ multiplied by the single link throughput, where $\#(X)$ is the number of transfers in the traffic X. For example, the traffic X shown in Fig. 1 has a number of transfers $\#(X) = 25$ and the duration of the traffic is $\Lambda(X) = 6$. Its aggregate liquid throughput is the ratio $25/6$ of a single link throughput, i.e. $(25/6) \times 100MB/s = 416.67MB/s$, assuming a single link throughput of 100 *MB/s*.

Let us define a simultaneity of X as a *team* of X if it uses all bottlenecks of X (note that a traffic may not have a team, see Fig. 6). A team of X is *full* if it is a full simultaneity of X. Let $\mathfrak{R}(X)$ and $\mathfrak{I}(X)$ be respectively the sets of all full simultaneities and all full teams of X.

In order to form liquid schedules, we try to schedule transfers in such a way that all bottleneck links are always kept busy. Therefore we search for a liquid schedule by trying to assemble non-overlapping teams carrying out all transfers of the given traffic. To cover the whole solution space we need a means of generating all possible teams of a given traffic. This leads to an exponential complexity. It is therefore important that the team traversing technique be efficient and that each configuration is evaluated once and only once, without repetitions.

In sections 2 and 3, we present techniques for the construction of full simultaneities and full teams. In section 4 we show how to build a liquid schedule and prove that we find a solution whenever it exists. Predicted and measured liquid throughputs on a real network are given in section 5. Section 6 draws the conclusions.

2. Obtaining full simultaneities

The construction of liquid schedules requires the ability of traversing the set of all full teams of an arbitrary traffic. To limit redundant search steps, each full team should be constructed once and only once. We first optimize the retrieval of all simultaneities and then use that algorithm to retrieve all full teams.

Recall that in a traffic X, any mutually non-congesting combination of transfers is a simultaneity. A full simultaneity is a combination of non-congesting transfers taken from X, such that its complement in X contains only transfers congesting with that simultaneity.

We can categorize full simultaneities according to the presence or absence of a given transfer *x*. A full simultaneity is *x*-positive if it contains transfer *x*. If it does not contain transfer *x*, it is *x*-negative. Thus the set of full simultaneities $\mathfrak{R}(X)$ is partitioned into two non-overlaping subsets: an *x*-positive and *x*-negative subset of $\mathfrak{R}(X)$. For example, if *y* is another transfer, the set of *x*-positive full simultaneities may be further partitioned into *y*-positive and *y*-negative subsets. Repetition of this concept allows us to design a recursive technique traversing whole set of all full simultaneities $\mathfrak{R}(X)$ one by one without repetitions.

Let us define a *category* of full simultaneities of *X* as an ordered triplet (*excluder, depot, includer*), where the includer is a simultaneity of *X* (not necessarily full) and the transfers of *X* non-congesting with the includer are either in the depot or in the excluder.

We say that a full simultaneity is *covered* by a category *R*, if the full simultaneity contains all the transfers of the category's includer and does not contain any transfer of the category's excluder. Consequently, any full simultaneity covered by a category is the category's includer together with some transfers taken from the category's depot. The collection of all full simultaneities of *X* covered by a category *R* is defined as the *coverage* of *R*. We denote the coverage of *R* as $\phi(R)$.

The category $(\varnothing, X, \varnothing)$ is a *prim-category* since it covers all full simultaneities of *X*, i.e. $\phi(\varnothing, X, \varnothing) = \mathfrak{R}(X)$.

By taking an arbitrary transfer *x* from the depot of a category *R*, we partition the coverage of *R* into *x*-positive and *x*-negative subsets. The *x*-positive and *x*-negative subsets of a coverage of *R* respectively are coverages of two categories derived from *R*: a positive sub-category and a negative sub-category of *R*.

The positive sub-category R_{+x} is formed from the category *R* by adding transfer *x* to its includer, and removing from its depot and excluder[1] all transfers congesting with *x*. The negative sub-category R_{-x} is formed from the category *R* by moving transfer *x* from its depot to its excluder (see Fig. 4).

1. Since transfers congesting with *x* can not be in a full simultaneity covered by R_{+x}, we may safely remove them from the excluder.

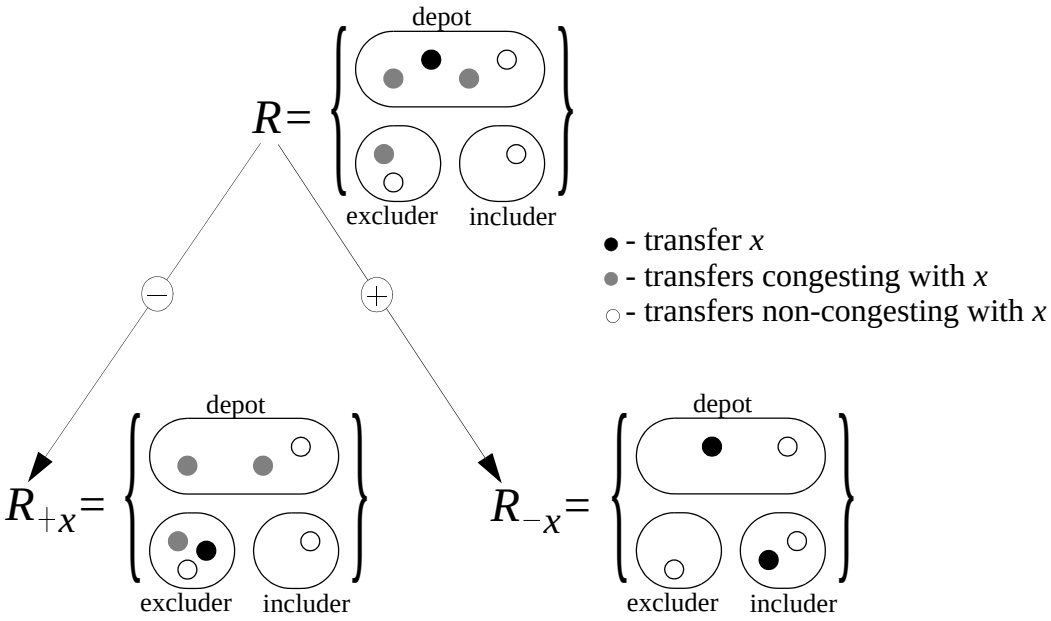


Fig. 4. Fission of a category into two sub-categories

The coverage of R is partitioned by the coverages of its sub-categories R_{+x} and R_{-x}, i.e. the coverage of a category is the union of coverages of its sub-categories: $\phi(R_{+x}) \cup \phi(R_{-x}) = \phi(R)$, where the coverages of the sub-categories have no common transfers, $\phi(R_{+x}) \cap \phi(R_{-x}) = \varnothing$. The replacement of a category R by its two sub-categories R_{+x} and R_{-x} is defined as a *binary fission* of a category.

A *singular* category is a category that covers only one full simultaneity. That full simultaneity is equal to the includer of the singular category. The depot and excluder of a singular category are empty.

We apply the binary fission to the prim-category and split it into two categories. Then, we apply the fission to each of these categories. Repeated fission increases the number of categories and narrows the coverage of each category. Finally, the fission will lead to singular categories only, i.e. categories whose coverage consists of a single full simultaneity. Since at each stage we have been partitioning the set of full simultaneities, at the final stage we know that each full simultaneity is covered by one and only one singular category.

The algorithm carries out recursively the fission of categories and yields all full simultaneities without repetitions.

There is a further optimization to be considered. Full simultaneities covered by a category have no transfer from the category's excluder. Therefore each full simultaneity covered by a category must

contain a congesting transfer for each member of the excluder. Since we keep in the excluder transfers which do not congest with the includer, congesting transfers must be taken from the depot. A category whose depot doesn't have a congesting transfer for at least one of the excluder's transfers is *blank*. The coverage of a blank category is empty and there is therefore no need to pursue its fission.

Let a category within X be *idle* if its includer and its depot together don't use all bottlenecks of X. The coverage of an idle category does therefore not contain a team.

An algorithm that is carrying out successive fissions, starting from the prim-ancestor and contiguously removing all the blank and idle categories ultimately leads to all full teams.

3. Speeding up full team formation

This section presents a further method for speeding up the search for all full teams $\mathfrak{I}(X)$ of an arbitrary traffic X.

Let us consider from the original traffic X only those transfers that use bottlenecks of X and call this set of transfers *skeleton* of X. We denote the skeleton of X as $\varsigma(X)$. Obviously, $\varsigma(X) \subset X$.

Considering the skeleton of a traffic X as another traffic, the bottlenecks of the skeleton of a traffic are the same as the bottlenecks of the traffic. Consequently, a team of a skeleton is also a team of the original traffic.

Let us obtain all full teams of the traffic's skeleton by applying the fission algorithm eliminating the idle categories.

Then, a full team of the original traffic may be obtained by adding a combination of non-congesting transfers to a team of the traffic's skeleton.

We therefore obtain the set of a traffic's full teams $\mathfrak{I}(X)$, by carrying out the following steps:

1. Obtain the set of the skeleton's full teams $\mathfrak{I}(\varsigma(X))$ by applying the fission algorithm.
2. Create for each skeleton's full team a category by
   2.1. initializing the includer with the transfers of the skeleton's full team,
   2.3. initializing the excluder as empty,
   2.2. and putting into the depot all transfers of X non-congesting with the includer.
3. Apply the fission to each category, discarding the check for idle categories, since the includer is already a team, i.e. it uses all bottlenecks.

By first applying the fission to the skeleton and then expanding the skeleton's full teams to the traffic's full teams, we strongly reduce the required processing time and at the same time we obtain all full teams of the original traffic without repetitions.

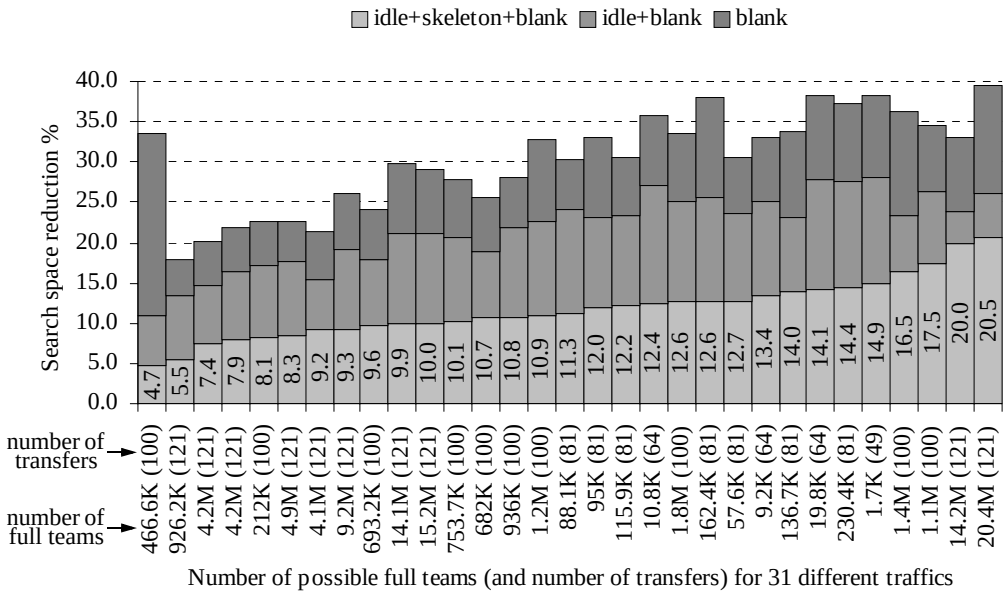


Fig. 5. Search space reducement

We measured the reduction in search space according to the different search space reduction methods we propose. We consider 31 different traffics within the T1 32 node cluster computer (Fig. 8). The search space is given by the number of nodes that are being explored within the recursion tree. Fig. 5 shows the obtained search space reductions compared with a naive algorithm that would build full teams without any of the proposed optimisations. The skeleton algorithm reduces on average the search space to 12.48%, i.e. full teams are computed 8 times faster than without search space reduction techniques. Note that all presented algorithms, including the naive algorithm, are smart enough to avoid repetitions of full simultaneities.

4. Liquid schedules

Having the capability of building full teams, let us now show how to compute the liquid schedule of a data exchange. A *schedule* α of a traffic X is a collection of simultaneities of X partitioning the traffic X. A *step* of a schedule α is an element of the schedule α. $\#(\alpha)$, the *length* of a schedule α, is the number of steps in α. A schedule of a traffic is *optimal* if the traffic does not have any shorter schedule. If the length of a schedule is equal to the duration of the traffic then the schedule is *liquid*. A liquid schedule is optimal, but the inverse is not always true, meaning that a traffic may not have a liquid schedule. Fig. 6. demonstrates a traffic that does not have a team and therefore no liquid schedule. Fig. 7 shows a liquid schedule for the collective traffic shown in Fig 1.

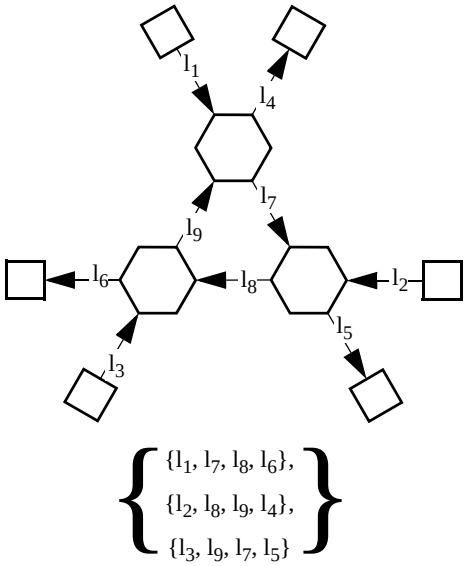


Fig. 6. This traffic has no team and no liquid schedule.

The duration of a traffic X is the load of its bottlenecks. If a schedule is liquid, then each of its timeframes must use all bottlenecks. Inversely, if all steps of a schedule use all bottlenecks, the schedule is liquid.

$$\left\{ \begin{array}{l} \left\{ \begin{array}{c} \{l_1, \mathbf{l_{12}}, l_9\}, \\ \{l_2, l_7\}, \\ \{l_3, l_8\}, \\ \{l_4, \mathbf{l_{11}}, l_6\}, \\ \{l_5, l_{10}\} \end{array} \right\} \left\{ \begin{array}{c} \{l_1, \mathbf{l_{12}}, l_{10}\}, \\ \{l_2, l_6\}, \\ \{l_4, \mathbf{l_{11}}, l_7\}, \\ \{l_5, l_9\} \end{array} \right\} \left\{ \begin{array}{c} \{l_1, \mathbf{l_8}\}, \\ \{l_2, \mathbf{l_{12}}, l_9\}, \\ \{\mathbf{l_3}, l_6\}, \\ \{l_4, l_{10}\}, \\ \{l_5, \mathbf{l_{11}}, l_7\} \end{array} \right\} \\ \left\{ \begin{array}{c} \{l_1, l_7\}, \\ \{l_2, \mathbf{l_8}\}, \\ \{\mathbf{l_3}, \mathbf{l_{12}}, l_9\}, \\ \{l_5, \mathbf{l_{11}}, l_6\} \end{array} \right\} \left\{ \begin{array}{c} \{l_1, l_6\}, \\ \{l_2, \mathbf{l_{12}}, \mathbf{l_{10}}\}, \\ \{\mathbf{l_3}, l_7\}, \\ \{\mathbf{l_4}, \mathbf{l_{11}}, \mathbf{l_8}\} \end{array} \right\} \left\{ \begin{array}{c} \{\mathbf{l_3}, \mathbf{l_{12}}, \mathbf{l_{10}}\}, \\ \{\mathbf{l_4}, \mathbf{l_9}\}, \\ \{\mathbf{l_5}, \mathbf{l_{11}}, \mathbf{l_8}\} \end{array} \right\} \end{array} \right\}$$

Fig. 7. A liquid schedule of the collective traffic shown in Fig. 1.

The necessary and sufficient condition for the liquidity of a schedule is that all bottlenecks be used by each step of the schedule. Since a simultaneity of X is defined as a *team* of X, if it uses all bottlenecks of X, an equivalent condition for the liquidity of a schedule α on X is that each step of α be a team of X.

4.1. Liquid schedule naive search algorithm

Let us propose a naive liquid schedule construction technique. Consider all possible teams of the original traffic *X*. Take one of them (for example A_1) and consider it as the first step of the liquid schedule. Remove the team A_1 from the traffic and look at the reduced traffic. The choice for the second step is limited by only those teams of the original traffic *X*, which are included in the reduced traffic $X - A_1$. Take a candidate for the second step from the current choice (for example $A_{1,1}$). Remove from the traffic the team $A_{1,1}$ as well. Similarly the choice for the third step is limited by only those teams of the original traffic *X*, which are included in the reduced traffic $X - A_1 - A_{1,1}$, etc.

The algorithm recursively search a liquid schedule in the depth-wise order. A node in the search tree represents a dead end if there are no choice for the successive step, i.e. no team of the original traffic may be formed from the reduced traffic (i.e. not yet carried out transfers). At the dead end nodes the algorithm backtracks one or more steps back and analyses other possibilities. The algorithm stops at the node whose reduced traffic is a team of the original traffic. The collection of all teams on the path from the root to that terminal node is a liquid schedule.

4.2. Speeding up the search of a liquid schedule

Let's analyse the liquid schedule shown in Fig. 7. Remove from the traffic a few steps of the schedule and look at the reduced traffic.

Load of bottlenecks decreases in the reduced traffic. However the reduced traffic may contain additional bottlenecks. More steps of a liquid schedule are carried out more additional bottlenecks appears in the reduced traffic (not yet carried out transfers).

The construction strategy of a liquid schedule presented in the subsection 4.1. form a choice of candidates to the successive step from all teams of the original traffic *X* included in the reduced traffic.

However note, that the steps of the liquid schedule, are not only teams for the original traffic, but they are also teams of the corresponding reduced traffics. We are going to prove that this property of steps is a necessary and sufficient condition of the liquidity of a schedule. Note that this properly is valid independently from the order of steps.

Therefore a step which is a team of the original traffic (uses the bottlenecks of the original traffic) but does not use the additional bottlenecks of the reduced traffic (isn't a team of the reduced traffic) may not lead to a liquid schedule. Such a choice ultimately brings to a dead end and algorithm shall backtrack for evaluating other choices.

Since the set of bottlenecks in the reduced traffic is larger than the set of bottlenecks of the original traffic, the number of teams of the original traffic is much fewer than the number of teams of the original traffic (included in the reduced traffic). Therefore by limiting our choice at each step by the set of teams of the reduced traffic we considerably reduce the search space without affecting the solution space.

DISCUSSION. Suppose A is a timeframe of a liquid schedule α on a traffic X. Therefore A is a team of α. Remove the team A from X so as to form a new traffic $X - A$. The duration of the new traffic $X - A$ is the load of the bottlenecks in $X - A$. The bottlenecks of X are also the bottlenecks of $X - A$. The load of a bottleneck of X decreases by one in the new traffic $X - A$ (note that the new traffic $X - A$ may have additional bottlenecks). The schedule α shortened by one element A is a schedule for $X - A$. The new schedule $\alpha - \{A\}$ has as many timeframes as the duration of the corresponding new traffic $X - A$. Therefore, if α is a liquid schedule on X then for any of its step A the schedule $\alpha - \{A\}$ is a liquid schedule on $X - A$.

Consider traffic X as a problem whose solution is a liquid schedule α. The technique presented in section 3, is capable of generating the set of all teams of X. If X has a solution α then a timeframe A of the schedule α is a member of the set of all teams of X and $\alpha - \{A\}$ is a schedule on $X - A$. Therefore the problem X can be reduced into smaller problems. Examine each possible team A of X and search inductively (e.g. recursively) a solution for $X - A$. If a solution exists for X, then this method will find it. If the method does not find a solution for X, then, since we explored the full solution space, we conclude that X does not have a liquid schedule.

We limit at each iteration our choice to the collection of only those teams of the original traffic which are also teams of the current reduced sub-traffic (having an expanded number of bottlenecks). By doing so, we considerably reduce the search space without affecting the solution space.

By limiting the choice of the next step only by full teams of the reduced traffic we again reduce the search space of the construction algorithm. Let us show that here the solution space is not affected as well. Let us modify a liquid schedule so as to convert one of its teams into a full team. Let X (a traffic) have a solution α (a liquid schedule). Let A be a timeframe of α. If A is not a full team of X, then, by moving the necessary transfers from other timeframes of α, we can convert timeframe A to a full team. Evidently, the properties of liquidity (partitioning, simultaneousness and length) of α will not be affected. Therefore if X has a solution then it has also a solution when the team of one of its steps is full, hence the choice of the teams in the construction may be narrowed from the set of all teams to the set of full teams only.

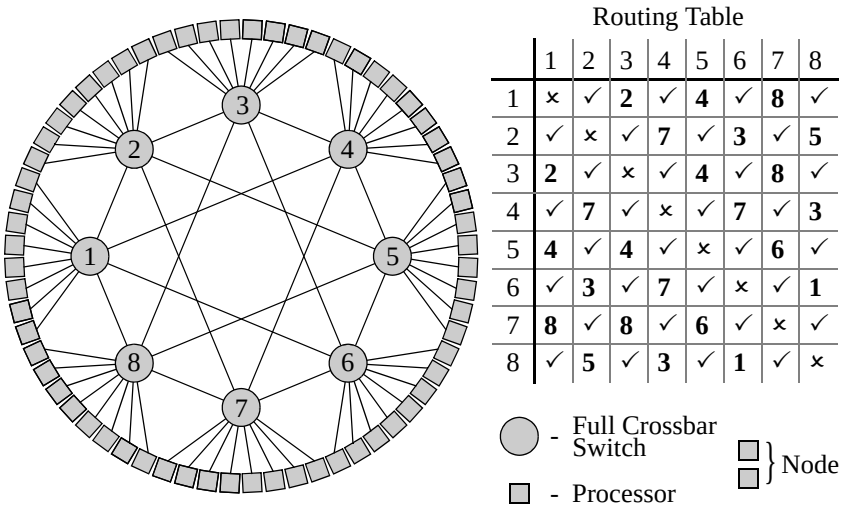
By a choice of a full team A of a traffic X we are faced with the new smaller problem of searching a liquid schedule for a traffic $X - A$. The traffic $X - A$ may not have a solution, or it may not have even a team. In these cases we have to backtrack to evaluate other choices. Evaluation of all choices ultimately leads to a solution if it exists.

Fig. 7 shows a liquid schedule built as explained above. Each its successive step being a team of the reduced traffic incorporates all bottlenecks of that reduced traffic (shown in bold).

Thanks to the presented chain of optimizations, for more than 90 percent of our testbed topologies (see Section 5) the search of liquid schedules took less than one tenth of a second on a single 500MHz processor. For 8 topologies out of 363 solution was not found within 24 hours.

5. Testbed and measurements

In this section we present a testbed consisting of test traffics for different topologies. Measurements of collective data exchange throughputs will help us to validate the efficiency of our scheduling strategy.



	1	2	3	4	5	6	7	8
1	×	✓	**2**	✓	**4**	✓	**8**	✓
2	✓	×	✓	**7**	✓	**3**	✓	**5**
3	**2**	✓	×	✓	**4**	✓	**8**	✓
4	✓	**7**	✓	×	✓	**7**	✓	**3**
5	**4**	✓	**4**	✓	×	✓	**6**	✓
6	✓	**3**	✓	**7**	✓	×	✓	**1**
7	**8**	✓	**8**	✓	**6**	✓	×	✓
8	✓	**5**	✓	**3**	✓	**1**	✓	×

Fig. 8. Architecture of the T1 cluster computer.

As basic network topology for our testbed, we use the Swiss-T1 cluster (called henceforth T1, see Fig. 8). The network of the T1 forms a K-ring [13] and has a static routing scheme. The throughputs of all links are identical and equal to 86*MB/s*. The cluster consists of 32 nodes, each one comprising 2 processors [14], [15].

The test traffics are selected from different configurations of all-to-all collective data exchanges between a set of emitting and receiving processors, where each emitting processor sends one packet to each receiving processor. Within each node, one processor is an emitting and the other processor is a receiving unit. Therefore any given allocation of nodes gives us an equal number of emitting and receiving processors.

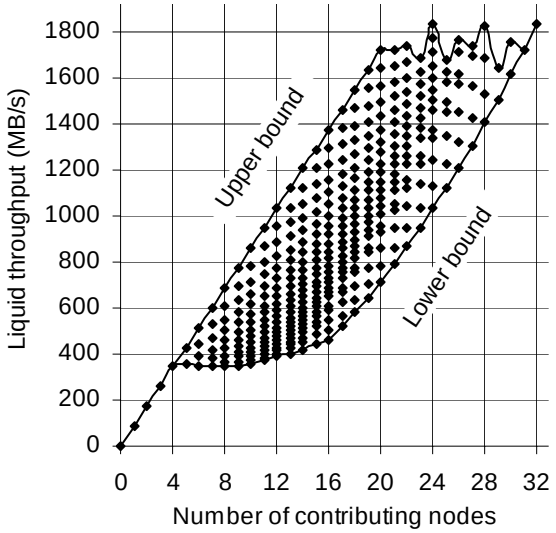


Fig 9. Liquid throughput in relation to the number of nodes with variations according to sub-topologies.

Since the T1 cluster incorporates 32 nodes, there exist 2^{32} = 4294967296 possible allocations of nodes to an application. Considering only the number of nodes in front of each switch, there are only 5^8 = 390625 different node allocations, since there are 8 switches having each *n* used nodes ($0 \le n \le 4$). Because of symmetries within the network, many of these topologies are identical. To limit our choice to really different topologies, we've computed the liquid throughputs for each of 390625 topologies, taking in account the network's real routing tables. This resulted in only 363 different liquid throughput values. Accordingly, we have formed a test-bed consisting of 363 really different topologies. Each topology is characterized by its liquid throughput and the number of allocated nodes (see Fig. 9)[1].

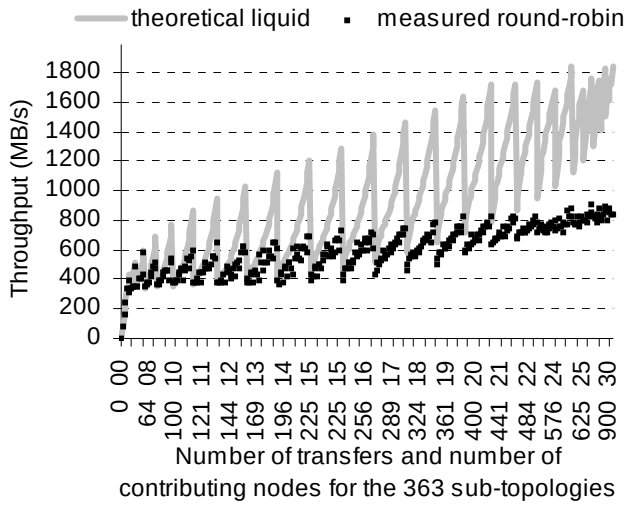


Fig. 10. Theoretical liquid throughput and measured round-robin schedule throughput for 363 network sub-topologies.

We have sorted these 363 traffics and placed them along an axis. They are sorted first by the number of nodes and then according to the value of the liquid throughput. Fig. 10 shows the liquid throughput values together with the measured throughput of a round-robin schedule.

For each measurement, the amount of data transferred from a transmitting processor to a receiving processor is equal to 2*MB*. For each topology, 20 measurements were made. The black dots represent the median of the collected results. Measured throughputs of the round-robin schedule are far below the network's potential liquid throughput. Throughputs of collective exchanges carried out according to a random schedule do not perform better.

1.The figure demonstrates that depending on the sub-topology, the liquid throughput for a given number of nodes may considerably vary.

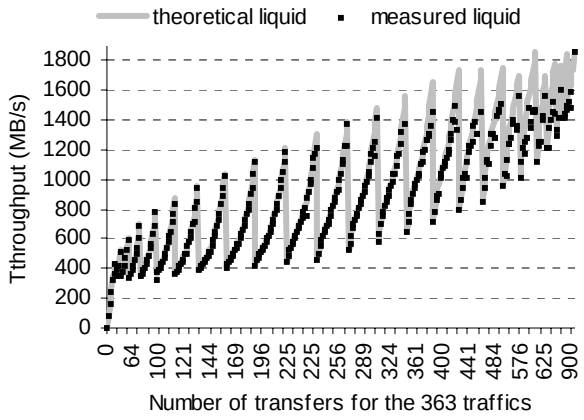


Fig. 11. Measured throughputs carried out according to computed liquid schedules.

We measured the throughput computed with our liquid scheduling technique, for the 363 data exchanges on the T1 cluster. Fig. 11 shows the measured throughput compared with the theoretical liquid throughputs.

Each black dot represents the median of 7 measurements. Processor to processor transfers have a size of 5*MB*. The measured throughputs are close to the theoretically computed liquid throughput. For many sub-topologies, the proposed scheduling technique allows to increase the aggregate throughput by a factor of two compared with a simple, topology un-aware scheduling technique.

Thanks to the proposed liquid schedule search space reduction techniques, computing a liquid schedule takes for more than 97% of the considered sub-topologies of the T1 cluster less than 1/10 of a second on a single 500MHz Alpha processor.

6. Conclusion

We propose a method for scheduling collective data exchanges in order to obtain an aggregate throughput equal to the network's liquid throughput. This is achieved by building at each step of the schedule mutually non-congesting sets of transfers using all bottleneck links. Exploration of the full solution space yields a liquid schedule if it exists. The proposed search space reduction techniques make the approach practical for networks having in the order of ten interconnected crossbar switches.

On the Swiss T1 cluster computer, the proposed scheduling technique allows for many sub-topologies to increase the collective data exchange throughput by a factor of two.

In the future, we intend to explore how to extend the presented scheduling technique in order to dynamically reschedule collective data exchanges when the set of planned exchanges evolves over time.

References

[1] H. Sayoud, K. Takahashi, B. Vaillant, "Designing communication network topologies using steady-state genetic algorithms", IEEE Communications Letters, Vol. 5, No. 3, March 2001, 113-115.

[2] Pangfeng Liu, Jan-Jan Wu, Yi-Fang Lin, Shih-Hsien Yeh, "A simple incremental network topology for worm-hole switch-based networks", Proc. 15th International Parallel and Distributed Processing Symposium, 2001, 6-12.

[3] P.K.K. Loh, Wen Jing Hsu, Cai Wentong, N. Sriskanthan, "How network topology affects dynamic loading balancing", IEEE Parallel & Distributed Technology: Systems & Applications, Vol. 4, No. 3, Fall 1996, 25-35.

[4] V. Puente, C. Izu, J. A. Gregorio, R. Beivide, J. M. Prellezo, F. Vallejo, “Improving parallel system performance by changing the arrangement of the network links”, Proc. of the International Conference on Supercomputing, May 2000, 44-53.

[5] M. Naghshineh, R. Guerin, “Fixed versus variable packet sizes in fast packet-switched networks”, Proc.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM '93., Networking: Foundation for the Future, IEEE Press, Vol. 1, 1993, 217-226.

[6] Benjamin Melamed, Khosrow Sohraby, Yorai Wardi, “Measurement-Based Hybrid Fluid-Flow Models for Fast Multi-Scale Simulation”, DARPA/NMS BAA 00-18 AGREEMENT No. F30602-00-2-0556, *http:// www.darpa.mil/ito/research/nms/meetings/nms2001apr/Rutgers-SD.pdf*

[7] K.G. Yocum, J.S. Chase, A.J. Gallatin, A.R. Lebeck, “Cut-through delivery in Trapeze: An Exercise in Low-Latency Messaging”, 6th IEEE International Symposium on High Performance Distributed Computing, 1997, 243-252.

[8] N.M.A. Ayad, F.A. Mohamed, “Performance analysis of a cut-through vs. packet-switching techniques”, Proc. Second IEEE Symposium on Computers and Communications, 1997, 230-234.

[9] J.-C. Bermond, L. Gargano, S. Perennes, A. A. Rescigno, and U. Vaccaro, “Efficient collective communication in optical networks”, Proc. of ICALP'96. Lecture Notes in Computer Science, 574-585, 1996.

[10] R. Jain, G. Sasaki, “Scheduling packet transfers in a class of TDM hierarchical switching systems”, IEEE International Conference on Communications ICC '91, Vol. 3, 1991, 1559-1563.

[11] Dah-Ming Chiu, Raj Jain, “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks”, Computer Networks and ISDN Systems, 1989, Vol. 17, 1-14.

[12] H. Ozbay, S. Kalyanaraman, A. Iftar, “On rate-based congestion control in high-speed networks: Design of an H-infinity based flow controller for single bottleneck”, Proc. of the American Control Conference, June 1998, 2376-2380.

[13] P. Kuonen, “The K-Ring: a versatile model for the design of MIMD computer topology”, Proc. of the High-Performance Computing Conference (HPC'99), San Diego, USA, April 1999, 381-385.

[14] Pierre Kuonen, Ralf Gruber, “Parallel computer architectures for commodity computing and the Swiss-T1 machine”, EPFL Supercomputing Review, Nov 99, pp. 3-11, *http://sawww.epfl.ch/SIC/SA/publications/SCR99/ scr11-page3.html*

[15] Ralf Gruber, “Commodity computing results from the Swiss-Tx project Swiss-Tx Team”, *http://www.grid-computing.net/documents/Commodity_computing.pdf*

[16] G. Campers and O. Henkes and J. P. Leclerq “Graph Coloring Heuristics: A Survey, Some New Propositions and Computational Experiences on Random and ‘{L}eighton's’ Graphs” Proc. Operational Research, 917-932, 1988.

[17] A. Hertz and D. de Werra “Using Tabu Search Techniques for Graph Coloring” in Computing(39) 345-351, 1987.

Annex

The search for a liquid schedule requires to partition the traffic into a set of non-overlapping mutually non-congesting transfers. The problem can also be formulated as an intersection graph colouring problem [16], [17]. Vertices of the graph are formed by transfers. Edges between vertices represent congestions between transfers.

Fig. 12 shows the graph whose vertices are to be coloured for the collective data exchange of Fig. 1. Vertex $x_{n,m}$ corresponds to a transfer from an emitting processor *n* to a receiving processor *m*. For example vertex $x_{4,1}$ represents the transfer T4→R1={l_4, $\mathbf{l_{11}}$, l_6}. The bold edges of the graph show congestions of transfers due to specific bottleneck links.

Whenever a liquid schedule exists, an optimal solution of the graph colouring problem is a liquid schedule. The chromatic number of the graph's optimal colouring is the length of the liquid schedule. Vertices having the same colour represent a step of the liquid schedule.

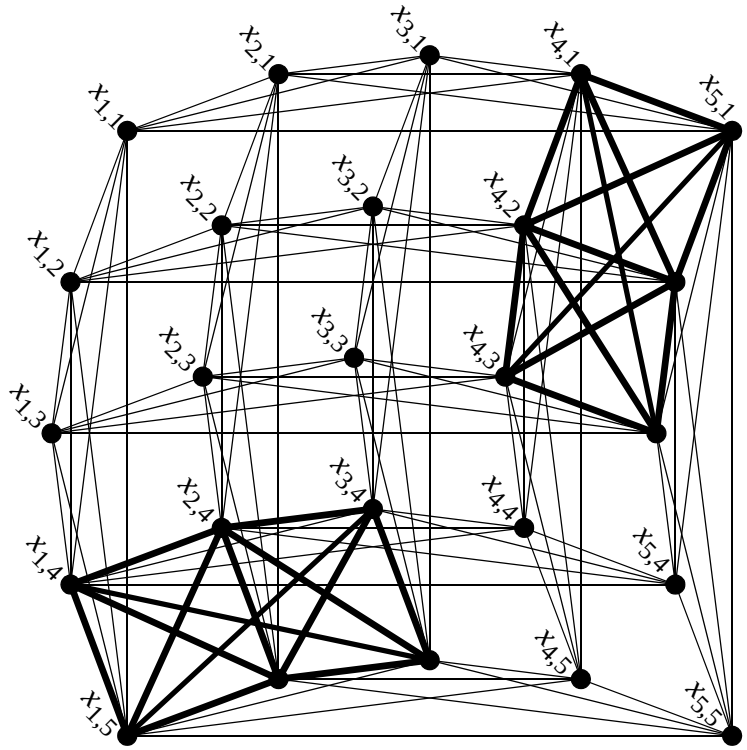


Fig. 12. Graph corresponding to the data exchange shown in the Fig. 1. The 25 vertices of the graph represent the transfers and the edges represent congestion relation between transfers.

Fig. 13 shows the ratio of the number of vertices of a graph to the density of its edges. We can label each edge of the graph by the link(s) causing the congestion. An all-to-all data exchange on the Swiss T1 cluster with 32 transmitting and 32 receiving processors forms a graph with $32 \times 32 = 1024$ vertices and 48704 edges. The approach we propose allows to compute in advance the chromatic number of the graph's optimal colouring (length of the liquid schedule). Furthermore, we further reduce the problem by first trying to "colour" vertices having edges representing all bottleneck links (creation of teams). Then we work on the reduced graph, formed by the original graph minus the coloured vertices (forming teams on sub-traffics). This suggests that our problem is a more constrained problem than the general graph colouring problem. It remains to be checked how our solution compares with solutions to variants of the graph colouring problem.

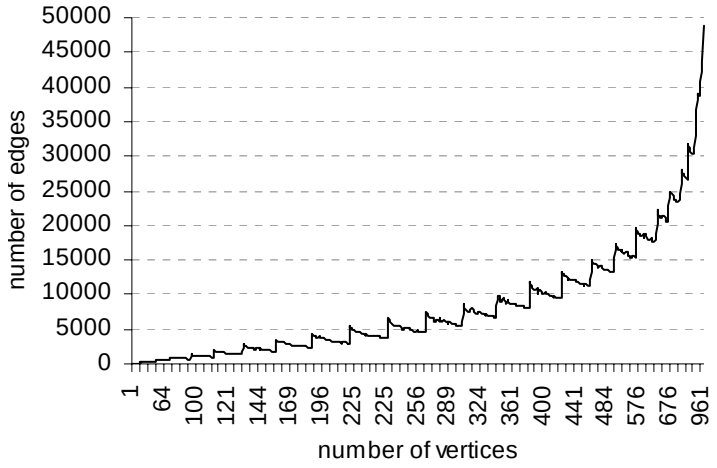


Fig. 13. Characteristics of the graphs corresponding to 363 data exchanges of the testbed shown in the Fig. 9.