Abstract

The upper limit of a network’s capacity is its liquid throughput. The liquid throughput corresponds to the flow of a liquid in an equivalent network of pipes. However, the aggregate throughput of a collective communication pattern (traffic) scheduled according to network topology unaware techniques may be several times lower than the maximal potential throughput of the network. In most of the cut-through, wormhole and wavelength division optical networks, there is a loss of performance due to congestions between simultaneous transfers sharing a common communication resource. We propose to schedule the transfers of a traffic according to a schedule yielding the liquid throughput. Such a schedule, called liquid schedule, relies on the knowledge of the underlying network topology and ensures an optimal utilization of all bottleneck links. To build a liquid schedule, we partition the traffic into time frames comprising mutually non-congesting transfers keeping all bottleneck links busy during all time frames. The search for mutually non-congesting transfers utilizing all bottleneck links is of exponential complexity. We present an efficient algorithm which non-redundantly traverses the search space and limits the search to only those sets of transfers, which are non-congesting and use all bottleneck links. We further propose a liquid schedule construction technique, which reduces the search space without affecting the solution space.

1. Introduction

Efficient Liquid Schedule Search Strategies for Collective Communications

Collective multicast communications are of increasing importance both in scientific and in commercial applications. Numerous applications require an efficient use of network resources for collective communications. Such applications comprise parallel acquisition and distribution of multiple video streams [Chan01], [Sitaram00], switching of simultaneous voice communication sessions [H323], [EWSD04], [SIP04], and high energy physics, where particle collision events need to be transmitted from a large number of detectors and filters to clusters of processing nodes [CERN01]. The aggregate throughput of a collective communication pattern (traffic) depends on the underlying network topology. The amount of data that has to pass across the most loaded links of the network, called bottleneck links, gives their utilization time. The total size of a traffic divided by the utilization time of the bottleneck links gives an estimation of the liquid throughput, which corresponds to the flow capacity of a non-compressible fluid in a network of pipes [Melamed00]. Both in wormhole switching networks and in Wavelength Division Multiplexing (WDM) optical networks, due to possible link or wavelength allocation conflicts, not any combination of transfer requests may be carried out simultaneously. The objective is to minimize the number of timeslots and/or wavelengths required to carry out a given set of transfer requests. Each transfer shall be allocated to one (and only one) time frame, such that no pair of transfers allocated to the same time frame use a common resource (link, wavelength).

The liquid scheduling problem is hard to solve. Solving the problem by applying a heuristic graph colouring algorithm, as shown in our previous publications, provides in short time a suboptimal solution. In the present contribution we propose an exact method for computing liquid schedules. In the majority of cases the method is fast enough to allow real time scheduling of a traffic.

2. The liquid scheduling problem

Let us consider a network topology (Fig. 1) consisting of ten end nodes $t_1 \ldots t_5, r_1 \ldots r_5$ (henceforth called processors), two wormhole cut-through switches $s_a, s_b$ and twelve unidirectional links $l_1 \ldots l_{15}, l_{r_1} \ldots l_{r_5}, l_{ab}, l_{ba}$ having identical throughputs. The processors $t_1 \ldots t_5$ only transmit data and $r_1 \ldots r_5$ only receive data. It’s easy to guess the routing, e.g. a message from $t_4$ to $r_3$ traverse links $l_{ab}, l_{ba}$ and $l_{r_3}$, and a message from $t_1$ to $r_2$ uses only links $l_{r_1}$ and $l_{r_2}$.

We denote transfers symbolically to mark out the occupied network links. For example the transfer from $t_4$ to
Let each sending processor have messages to be transmitted to each receiving processor and let all messages have identical sizes [Naghshineh93]. Thus, in the present example, we have 25 transfers to carry out. Each of the ten links \(l_{1},l_{15},l_{r_{1}},\ldots,l_{r_{5}}\) carries 5 transfers and the two links \(l_{ab},l_{ba}\) must each carry 6 transfers. Therefore the links \(l_{ab},l_{ba}\) are the network bottlenecks and have the longest active time. If the duration of the whole collective communication is as long as the active time of the bottleneck links, we say that the collective communication reaches its liquid throughput. In that case the bottleneck links are obviously kept busy all the time along the duration of the communication traffic. Assuming in this example a single link throughput 1Gbps, the liquid throughput offered by the network is \((25/6) \times 1Gbps = 4.17Gbps\). Under identical packet size and link throughputs (kept all along this paper for the sake of simplicity) the liquid throughput of a traffic \(X\) is the ratio \#(\(X\))/\(\Lambda(\(X\))\) multiplied by the single link throughput, where \#(\(X\)) is the total number of transfers and \(\Lambda(\(X\))\) is the number of transfers carried out by one bottleneck link.

Now let us see if the order in which the transfers are carried out in this wormhole network has an impact on the collective communication performance. A straightforward schedule to carry out these 25 transfers is the round-robin schedule, according to which at first each transmitting processor sends the message to the receiving processor staying in front of it, then to the receiving processor staying at the next position, etc. Such a round robin schedule consists of 5 phases. The transfers of the first and second \(\{\text{transfers}\}\) and fifth \(\{\text{transfers}\}\) phase of the round-robin schedule may be carried out simultaneously, but the third \(\{\text{transfers}\}\), \(\{\text{transfers}\}\), \(\{\text{transfers}\}\) and fourth \(\{\text{transfers}\}\), \(\{\text{transfers}\}\), \(\{\text{transfers}\}\) phases contain congesting transfers, e.g. link \(l_{ab}\) (marked thick) can not be simultaneously used by the two transfers \(\{\text{transfers}\}\) and \(\{\text{transfers}\}\). None of these two phases can be carried out in less than two time frames and therefore the whole schedule lasts 7 time frames, instead of seemingly 5. Therefore the performance of our collective communication carried out according to the round-robin schedule corresponds to the throughput of 25/7 = 3.57 messages per time frame or \((25/7) \times 1Gbps = 3.57Gbps\), which is less than the liquid throughput.

Nevertheless, a solution exists to schedule the 25 transfers within 6 time frames. The sequence of time frames \{\text{transfers}\} is an example of the liquid schedule for the 25-transfer collective communication request.

3. Definitions

The method we propose allows us to efficiently build liquid schedules for non-trivial network topologies. Thanks to liquid schedules we may considerably increase the collective data exchange throughputs, compared with traditional topology unaware schedules such as round-robin or random schedules. The present section introduces the definitions that will be further used for describing the liquid schedule construction method.

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 routing schema. 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. a collective data exchange).

\[
\begin{align*}
\{t_{11},t_{12}\}, \{t_{12},t_{13}\}, \{t_{13},t_{14}\}, \{t_{14},t_{15}\}, \{t_{15},t_{1a},t_{1b}\}, \{t_{15},t_{1ab},t_{1a}\} \\
\{t_{12},t_{13},t_{14}\}, \{t_{13},t_{14},t_{15}\}, \{t_{14},t_{15},t_{1a}\}, \{t_{15},t_{1ab},t_{1a}\}, \{t_{15},t_{1ab},t_{1b}\} \\
\{t_{1a},t_{1b},t_{ab},t_{15}\}, \{t_{1a},t_{1b},t_{ab},t_{15}\}, \{t_{1a},t_{1b},t_{ab},t_{15}\}, \{t_{1a},t_{1b},t_{ab},t_{15}\}, \{t_{1a},t_{1b},t_{ab},t_{15}\}
\end{align*}
\]

Fig. 2. Example of a traffic composed of 25 transfers carried out over the network shown on Fig. 1.
Fig. 2 is a particular case of a traffic. Any collective exchange comprising transfers between possibly overlapping sets of sending and receiving processors is a traffic.

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. Note that we will be limiting 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 member 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 that simultaneity. A simultaneity of a traffic obviously can be carried out within one time frame (the time to carry out a single transfer). The load \( \lambda(l, X) \) of link \( l \) in a traffic \( X \) is the number of transfers in \( X \) using link \( l \). The duration \( \Lambda(X) \) of a traffic \( X \) is the maximal value of the load among all links involved in the traffic.

\[
\Lambda(X) = \max\{\lambda(l, X)\} \quad \forall l
\]

The links having maximal load values, i.e. \( \lambda(l, X) = \Lambda(X) \), 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 \( X \).

We define a simultaneity of \( X \) as a team of \( X \) if it uses all bottlenecks of \( X \). A team of \( X \) is full if it is a full simultaneity of \( X \). Let \( \mathcal{R}(X) \) and \( \mathcal{I}(X) \) 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, i.e. we partition the traffic into teams. To cover the whole solution space we need to generate all possible teams of a given traffic. This is an exponentially complex problem. It is therefore important that the team traversing technique be non-redundant and efficient, i.e. each configuration is evaluated once and only once, without repetitions.

4. Obtaining full simultaneities

To obtain all full teams, 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 \( \mathcal{R}(X) \) is partitioned into two non-overlapping subsets: an \( x \)-positive and \( x \)-negative subset of \( \mathcal{R}(X) \). 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. Iteration of this concept allows us to recursively traverse the whole set of all full simultaneities \( \mathcal{R}(X) \), 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), the excluder contains some transfers of \( X \) non-congesting with the includer and the depot contains all the remaining transfers non-congesting with the includer.

A category, defined by the transfers of its includer and excluder, constrains a subset of full simultaneities. We therefore 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 \( \phi(R) \).

Transfers of a category’s includer form a simultaneity (not full). By adding different variations of transfers from the depot, we may obtain all possible full simultaneities covered by the category.

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

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 respective \( x \)-positive and \( x \)-negative subsets of a coverage of \( R \) are coverages of two categories derived from \( R \): a positive subcategory and a negative subcategory of \( R \).

The positive subcategory \( R_{+x} \) is formed from the category \( R \) by adding transfer \( x \) to its includer, and by removing from its depot and excluder\(^1\) all transfers

1. Since transfers congesting with \( x \) are naturally excluded from a full simultaneity covered by \( R_{+x} \), we may safely remove them from the excluder (and avoid redundancy in the exclusion constraint).
congesting with \( x \). The negative subcategory \( R_{-x} \) is formed from the category \( R \) by moving transfer \( x \) from its depot to its excluder. The replacement of a category \( R \) by its two subcategories \( R_{+x} \) and \( R_{-x} \) is defined as a "fission" of the category. Fig. 3 and Fig. 4 show an example of fission of a category into positive and negative 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: \( \phi(R_{+x}) \cup \phi(R_{-x}) = \phi(R) \), and the coverages of the sub categories have no common transfers, \( \phi(R_{+x}) \cap \phi(R_{-x}) = \emptyset \).

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. Eventually, 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 recursively carries out the fission of categories and yields all full simultaneities without repetitions.

There is a further optimization to be considered. Take a category. A full simultaneity must contain no transfer from that category's excluder in order to be covered by that category. In addition, since the full simultaneity is full, it is in congestion with all transfers that it does not contain. Obviously any full simultaneity covered by some category must congest with each member of that category's excluder. Therefore, transfers congesting with the transfers of the excluder must be available in the depot of the category. If the excluder contains at least one transfer, for which the depot has no congesting transfer, then this category is blank. The includer of a blank category, cannot be further extended by the transfers of the depot to a simultaneity which is full (and congests with every remaining transfer of the excluder). The coverage of a blank category is therefore empty and there is no need to pursue its fission.

Let us now instead of retrieving all full simultaneities retrieve all full teams (i.e. those full simultaneities, which ensure the utilization of all bottleneck links).

A category within \( X \) is idle if its includer and its depot together don't use all bottlenecks of \( X \). This means that we can not grow the current simultaneity (i.e. the includer of the category) into a full simultaneity, which will use all bottlenecks. The coverage of an idle category does therefore not contain a full simultaneity, which is a team. Idle categories allow us to prune the search tree.

Carrying out successive fissions, starting from the prim-category and continuously removing all the blank and idle categories ultimately leads to all full teams.

### 5. Speeding Up the Search for Full Teams

This section presents an additional method for speeding up the search for all full teams \( \mathcal{F}(X) \) 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

---
1. The category's excluder, according to the fission algorithm, keeps no transfer congesting with the includer.
transfers the skeleton of $X$. We denote the skeleton of $X$ as $\zeta(X)$. Obviously, $\zeta(X) \subset X$.

Fig. 5 shows the relative size of skeletons compared with the size of the corresponding traffic, for 362 different traffic patterns within the T1 32 node cluster computer (see Fig. 10, in section 7). The skeleton sizes are on average 31.5% of the corresponding traffic sizes.

![The skeleton content of traffic](image)

Fig. 5. Proportion of the number of transfers within a skeleton, compared with the number of transfers of the corresponding traffic.

When 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.

We may first obtain all full teams of the traffic’s skeleton by iteratively applying the fission algorithm and by 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 $\mathcal{I}(X)$ by carrying out the following steps:

1. Obtain the set of the skeleton’s full teams $\mathcal{I}(\zeta(X))$ 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;
   2.2. Initializing the excluder as empty;
   2.3. 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 processing time and at the same time we obtain all full teams of the original traffic without repetitions.

We measured the reduction in search space according to the different search space reduction methods we propose. We consider 23 different traffic patterns within the T1 cluster computer (see section 7). The search space is given by the number of categories that are being iteratively traversed by the fission algorithm. Fig. 6 shows the obtained search space reductions compared with a naive algorithm that would build full teams according to a coverage partitioning strategy, i.e. by constructing categories thanks to the fission algorithm, but without any of the proposed optimizations.

![Search space reduction](image)

Fig. 6. Search space reduction obtained by idle+skeleton+blank optimization steps.

The skeleton algorithm together with the idle and blank optimizations reduces on average the search space to 10.6%, i.e. full teams are computed 9.43 times faster than without search space reduction techniques. Note that in the above comparison even the naive algorithm is smart enough to avoid repeatedly exploring the full simultaneities.

6. Construction of liquid schedules

Having the capability of building full teams, this section presents the general method for building liquid schedules on irregular topologies for any collective communication pattern. Note that we neglect network latencies, consider a constant packet size and assume static routing.
Let us introduce the definition of a schedule. By defining a partition of $X$ as a disjoint collection of non-empty subsets of $X$ whose union is $X$ [Halmos74], a schedule $\alpha$ of a traffic $X$ is a collection of simultaneities of $X$ partitioning the traffic $X$. An elements of a schedule $\alpha$ is called time frame. The length $\#(\alpha)$ of a schedule $\alpha$ is the number of time frames in $\alpha$. 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 $1$ of the traffic, then the schedule is liquid, i.e. a schedule $\alpha$ of a traffic $X$ is liquid if $\#(\alpha) = \Lambda(X)$.

The necessary and sufficient condition for the liquidity of a schedule is that all bottlenecks be used by each time frame 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 $\alpha$ on $X$ is that each time frame of $\alpha$ be a team of $X$.

A liquid schedule is optimal, but the inverse is not always true, meaning that a traffic may not have a liquid schedule. Fig. 8 shows a simple traffic with three bottleneck links. Since there is no schedule whose time frames keep all bottleneck links all the time busy, this traffic has no team and therefore no liquid schedule.

### 6.1. Liquid schedule naive search algorithm

We first propose a simple technique for the construction of a liquid schedule and then introduce an optimization improving the efficiency of liquid schedule construction.

Our strategy for finding a liquid schedule relies on partitioning the traffic into a set of teams forming the sequence of time frames. Associate to the traffic $X$ all its possible teams $A_1,A_2,\ldots,A_n$ which could be selected as the schedule’s first time frame. $X - A_1, X - A_2, \ldots$ is the variety of possible subtraffics remaining after the choice of the first time frame. Each of the possible subtraffics $X_i$ remaining after the selection of the first time frame has its own set of possibilities for the second time frame $\mathcal{N}(X_i) = \{A_{i,1}, A_{i,2}, A_{i,3}, \ldots\}$. The choice of the second team for the second time frame yields a further reduced sub-traffic (see Fig. 9).

Dead ends are possible if there are no choice for the next time frame, i.e. no team of the original traffic may be formed from the transfers of the reduced traffic. A dead end situation may occur, for example, when the remaining sub-traffic appears to be like the one shown in Fig. 8. Once a dead end is faced, backtracking occurs.

The construction recursively advances and backtracks until a valid liquid schedule is formed. A valid liquid schedule is obtained, when the transfers remaining in the reduced traffic form one single team for the last time frame of the liquid schedule.

We use the search tree shown in Fig. 9 and assume that at any stage the choice $\mathcal{N}(X_{\text{sub}})$ for the next time frame is among the set of the original traffic’s teams $\mathcal{S}(X)$, i.e. $\mathcal{N}(X_{\text{sub}}) = \{A \in \mathcal{S}(X) | A \subset X_{\text{sub}}\}$. In the next sub sections we reduce the search space by considering newly emerging bottlenecks at successive time frames.

---

1. The duration of a traffic $X$ is the load of its bottlenecks.
6.2. Search space reduction by considering newly emerging bottlenecks

We observe in Fig. 7 that when we step from one time frame to the next, additional new bottleneck links emerge, e.g. from time frame 3 on, links $l_{i3}$ and $l_{p3}$ appear as new bottlenecks.

In the construction strategy presented in the previous subsection we considered as a possible time frame any team of the original traffic $X$ that can be built from the transfers of the reduced subtraffic. We have shown in our previous publications that for the liquidity of a schedule, it is necessary for each time frame to be not only a team of the original traffic but also a team of the reduced subtraffic.

If $A$ is a liquid schedule on $X$ and $A$ is a time frame of $\alpha$, then $\alpha - \{A\}$ is a liquid schedule on $X - A$.

Thus a liquid schedule may not contain a time frame which is a team of the original traffic but is not a team of a subtraffic obtained by removing some of the other time frames. Therefore we can 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 subtraffic.

Since the reduced subtraffic contains additional bottleneck links, there are less teams in the reduced subtraffic than teams remaining from the original traffic.

By considering in each time frame all occurring bottlenecks, we considerably reduce the search space without affecting the solution space, i.e.

$$\mathcal{N}(X_{sub}) = \mathcal{N}(X_{sub})$$

6.3. Liquid schedule construction optimization by considering only full teams

We can build a liquid schedule by limiting the choice of teams of the reduced subtraffic to its full teams.

Let us modify a liquid schedule so as to convert one of its teams into a full team. We assume that a traffic $X$ has a liquid schedule $\alpha$. Let $A$ be a time frame of $\alpha$. If $A$ is not a full team of $X$, then, by moving the necessary transfers from other time frames of $\alpha$, we can convert the team $A$ to a full team. Evidently, the properties of liquidity (partitioning, simultaneousness and length) of $\alpha$ will not be affected.

Therefore if a liquid schedule is built by a choice of a non full team $\tilde{A}$ of $X_{sub}$ at any stage of construction, then the liquid schedule could have also been built by a choice of a full team $A$ of $X_{sub}$, such that $\tilde{A} \subset A$. Therefore the choice of the teams in the construction may be narrowed from the set of all teams to the set of full teams only, i.e.

$$\mathcal{N}(X_{sub}) = \mathcal{N}(X_{sub})$$

The expression below summarizes the search space reduction by building liquid schedule using full teams of the reduced traffic.

$$\{A \in \mathcal{N}(X) | A \subset X_{sub}\} \subseteq \mathcal{N}(X_{sub}) \subseteq \mathcal{N}(X_{sub})$$

7. Experimental verification

As basic network topology for our testbed, we use the Swiss-T1 cluster (called T1, see Fig. 10). The network of
the T1 forms a K-ring [Kuonen99] and has a static routing
scheme. The throughputs of all links are identical and equal
to 86MB/s [Horst95]. The cluster consists of 32 nodes,
each one comprising 2 processors, i.e. 64 processors,
[SWISSTX99], [Gruber00].

Fig. 10. Architecture of the T1 cluster computer
interconnected by a high performance wormhole
switch fabric.

The sample traffic patterns are selected from different
configurations of half-to-half collective data exchanges
between a set of sending and a set of receiving processors,
where each sending processor carries out a transmission to
each receiving processor. We identified for T1 architecture
362 different collective communication patterns. The
details of the choice of communication patterns are
presented in our previous publications.

The 362 different traffic patterns were scheduled both
by our liquid scheduling algorithms and according to
topology-unaware round-robin schedule. Overall
throughput results for each method are measured and
presented in Fig. 11 for comparison. The values of the
theoretical liquid throughput are also given.

Fig. 11. Theoretical liquid throughputs and measured
throughputs for traffics scheduled according to
round-robin and liquid schedules.

Each black dot represents the median of 7 overall
throughput measurements carried out according to liquid
schedules. Processor to processor transfers have a size of
5MB. The measured aggregate throughputs (black dots) are
very close to the theoretically expected values of the liquid
throughput (light gray area). For many topologies, liquid
scheduling allows to increase the aggregate throughput by
a factor of two compared with topology-unaware round-
robin scheduling (dark gray dots).

Thanks to the presented search space reduction
algorithms, the computation time of a liquid schedule takes
for more than 97% of the considered topologies less than
1/10 of a second on one Compaq Alpha 500MHz
computer.

8. Conclusions

In high performance networks based on cut-through
wormhole switch fabrics or on wavelength division
multiplexing optical networks, significant performance
drops may be observed due to congestions between
transfers sharing common resources. We propose a method
for scheduling collective communications which avoids
congestions. The proposed scheduling method yields an
aggregate throughput equal to the network’s theoretical
upper limit, i.e. its liquid throughput. Efficient computation
of the liquid schedule is achieved by breaking the overall
traffic request into time frames within which all the
transfers of the traffic are allocated. To ensure a liquid
schedule, the time frames must incorporate as many
bottleneck links as possible and utilize all bottleneck links. In
order to compute the liquid schedule we propose a method
for traversing efficiently and without redundancy all
candidate subsets of simultaneous transfers.

We obtained a considerable speed up in the construction
of liquid schedules by carrying out the following
optimizations.

Full teams are enumerated by partitioning the solution
space using inclusion and exclusion constraints. The blank
optimization identifies empty partitions, which do not need
to be further evaluated. The idle optimization identifies
partitions containing no full teams, which do not need to be
further evaluated. The skeleton optimization speeds up the
retrieval of full teams, first by considering only the
transfers necessary to keep all bottleneck links busy and
then by adding up non-congesting transfers.

We construct liquid schedules by partitioning the traffic
into teams. The construction of the liquid schedule is
accelerated by limiting at each time frame the choice to
teams, which use also the newly emerging bottleneck links, i.e. teams of the reduced traffic. Choosing only full teams of the reduced traffic further speeds up the construction of the liquid schedule.

Measurements on the traffic carried out on various sub-topologies of the Swiss T1 cluster computer have shown that for most of the sub-topologies we are able to increase the collective communication throughput by a factor between 1.5 and 2. In congestion sensible coarse-grain transmission networks, i.e. wireless networks, wormhole or lightpath switching networks, liquid scheduling may considerably improve the utilization of transmission resources such as communication links, wavelengths and orthogonal frequency spectra. Liquid schedules avoid congestions and minimize the overall transmission time for collective communications.

In the future, we intend to develop multipath routing solutions, which increase the traffic’s fault-tolerance against link failures and at the same time keep the throughput liquid.

References

[Gruber00] Ralf Gruber, “Commodity computing results from the Swiss-Tx project Swiss-Tx Team”, Grid Computing Meeting, 2000
[Quadrics] www.quadrics.com