Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer

Hans-Wolfgang Loidl[1] and Kevin Hammond[2]

[1] Department of Computing Science, University of Glasgow, Scotland, U.K.
E-mail: `hwloidl@dcs.glasgow.ac.uk`
[2] Division of Computer Science, University of St. Andrews, Scotland, U.K.
E-mail: `kh@dcs.st-and.ac.uk`

Abstract. This paper studies critical runtime-system issues encountered when packing data for transmission in a lazy, parallel graph reduction system. In particular, we aim to answer two questions:

- *How much graph should go into a packet?*
- *How aggressively should a processor look for work after requesting remote data?*

In order to answer the first question, we compare various *packing schemes*, of which one extreme packs just the node that is demanded ("incremental fetching"), and the other packs all the graph that is reachable from that node ("bulk fetching"). The second question is addressed by considering various mechanisms for latency hiding during communication, ranging from fully synchronous communication with no attempt to mask latency, to full thread migration during asynchronous communication.
In order to make our results as general as possible, we have used the GranSim simulator to study a wide variety of parallel machine configurations. Based on these measurements we propose concrete improvements for parallel graph reducers such as the GUM implementation of Glasgow Parallel Haskell.

1 Introduction

Our motto for parallel functional programming is: the more implicit the better. Thus, it is our goal to hide most of the details of a parallel system in the runtime system. This enables the programmer to work on a more abstract level by only annotating potential parallelism in the program and to ignore machine-specific details. However, in order to achieve good parallel performance the runtime system must be able to perform efficient communication without additional hints by the programmer. Since we wish to achieve good performance on a variety of architectures, it is essential to consider communication schemes that can take advantage of the available bandwidth in order to mask network latency for slow networks, as well as schemes aimed at low latency networks.

In this paper we study the performance of parallel algorithms when different schemes are used to pack graph into a communication packet (*packing schemes*) and when different schemes are used to hide latency costs during communication (*rescheduling schemes*). We show that it is important to study both kinds of schemes in order to achieve high performance.

In order to measure the impact of packing and rescheduling schemes we use a set of non-trivial Glasgow Parallel Haskell (GPH) programs. As these programs exhibit fairly complex communication patterns the results obtained from these measurements should also be applicable for the large-scale parallel programs that we are currently working on.

We have chosen to use the *GranSim* [HLP95, Loi96] simulator as the testbed for our measurements. This gives us a high degree of flexibility in simulating a wide range of parallel machines. GranSim has been designed to yield an accurate simulation for machines with very different communication characteristics using state-of-the art compilation technology. The visualisation tools accompanying GranSim have also proved extremely useful both for monitoring performance and detecting possible performance bottlenecks.

To date we have not repeated our experiments on the actual GUM implementation. There are two main reasons for this:

- such results can only confirm individual points on the parallel machine spectrum, rather than explore the entire design space;
- a major objective for GranSim is to drive the implementation priorities for GUM, in order to save implementation effort — the results reported here reflect part of the study that is needed to achieve that priority setting.

In due course, we do expect to report comparisons for those techniques that do prove worth implementing, however.

The rest of the paper is structured as follows: Section 2 discusses principles of parallel graph reduction. Section 3 introduces the notions of packing and rescheduling schemes, discussing advantages and disadvantages of individual schemes. Section 4 compares the efficiency of different packing schemes. Section 5 compares the efficiency of different rescheduling schemes. Section 6 discusses related work. Finally, Section 7 draws conclusions from the measurements and discusses future work.

2 Parallel Graph Reduction

We use Glasgow Parallel Haskell (GPH), a parallel version of Haskell [PH+97]. GPH specifies parallelism using `seq` and `par` combinators, which act as sequential and parallel composition, respectively (see [THL+96] for details). Our computation model is one of parallel graph reduction [Pey89]. This section discusses the most important aspects of the model we are using.

Sparks. In GPH expressions that should be evaluated in parallel have to be marked with a `par` combinator. Currently these annotations must be added by the programmer. The construct

```
x `par` e
```

creates a *spark* for evaluating `x` and then proceeds with the evaluation of `e`. The result of the whole construct is the result of `e`. Typically `x` is a variable that occurs in the expression `e`. A spark represents potential parallelism and basically consists of a pointer to a *thunk* (an unevaluated closure). All sparks are put into a distributed spark pool. When a processor is idle the runtime-system tries to get a spark from the local spark pool. If successful, it turns the spark into a *thread*. If there is no local spark available, however, the system then tries to obtain work from a remote processor. An important design decision, which is examined in this paper, is how aggressive to be when looking for remote work while other threads are blocked waiting for remote data.

Evaluate-and-die. The specific variant of parallel graph reduction we are using is an *evaluate-and-die* model of computation [PCS89]. In this model a parallel thread simply picks up a spark, reduces a thunk to normal form and updates it with its result. No explicit notification of the parent process is necessary. Synchronisation between threads will only occur when one thread tries to access a value that is under evaluation by another thread. In this case the thread is put into a blocking queue that is attached to the thunk being evaluated. When overwriting the thunk with its result the threads in the blocking queue are reawakened.

It is important to note that the runtime-system is free to discard sparks. This means that one parallel thread can subsume potentially parallel computations for which sparks have been created. Therefore, this mechanism is able to automatically increase the granularity in a system with many sparks. This behaviour is similar to a lazy task creation mechanism [MKH90] with sparks taking the role of lazy futures.

Communication. Our communication mechanism is described in detail in the following section. Two basic design decisions are worth mentioning at this point, however: in order to avoid duplicating reduction work, we preserve all graph sharing; and we always move data to the processors running the threads that need it rather than moving threads to the processor where the data happens to reside.

A consequence of the latter decision is that thunks are evaluated on the processor that needs the result of the thunk. The alternative approach of offloading the thunk to the processor that holds the data it needs would improve data locality, but might increase communication costs if the thunk were small but the result large. This decision would repay further study.

It is worth noting in passing that because thunks are just normal graph nodes, the same communication method is used to transfer both data and work, and even for thread migration.

At present we make no attempt to combine consecutive packets sent between the same two processors into a single message. It seems unlikely that this would have a significant effect on performance, however, unless the packet creation overhead was high. It also seems unlikely that our system would generate such patterns of communication except in rare instances.

3 Packing and Rescheduling Schemes

3.1 Packing Schemes

Packing schemes prescribe how much of the graph to transfer to a processor that demands the value of a node or that is given a sparked closure in response to request for work (for convenience we will classify both data transfer mechanisms as *fetching*). Two contrasting packing schemes are *incremental fetching* and *bulk fetching*.

For incremental fetching, only the closure that is requested is sent to the processor that demanded it. This scheme aims to minimise the total number of closures that are sent during the execution of the program ("lazy fetching"). The fixed overhead for this scheme is small because exactly one closure is sent for every fetch request. The cost is that many messages will need to be sent if large parts of the graph need to be shared across processors, and this may prove to be inefficient in a high-latency system.

In contrast, bulk fetching transfers a group of related nodes in a single packet ("eager fetching"). The per-packet overhead is higher because packet construction/deconstruction is much more complicated. The gain is in reduced latency, because many nodes will be transferred in a single packet, and so will not need to be transferred individually if they are needed.

In our implementation of bulk fetching, packets are constructed by traversing the graph to be transmitted in a breadth-first fashion. Each closure is given a global address which is used to preserve sharing both across the system and within the packet. When packing a thunk the original closure is overwritten with a *FetchMe* closure, which acts as a global indirection to remote data. Communication of data is triggered if a thread enters such a *FetchMe* closure during evaluation. Unlike thunks, normal form closures are simply copied. See [THM+96] for a more detailed discussion of this communication mechanism.

In this paper we are interested in refinements of the basic bulk fetching scheme that limit the amount of the graph being sent. We investigate two possibilities: specifying a bound on the packet size or specifying a bound on the number of thunks that can be packed into a single packet. If neither limit is specified, all the graph that is reachable from the requested node will be packed into the packet. Note that packing multiple thunks into one packet essentially amounts to eager work distribution, which may consequently lead to an uneven distribution of sparks. For reference, the GUM implementation [THM+96] currently uses a full-subgraph packing scheme but imposes a limit on the packet size.

Figure 1 depicts the bulk fetching mechanism in action on a simple graph that involves sharing. The left hand side shows the graph before packing takes place, the right hand side shows the graph as it has been updated following packing. The centre of the diagram shows the packet that is constructed to transmit the graph. Shading is used to depict thunks, normal form closures are left unshaded. Note that thunks are replaced by *FetchMe* closures (lightly shaded) in the packed graph whereas normal form closures are left unchanged (and so duplicated on

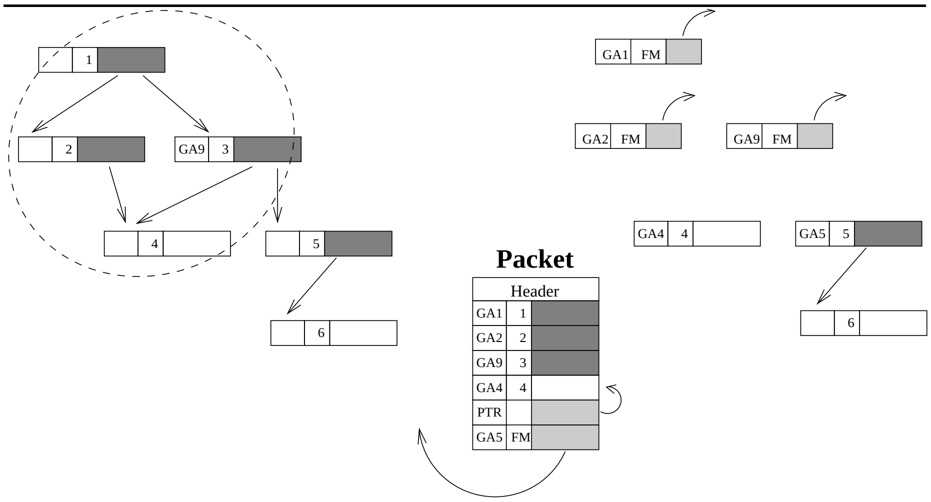

Fig. 1. The bulk fetching mechanism (with 3 thunks per packet)

the remote processor). Since this example shows a packing scheme that packs a maximum of 3 thunks into a packet, one thunk is left behind on the original processor (referenced by a *FetchMe* in the packet).

As closures are added to the packet, they are assigned global addresses (*GAs*) if they were previously held entirely locally. Otherwise their existing *GA* is used. Sharing is preserved within the packet by using special small *PTR* indirection nodes.

3.2 Rescheduling Schemes

Rescheduling schemes prescribe what the processor should do while communication is taking place. The two basic choices are between *synchronous* and *asynchronous* communication.

With synchronous communication, the processor will be idle until it receives a response to its fetch request. If asynchronous communication is used, however, another thread can be executed in the interim. This amounts to latency hiding, since useful work can be performed until the requested data arrives. In high latency systems latency hiding is usually advantageous, effectively reducing processor idle time. However, in low latency systems the overhead attached to descheduling one thread and scheduling another may well be greater than the wasted time, and synchronous communication should be used. Some recent approaches have attempted to use lightweight threads and active messages to mask latency [Ost93, Cha97]. Such an approach may be beneficial for high latency systems (primarily by allowing multiple data fetch messages to be transmitted as soon as a closure that needed several remote closures was activated), but would require significant changes to our implementation in order to support the notification model.

A refinement of the asynchronous rescheduling scheme is to specify how aggressive the system should be in acquiring new work while waiting for the answer of a fetch request. We compare four rescheduling schemes:

1. only execute another runnable thread;
2. turn a spark into a thread if no runnable threads are available;
3. try to acquire a remote spark if the processor has no local sparks;
4. try to migrate another runnable thread if no remote sparks can be found.

These schemes are cumulative, so that thread migration will only be attempted if the three previous schemes have failed, etc. Note that the third and fourth 'global' rescheduling schemes will involve communication in order to obtain new work. In particular, the fourth scheme may introduce gratuitous thread migration towards the end of the computation, when the system load is low.

The rescheduling scheme is only relevant when there are outstanding fetch requests to process. If the processor is idle, it will attempt to use all four schemes to obtain work regardless of the rescheduling scheme in force.

4 An Evaluation of Packing Schemes

In order to evaluate the quality of the different packing and rescheduling schemes it is important to use realistic programs. In particular, it is not likely that simple programs with simple communication patterns will yield meaningful performance measurements for the schemes considered here. We therefore use the following set of example programs:

- `coins` (ca. 30 lines), a divide-and-conquer program that, given a set of coins and a value, will compute the number of ways this value can be paid using those coins;
- `linsolv` (ca. 500 lines), a linear system solver based on a multiple homomorphic images approach, which is typical for many algorithms in symbolic computation;
- `determinant` (ca. 100 lines), a determinant computation, which is the main component of the above linear system solver (both `linsolv` and `determinant` have a divide-and-conquer structure);
- `cfd` (ca. 800 lines), a computational fluid dynamics program, exhibiting an iterative transformation structure typical for many applications in numerical computation.

The principal data structure for the `coins` program is a list of integers, for `linsolv` and `determinant` is a list of lists of integers, and for `cfd` is an array of floating point values. All of these programs, in particular `cfd`, perform a large amount of communication involving these data structures.

Our test programs are already quite efficient. The purpose of the measurements reported here is to show the extent to which a *tuned* parallel algorithm

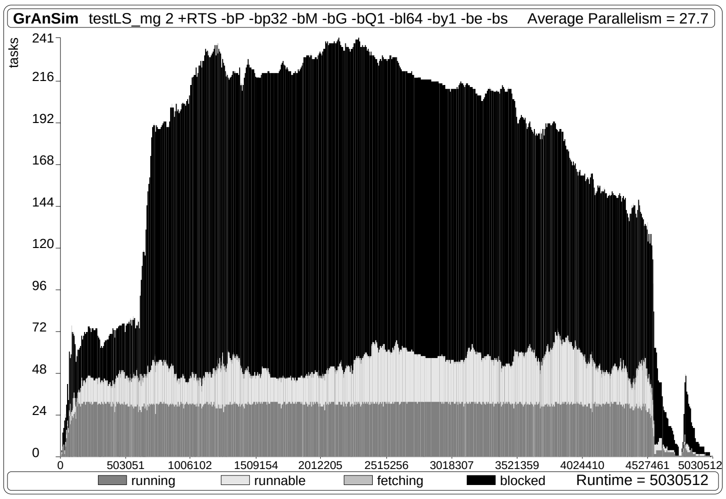

Fig. 2. Activity profile of `linsolv`

can be improved using runtime-system techniques, rather than to show maximal improvements. As an example, Figure 2 shows the overall activity profile for the **`linsolv`** example program running on 32 processors. Note that the number of running threads is almost always at the maximum (32 in this case). Only at the end does the CPU utilisation drop before showing a final peak of parallel threads. This indicates that there is a minor sequential bottleneck in the final combination stage of this otherwise very efficient divide-and-conquer algorithm.

4.1 Bulk versus Incremental Fetching

First we compare the performance of basic bulk fetching (as used in GUM) with incremental fetching (as used in GRIP [PCSH87]). In order to obtain results that are applicable for a wide range of machines we perform each of the measurements for a number of widely varying latencies.

In order to allow comparisons to be drawn for several different cases, each graph plots relative speedup over a sequential program for a range of latencies. Time is measured in terms of abstract machine cycles. This allows low-level details such as the clock frequency of the processor to be abstracted.

Since small differences are more important for lower latencies, the use of a logarithmic scale allows these differences to be more easily seen. The graphs cover the spectrum of parallel machines from shared-memory machines (about 10–20 cycles) through fast distributed-memory machines like GRIP (about 400 cycles) to workstation networks (between 50,000–100,000 cycles). No attempt is made to model network topology, however, and we also assume that sufficient bandwidth

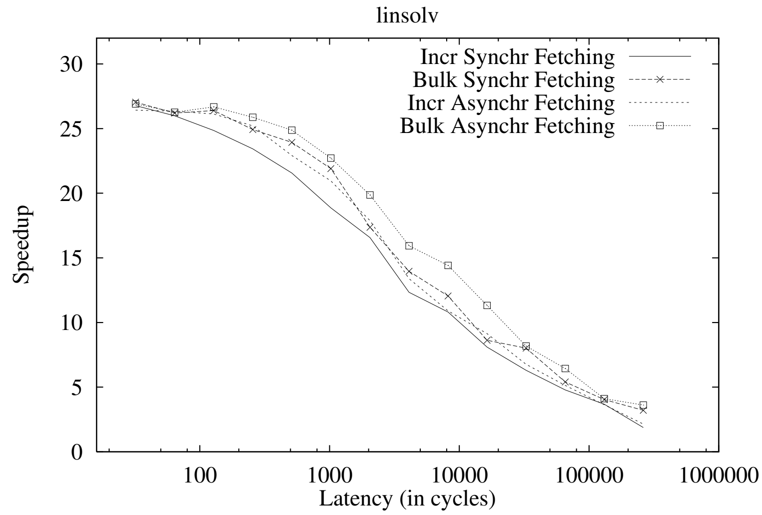

Fig. 3. Bulk vs incremental fetching for `linsolv`

is available to avoid communication bottlenecks. The communications latency is therefore effectively constant throughout the computation.

When comparing bulk and incremental fetching, it is also necessary to consider the most extreme rescheduling schemes. The absolute performance of the resulting four combinations is shown in Figure 3 using `linsolv` as our example program. Overall a bulk fetching scheme with asynchronous communication performs best for all latencies, whereas the incremental synchronous fetching scheme is worst. This is not an entirely obvious result: the speculative nature of bulk fetching may result in unnecessary data being transmitted, or in the same thunks being packed repeatedly, and synchronous communication carries less task rescheduling overhead at low latencies.

Figure 4 shows the same relative runtimes for the same four configurations treating incremental synchronous fetching as the base case. Up to latencies of about 100 cycles, all schemes have almost exactly the same performance. For latencies exceeding 500 cycles, however, bulk asynchronous fetching yields a significant improvement over the other schemes. We observe a reduction of the total runtime of 17% and 28% for latencies between 1,000 and 50,000 cycles. For higher latencies the relative runtimes vary significantly since data locality becomes crucially important. Bulk asynchronous fetching is still consistently the best scheme with a runtime improvement of about 50% at a latency of 260,000 cycles.

The relative runtime graph for `coins` on the right hand side of Figure 4 shows a similar overall picture with the bulk fetching schemes performing best. However, this graph shows a problem that can occur for incremental asynchronous

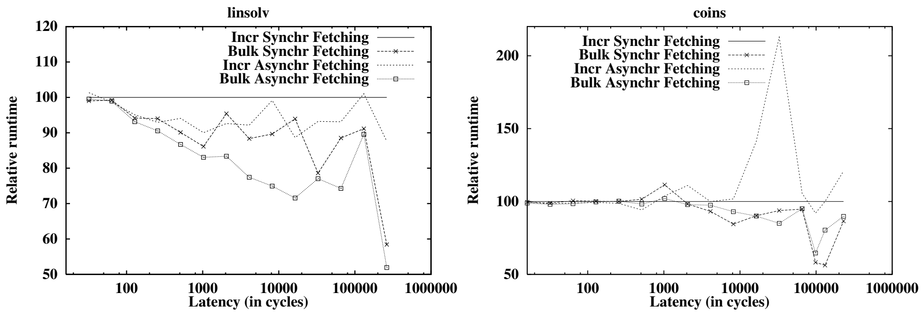

Fig. 4. Bulk vs incremental fetching for `linsolv` and `coins`

fetching: performance can drop drastically if many threads simultaneously compete for the same remote data ("data stealing"). Normally, the communication cost is within a factor of 2 of that for incremental synchronous communication but it exceeds a factor of 3 for the spike shown in the graph. Despite this problem with incremental fetching, there is, somewhat surprisingly, no evidence that data stealing has affected any of the bulk fetching results. This is perhaps because bulk fetching creates duplicates of those normal form closures that are likely to be required by several tasks.

4.2 Bounded Packet Size

One way to reduce communication costs is to impose a limit on the size of graph that is transmitted in response to a fetch request. The graph is packed in a breadth-first manner, with the fringe of the graph replaced by *FetchMe* closures if the packet is too small for the complete graph. Measurements taken from the GUM system [THM+96], which uses a similar mechanism, show a significant performance improvement as the packet size limit is increased from 16 words up to the default value of 1024 words.

While the average packet size in the GUM measurements was quite large, the average size for the example programs reported here was, in contrast, quite small. Even with full-subgraph packing the average packet size for `coins` is between 3 and 4 closures, for `linsolv` it is between 5 and 15 closures (the largest we recorded), and for `cfd` it is between 4 and 10 closures. The average closure size in these examples is also very close to the minimum size for updatable closures: four words. This explains why limiting the packet size has little effect on performance for our programs.

4.3 Limited Number of Thunks

Another way to throttle the communication is to limit the number of thunks that can be packed in one packet. Since thunks represent potential work, specifying such a limit intuitively increases the resistance against moving work to other processors. In a lazy fetching scheme, except for the root of the graph, only normal form closures are packed. As there is no danger of duplicating work, normal forms can always be copied without destroying data locality. However, this might result in very small packets and hence in a rather high relative overhead for packing the graph.

The other extreme is the scheme used, for example, in GUM: no limit on the number of thunks in a packet. This increases the average packet size but runs the risk of offloading thunks which may be needed later by the current processor. This "*thunk stealing*" effect will cause additional communication if a thunk must subsequently be returned to the originating processor. Although this behaviour has been observed in the GUM system for several programs that communicate heavily, its overall effect on performance has not been measured precisely.

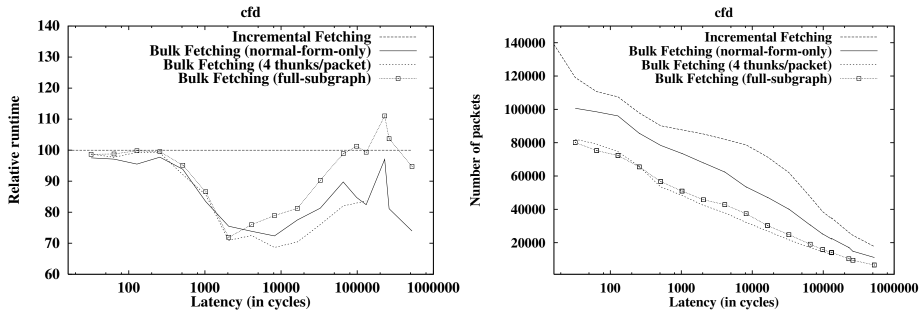

Fig. 5. Relative runtimes and number of packets sent for `cfd`

Figure 5 compares the runtimes for the full-subgraph and normal-form-only packing schemes using bulk fetching, against those for incremental fetching for `cfd`. Interestingly, for most latencies a full-subgraph packing scheme does not give the best performance. For this example program it is better to use a normal-form-only packing scheme. For very high latencies (greater than 100,000 cycles) full-subgraph packing is worse even than incremental fetching, and normal-form-only packing yields a performance improvement of about 20%. The communication graph on the right hand side of Figure 5 shows that full-subgraph packing does not in fact generate the least number of packets, as might be expected. Choosing a limit of 4 thunks per packet actually results in least communication, because it reduces the probability of thunks being stolen while still allowing reasonable work distribution. This compromise between large packet sizes and low probability of thunk stealing shows the best performance for `cfd`.

In Figure 6 we show the relative runtime of the `linsolv` program using packing schemes with different numbers of thunks per packet (left hand side) and the corresponding amount of communication (right hand side). In this case normal-

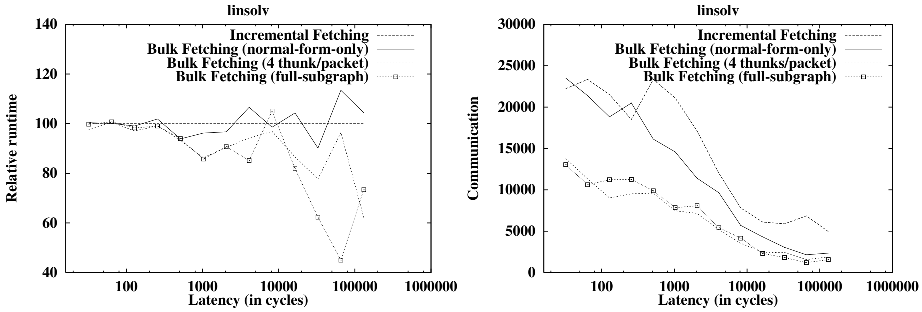

Fig. 6. Relative runtimes and number of packets sent for `linsolv`

form-only packing does not generally achieve an improvement in performance. The amount of communication is close to the incremental fetching scheme and much higher than for the other bulk fetching schemes. However, when increasing the number of thunks per packet to 4 we can surpass the performance of the full-subgraph packing scheme for some latencies. The number of packets sent in this case is consistently below the number of packets sent with a full-subgraph packing scheme. This indicates that thunk stealing is occurring in this program, too. However, it is less harmful than in `cfd`.

5 An Evaluation of Rescheduling Schemes

The different rescheduling schemes introduced in Section 3.2 have a direct influence on the distribution of work and on the locality of data in the system. The first and second ("local") rescheduling schemes only try to access work present on the local processor and are therefore refinements of asynchronous communication. The second rescheduling scheme is slightly more aggressive than the first in that it also creates new work if the processor is idle. Therefore, this scheme is likely to increase the overall workload on the machine and would be advantageous for programs creating a rather small number of threads under the default scheme, but for which there is a potential reserve of parallelism. The third and fourth ("global") rescheduling schemes are most aggressive in attempting to acquire work from other processors while awaiting a response to a message.

These schemes can consequently be seen as an alternative way to distribute the system workload. However, in doing so they also increase the total amount of communication. Since these schemes only make a difference if no local work is available, they are not generally very effective in creating an even workload. They should be seen as a cheap form of load distribution that comes as part of a refined asynchronous communication scheme.

Figure 7 compares the speedups (left hand side) and the relative runtimes (right hand side) for `linsolv` using different rescheduling schemes. The base

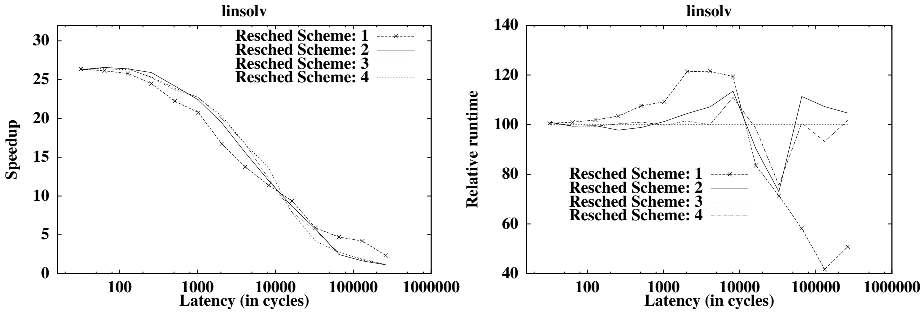

Fig. 7. Speedups and relative runtimes with different rescheduling schemes for `linsolv`

case for the relative runtimes is the third rescheduling scheme, which is the one used in GUM. The packing scheme is fixed to be full-subgraph packing.

The quality of the rescheduling schemes clearly depends on the latency. For low latencies the more aggressive global schemes perform best since there is little cost associated with fetching work from remote processors. The improved load distribution outweighs the increased communication caused by a deteriorated locality of the data. However, for high latencies the dominant cost becomes that of moving data between processors. In this case, data locality is more important than an even load distribution. Therefore, the local rescheduling schemes usually perform better than the more aggressive schemes.

It is also worth noting that the difference between the rescheduling schemes is smaller for full-subgraph packing. This is because there are fewer fetch requests if such a packing scheme is used, and hence the more aggressive rescheduling schemes do not have many opportunities to acquire remote work while waiting for remote data to arrive. The relative runtimes given on the right hand side of Figure 7 show that the runtime for the least aggressive scheme lies between 120% (medium latencies) and 40% (high latencies) of the base case. For latencies higher than about 15,000 cycles this scheme yields a clear improvement in overall runtime.

6 Related Work

The importance of packing schemes on distributed memory machines has already been observed in systems such as Alfalfa [Gol88], where poor data partitioning was cited as the main reason for the poor performance of matrix multiplication and other programs. Goldberg suggested that compiler controlled data partitioning could overcome this problem by ensuring a good initial data distribution. However, our experience with large application programs such as Lolita [MSS94] and the programs studied here shows that parallel symbolic programs operate mainly on large dynamic data structures. This suggests that a more sophisticated dynamic management of the distributed heap is required.

In his PhD thesis, Kesseler [Kes96] examines different "copying strategies" (corresponding to our packing schemes) for the Concurrent Clean system. The main difference from our work is that his schemes have to maintain uniqueness information attached to a graph. Thus, in his case a copying strategy that violates this uniqueness information might change the semantics of the program. This complication does not arise in our work.

Since his lazy normal form copying strategy only copies those non-normal-form closures that are suitably annotated, this is close to our normal-form-only packing scheme. Kesseler also considers data locality and its effect on performance, an area that we have not yet studied in detail.

As already mentioned, the GUM system [THM$^+$96] uses a full-subgraph packing scheme with a default packet size of 1024 words. The rescheduling scheme that is used corresponds very closely to our third scheme: a processor may turn a spark into a thread while waiting for remote data and may try to find a remote spark. GUM does not support thread migration. The GRIP system [PCSH87] uses incremental synchronous communication. This is motivated by the low latency of this machine (around 400 cycles).

In [TD94] Toyn and Dix give an algorithm for the efficient lazy transfer of arbitrary pointer structures. The structure of this algorithm is similar to that of a copying garbage collector and therefore closely related to our graph traversal during packing. The invariants of the algorithm are similar to those in our packing schemes: sharing and cycles have to be preserved during the packing processes. This is achieved by using two kinds of special nodes similar to our *FetchMe* and *PTR* nodes. However, in our model the original graph is updated with *FetchMe* closures whereas Toyn and Dix's algorithm leave it unchanged. They also only consider a lazy packing scheme without discussing variants for tuning the granularity of the transfer.

In the pHluid [FN96] distributed memory implementation of a shared memory parallel functional language a key issue for achieving good performance is to avoid distributing work unnecessarily. This is achieved by handling a queue of stealable threads in a LIFO manner and stealing work from the old end of the queue (similar to the mechanism used in the HDG machine [KLB91]).

The pHluid heap is partitioned into pages that are owned by processors. These pages act as units for communication. A simple cache coherence algorithm without invalidation is used to manage communication. Thus, in this model no thunk stealing will happen because the result of an evaluation is always sent to the original processor. This enforces good locality of data in the long run but may cause more communication than our model if many updates need to be performed on a remote page. Overall, the packing and rescheduling schemes that correspond most closely to the pHluid model are bounded packet size packing and aggressive asynchronous rescheduling.

The abstract TAM dataflow machine [CGS93] also emphasises latency tolerance by using asynchronous communication, implemented in this case via split-phase memory access operations. As in the work by Ostheimer [Ost93] and Chakravarty [Cha97], TAM uses multi-threading and active messages. Performance thus depends on the source program providing large amounts of multi-threaded parallelism. Only data that is needed immediately is transferred, similarly to our incremental fetching scheme.

7 Conclusion

We have studied communication issues in a parallel graph reduction system, covering machines with widely varying latencies. We have proposed methods for limiting the amount of communication (*packing schemes*) and for distributing work during the idle time caused by communication (*rescheduling schemes*).

The results in this paper are representative for our measurements of all possible combinations of packing and rescheduling schemes on a set of non-trivial programs that exhibit a significant amount of communication using fairly complex communication patterns. The structure of the parallelism in and the communication behaviour shown by these programs lead us to believe that these results can be applied in the ongoing parallelisation of very large programs. The programs that we are currently studying include a program that determines accident blackspots based on a large database of traffic accident reports [WH96] and Lolita [MSS94], a large scale natural language engineering system (about 60,000 lines of Haskell).

From our measurements we draw the following conclusions:

– *Rescheduling schemes:* For low latencies, where an even load distribution is more important than good locality of data, aggressive rescheduling schemes deliver good work distribution and therefore good performance. For high latencies, however, the improved load distribution does not compensate for reduced data locality. The crossover point usually lies between 15,000 and 30,000 cycles, i.e. loosely-coupled multiprocessors.
– *Packing schemes:* In general, full-subgraph packing proves to be the best packing scheme. In practice, there is little danger that such a packing scheme will cause a disastrously uneven load distribution.
– *Thunk stealing:* Occasionally the full-subgraph packing scheme causes *thunk stealing*: the gratuitous offloading of thunks that will be needed later. This increases communication costs and hence reduces performance. We believe that thunk stealing is the reason for full-subgraph packing sometimes being worse than those schemes that pack a limited number of thunks per packet (e.g. `cfd`). This does not happen very frequently, however.
– *Bulk vs incremental fetching:* For low latencies (up to about 100 cycles) there is no difference in the performance of bulk and incremental fetching. Especially for very high latencies (more than about 50,000 cycles) bulk fetching achieves significant runtime improvements compared to incremental fetching even when using asynchronous communication for latency hiding.
– *Bounded packet size:* The average packet size is in general very small, even for full-subgraph packing (in our example programs it is almost always smaller than 15 closures). Therefore, changing the packet size, as has been previously

suggested for improving communication performance, has hardly any effect on the runtime of the program.

From these observations taken from our simulation results, we can make the following suggestions for improving the runtime-system of a parallel graph reduction machine such as GUM:

– Choosing a small packet size is not an effective means of tuning the granularity of the communication. This is due to the small average number of closures per packet in most programs.
– For programs with a high degree of communication a *normal-form-only packing* scheme should be used in order to reduce the amount of thunk stealing. It is probably not worthwhile implementing a more general scheme that allows the user to specify the number of thunks per packet because good values for such a parameter are very hard to predict.
– When running on a high-latency system (more than about 15,000 cycles) a *less aggressive rescheduling scheme* should be used in order to maintain good data locality. Our recent measurements with Lolita seem to underline the importance of data locality in large application programs. Currently, GUM uses a global rescheduling scheme similar to our third rescheduling scheme.

As further work we plan to quantify the impact of thunk stealing on the performance of larger parallel programs exhibiting a less regular structure of parallelism. Our initial results indicate that limiting the number of thunks per packet is an effective way of avoiding thunk stealing. Another aspect we would like to investigate is the impact of our rescheduling schemes on the total heap consumption of the parallel program. For large programs a too aggressive rescheduling scheme might unnecessarily create threads that hold on to a large partition of the heap increasing the total heap residency. It would also be interesting to compare our rescheduling schemes, which influence the load distribution, with other work on load balancing.

References

[Cha97] M. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD Thesis, Technical Univ. of Berlin, Feb. 1997.

[CGS93] D.E. Culler, S.C. Goldstein, K.E. Schauser, T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, **18**:347–370, Jun. 1993.

[FN96] C. Flanagan and R.S. Nikhil. pHluid: The Design of a Parallel Functional Language Implementation on Workstations. In *ICFP'96 — Intl. Conf. on Functional Programming*, pp. 169–179, Philadelphia, PA, May 24–26, 1996. ACM Press.

[Gol88] B. Goldberg. Multiprocessor Execution of Functional Programs. *Intl. Journal of Parallel Programming*, **17(5)**:425–473, Oct. 1988.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — Conf. on High Performance Functional Computing*, pp. 208–221, Denver, CO, Apr. 10–12, 1995.

[Kes96] M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Univ. of Nijmegen, Apr. 1996.

[KLB91] H. Kingdon, D. Lester, and G. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, **34(4)**:290–301, 1991.

[Loi96] H-W. Loidl. *GranSim User's Guide.* Dept. of Computing Science, Univ. of Glasgow. Jul. 1996.

[MKH90] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP'90 — Conf. on Lisp and Functional Programming*, pp. 185–197, Nice, France, Jun. 27–29, 1990.

[MSS94] R.G. Morgan, M.H. Smith, and S. Short. Translation by Meaning and Style in Lolita. In *Intl. BCS Conf. — Machine Translation Ten Years On*, Cranfield Univ., Nov. 1994.

[Ost93] G. Ostheimer. *Parallel Functional Programming for Message-Passing Multiprocessors.* PhD thesis, Univ. of St Andrews, Mar. 1993.

[PCS89] S.L. Peyton Jones, C. Clack, and J. Salkild. High-Performance Parallel Graph Reduction. In *PARLE'89 — Parallel Architectures and Languages Europe*, LNCS 365, pp. 193–206. Springer-Verlag, 1989.

[PCSH87] S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *FPCA'87 — Intl. Conf. on Functional Programming Languages and Computer Architecture*, LNCS 274, pp. 98–112, Portland, OR, Sep. 14–16, 1987. Springer-Verlag.

[Pey89] S.L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, **32(2)**:175–186, Apr. 1989.

[PH+97] J.C. Peterson, K. Hammond (eds.) *et al.* *Haskell 1.4 — A Non-Strict, Purely Functional Language*, Apr. 1997.

[TD94] I. Toyn and A.J. Dix. Efficient Binary Transfer of Pointer Structures. *Software – Practice and Experience*, **24(11)**:1001–1023, Nov. 1994.

[THL+96] P. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones, and J. Wu. A Case Study of Data-intensive Programs in Parallel Haskell. In *Glasgow Workshop on Functional Programming 1996*, Ullapool, Scotland, Jul. 8–10.

[THM+96] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pp. 79–88, Philadelphia, PA, May 1996.

[WH96] J. Wu and L. Harbird. A Functional Database System for Road Accident Analysis. *Advances in Engineering Software*, **26(1)**:29–43, 1996.