Dynamic Chunking in Eden

Jost Berthold

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
`berthold@informatik.uni-marburg.de`

Abstract. Parallel programming generally requires awareness of the granularity and communication requirements of parallel subtasks, since without precaution, the overhead for parameter and result communication may outweigh the gain of parallel processing. While this problem is often solved explicitly at the language level, it can also be alleviated by optimising message passing mechanisms in the runtime environment. We describe how a simple buffering mechanism introduces dynamic list chunking in the runtime environment of the parallel functional language Eden. We discuss design and implementation aspects of dynamic chunking and compare its effects to the original version in a set of measurements. Our optimisation is justified by a simple cost model, measurements analyse the overhead and illustrate the impact of the changed message passing mechanism.

1 Introduction

A major issue in parallel programming is to consider the granularity and communication need of parallel algorithms [6]. Regardless of the underlying language paradigm, communication latency in parallel algorithms may limit the achievable speedup. On the other hand, sending more data at a time can spoil the parallel system's synchronisation and lead to distributed sequential execution. In the field of lazy functional languages, a second obstacle is the conflict between demand-driven evaluation and parallelism [18]. Parallelism control in the coordination language generally has to balance between lazy evaluation and fast parallel startup.

The parallel functional language Eden [3] offers means to define parallel processes and control their execution and granularity explicitly at the language level. As investigated in [9], ingenious programming with respect to the particular language semantics of Eden coordination constructs leads to significantly better speedup, but such optimisations force the programmer to write far from obvious code and thus fail to meet the main intention of the functional paradigm in parallel programming: *"[to] eliminate [...] unpleasant burdens of parallel programming..."* ([8], foreword) by high abstraction. Benchmark programs often use a chunking technique to increase the size of messages between two processes; which we would like to call the *message granularity*, as opposed to the *task* granularity, which refers to the complexity of processes (as a general term, granularity of computation units is reciprocal to their number). However, the data communication of a parallel program is strongly influenced by the particular hardware and network setup. A common issue for benchmarking programs is to first experiment with different granularities in order to balance communication latency against synchronisation lacks, and then hand-tune the explicitly controlled (message) granularity from the experimental pre-results. The hand-tuning of programs involves severe program restructuring which decreases readability and maintainability. Simple lists are e.g. replaced by lists of lists, which requires complex and error-prone conversions. These problems could however be avoided by optimising the message passing mechanism in the runtime environment.

Such an optimisation should be located at a very low level in the communication facilities of the runtime system, thereby making it completely independent of the language semantics. The main idea in the optimisation is to save communication cost by automatically gathering successive messages to the same receiver. Several messages will thus be *dynamically chunked* in one single big message; as opposed to explicit static chunking of the *data* itself in the program's granularity control.

In this paper, we describe the implementation and the effects of this simple buffering mechanism in the runtime environment of the parallel functional language Eden. The paper is organised as follows: After a short introduction to the language Eden and its implementation in Section 2, we describe the aim of the optimisation as well as some design and implementation aspects in Section 3. The effect of our optimisation is described by a simple cost model in Section 4. Finally, we show measurements which analyse the overhead and the impact of the changed message passing mechanism. Section 5 concludes.

2 Parallel processing with Eden

2.1 Language Description

Eden extends Haskell [14] with syntactic constructs for *explicitly* defining processes, providing direct control over process granularity, data distribution and communication topology [3, 10]. Its two main coordination constructs are process abstraction and instantiation.

```
process::(Trans a, Trans b)=> (a -> b) -> Process a b
```

embeds functions of type `a->b` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both types `a` and `b` belong to the type class `Trans` of transmissible values. A *process abstraction* `process (\x -> e)` defines the behavior of a process with parameter `x` as input and expression `e` as output.
A *process instantiation* uses the predefined infix operator

```
( # )::(Trans a,Trans b)=> Process a b -> (a -> b)
```

to provide a process abstraction with actual input parameters. The evaluation of an expression `(process (\ x -> e1)) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* is responsible for evaluating and sending `e2`, while the new *child process* evaluates the expression `e1[x->e2]` and sends the result back to the parent. The (denotational) meaning of the above expression is identical to that of the ordinary function application `((\ x -> e1) e2)`.

Both input and output of a process can be a tuple, in which case one concurrent thread for each output channel is created, so that different values can be produced independently. Whenever one of their outputs is needed in the overall evaluation, the whole process will be instantiated and will evaluate and send all its outputs eagerly. This deviation from lazy evaluation aims at increasing the parallelism degree and at speeding up the distribution of the computation. Local garbage collection detects unnecessary results and stops the evaluating remote threads. In general, Eden processes do not share data among each other and are encapsulated units of computation. All data is communicated eagerly via (internal) channels, avoiding global memory management and data request messages, but possibly duplicating data.

2.2 Stream and List Processing

Data communicated between Eden processes is generally evaluated to normal form by the sender. Lists are communicated as streams, i.e. each element is sent immediately after its evaluation. This special communication property can be utilised to profit from lazy evaluation, namely by using infinite structures and by reusing the output recursively, as e.g. in the *workpool* skeleton [10]. Another obvious effect is the increased responsiveness of remote processes and the interleaving of parameter supply and parallel computation. Processing long lists of data is a prime example for functional parallel programs, e.g. in a simple parallel sorting program:

Example: The following function sorts a list of values in parallel by distributing it to child processes, which sort the sublists using a sequential sorting algorithm. Finally, the sorted sublists are merged together by the parent.

```
parsort :: (Trans a, Ord a) => ([a] -> [a]) -> [a] -> [a]
parsort _ [] = []
parsort seqsort xs = lmerge [(process seqsort) # sublist |
                       sublist <- unshuffleN noPe xs ] ‘using‘ spine
```

The sublists are created by a split function `unshuffleN :: Int -> [a] -> [[a]]` which uses the system value `noPe` to determine the number of available PEs in the parallel setup. The function `lmerge` merges the returned sorted sublists sequentially in a tree-shape manner. The evaluation strategy `spine` [16] is applied in order to start all processes simultaneously as soon as the result is needed. ◁

In the child processes, work is done essentially by comparing several inputs. The Eden sending policy leads to a large number of very small messages between the parent and the sorting processes and slows them down (note that the message passing latency also affects the evaluation in Eden, since values are sent eagerly after evaluation, whereas with lazy communication and global memory, data transmission does not affect the evaluation). If the program does not exploit stream communication, it is favourable to send more data together, ideally without disturbing the interleaving between parameter supply and evaluation.

We could modify the parallel sorting function, so that the sorter processes receive their input in bigger chunks instead of element per element:

Example:(cont.d)

```
parsortchunk :: (Trans a, Ord a) => Int -> ([a] -> [a]) -> [a] -> [a]
parsortchunk size seqsort xs =
      lmerge [ process (seqsort . concat) # (chunk size sublist) |
                     sublist <- unshuffleN noPE xs ] ‘using‘ spine
-- simple list chunking
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk k xs = (take k xs) : chunk k (drop k xs)
```

The chunking function aggregates every `size` elements to a sub-sublist, which is deconstructed by the receiver, so we reduce the number of messages. But this second version is much less intuitive, and it is far from obvious which parameter for `size` would be best. Another, even more obscure variant restructures the parallel sorting algorithm and chunks the output as well:

```
parsortchunk2 size seqsort xs =
      lmerge [ lmerge (process (map seqsort) # (chunk size sublist) |
                     sublist <- unshuffleN noPE xs ] ‘using‘ spine
```

In this version, each child process sorts several smaller lists, and the caller merges both each child’s results *and* the final result. This overhead for the caller is the price for less communication and a much better overlap of parallel evaluation and communication. We cannot tell the best `size` parameter for either variant without excessive tests, but it is clear that both variants perform better by saving communication. ◁

An improvement to this enigmatic optimised code is to use special *skeletons* for specific tasks as e.g. mapping a function to a huge list in parallel. Skeletons are generic patterns of parallelism which take the specific working functions as arguments, as described and discussed for Eden in [10]. Since a skeleton is implemented in a predefined library, it can do chunking implicitly and hidden from the programmer. Programs using skeletons are often easier to read, but skeletons are always restricted to their respective pattern of parallelism. In our example, a `map-fold` skeleton could do the work, but we are still free to spoil the performance by choosing an inappropriate chunk size, unless the skeleton developer has chosen one for us. Anyway, the chunk size would always be statically fixed.

The idea of this paper is to investigate the effects of an automatic chunking mechanism *inside the runtime system* of Eden, i.e. modifying the communication layer to send data messages in a packet. Such a feature in the runtime system apparently makes programming much easier and chooses the right chunking amount automatically, but will of course introduce a considerable overhead.

3 Dynamic Chunking in the Eden Runtime Environment

Eden's implementation extends the Glasgow-Haskell-Compiler (GHC, [13]) by a parallel runtime environment, which is explicitly controlled by a small number of primitive operations. Using these primitives, high-level process coordination is specified in a functional module. The runtime system itself provides means to instantiate new remote processes and to create and use the (now explicit) channels between them. Apart from that, it synchronises computations and controls process termination. The Eden runtime system as a whole has been described in the past (e.g. in [2, 1]) and will thus be omitted in this paper, the Eden message protocol being the only detail of topical interest, together with the more general properties of its message passing mechanisms shared with GUM [17].

3.1 Eden Message Protocol and Its Penalties

Message Protocol Eden processes communicate via 1:1 channels, which are represented by a link from an outport to an inport, structures which the RTS uses to address messages correctly. As a general rule, every message between processes contains these two ports. Eden processes send the following message types:

Msg.-Type	Sender (Port)	Receiver(Port)	[Data]

Create Process instantiates a process at the receiver PE.
Terminate stops a remote thread which sends data to a closed inport.
Value sends a single value as a subgraph in normal form.
Head sends an element (subgraph in normal form) of a list.

In addition, we also have messages to and from the system manager program *SysMan.c*, a stand-alone C + PVM program which controls the startup and shutdown of all PEs. Those messages do not belong to the Eden protocol, but to the system's communication as a whole, since they are sent between the PEs and not between processes.

Ready Announces a PE to *SysMan* (no data)
Task-Ids From *SysMan*. Contains the addresses of all PEs started (in PVM) for the parallel computation.
Finish From *SysMan*: Stops the parallel system. (no data)
From one of the PEs to *SysMan*: initiate system stop.

The message **Create Process** is sent by a thread in the generator (parent) process as an effect of the primitive operation `createProcess#`. The receiver unpacks the included subgraph into its heap and starts a new process by creating a thread to evaluate the subgraph.

Terminate is sent by the runtime system after garbage collection (and not by a process), when the marking of a garbage collection does not reach a synchronisation node which represents data evaluated remotely.

Messages **Value** and **Head** are the interesting ones for the work presented here. They both transmit evaluated data (as single values or as stream data) between processes. The included subgraph in normal form replaces a synchronisation node in the receiver's heap, which is linked to the receiving inport. This is a direct replacement for single values, while for stream data, a new Cons closure is created and its references filled with the received subgraph and a new synchronisation node for the rest of the stream/list.

Simple Cost Analysis Following the concept of stream communication in the language specification, if a child process receives or sends back a very long list, every element is sent in a separate **Head** message. Since data transmission is eager in Eden, the amount of messages is not limited by the demand-driven evaluation (as it would be in GUM, the GpH runtime environment). Sending a message always implies a certain penalty for the required actions in the underlying communication middleware. This penalty has been quantified by using special test primitives in a debug runtime system.

In the test program, we extract the time for all actions directly related to the message passing subsystem by repeatedly linearising a graph structure of variable size and either sending it or not – the difference indicates time spent for sending actions. The test program does not care about receiving those linearised subgraphs, so network latency is not involved. To quantify the influence of data sizes sent, we use a simple linear model, where sending time is estimated as basic time λ for each message plus variable time linearly growing with the message size in words, weighted by a factor β.

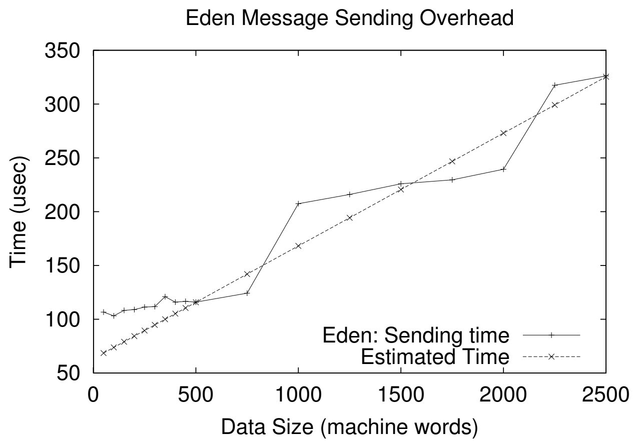


Fig. 1. Message sending penalty measurement and linear model: $time = \lambda + \beta \cdot datasize$

Fig. 1 show results of the measurements and the time estimation obtained by linear regression. The obtained values are $\lambda = 63.34\,\mu sec$ and $\beta = 0.1\,\mu sec$, showing that the amount of data has only a small impact on the time needed to send a message to another PE, compared to the basic sending action itself. We see that the linear model is not completely correct (influence of a step function, due to properties of the underlying message-passing middleware PVM and TCP/IP), but this deviation is not relevant to what we want to show.

3.2 Concept of Dynamic Low-level Chunking

Summing up, dynamic chunking aims at decreasing the defacto number of stream data messages between the PEs automatically by collecting "messages" sent by one process to another one in a send buffer. Messages in the buffer are then sent together in a "packet"[1]. This drastically reduces the amount of packets, while their data size increases. As explained, reducing the number of messages should be transparent to the language design and thus have no effect on language properties. It is obtained by introducing a new low-level layer of communication in the runtime system, whose particular functionality is explained in this section.

Collecting messages in the runtime system needs send buffers of sufficient size in every PE of the parallel system. Their size is at least the maximum size of one message plus additional room for administrative fields. We maintain one send buffer per communication partner, which is every PE in the system. Alternatives would be to have either only one send buffer or one buffer per channel (i.e. sending thread). Both solutions have obvious disadvantages, either in the administration of the buffers or in the achievable effects.

As well as the sender, the receiver of a packet must buffer it for processing to make the change transparent to the next layer of abstraction, the message processing unit of the runtime environment. We would in fact only need one single buffer to receive packets, if we processed the whole packet at once. On the other hand, having an own receive buffer for each PE makes it very easy to implement a fair processing manner, since we can choose between several buffers without losing data. The receive buffers are simply processed in a round robin manner and one message at a time, realising a fair PE communication.

Buffering other messages than the **Head** message would slow down the computation globally by artificial latency, which is absolutely clear for **Create Process** and **Terminate**, but also valid for **Value** messages, since no other message will follow a single value. To prevent deadlock situations (two PEs holding back each other's messages), the scheduler must as well force packets to be sent when there are no runnable threads.

In total, the criteria to send a packet are:

– if the packet contains an urgent message.
– before adding a message, if this message is bigger than the remaining space in the packet.

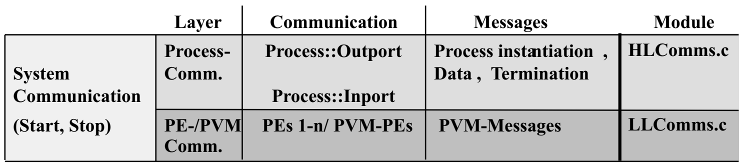
[1] In the following, we refer to "message" and "packet" in the sense that a packet is sent by the MP-System and contains several messages, where (virtually) "sending a message" means to add it to the packet.

- immediately after adding a message, if no other message can fit into the packet any more. The minimum message size in Eden is two ports.
- during scheduling, if the packet age is more than a given timeout value. The maximum age is a runtime system parameter accessible to the user.
- when the whole PE does not have any runnable threads (send all packets)

As well as the specific timeout value for packets (adjustable in milliseconds), information about all actions related to sending packets can be collected for statistical purpose. The methods which decide about sending packets are the place where all information about the buffering mechanism is brought together, e.g. average and maximum packet sizes, no. of timeouts, packets forced etc.

3.3 Implementation Remarks

The existing runtime system for GpH and Eden provides two communication layers (files *HLComms* and *LLComms*), but in the current implementation, this separation only structures the code and differentiates between the high-level message protocol of the virtual machine and the concrete message passing. As depicted in Fig. 2, *HLComms* defines methods to send, receive and process messages conforming to the described message protocol (different for GUM and Eden), while *LLComms* provides basic methods to map these abstract sending and receiving operations to the message passing system (MP-System), currently PVM [15]. So we find a 1:1 relation between (abstract) messages sent by a process and (concrete) messages in the MP-System, which had to be given up for our modification.



	Layer	Communication	Messages	Module
System Communication (Start, Stop)	Process-Comm.	Process::Outport Process::Inport	Process instantiation , Data , Termination	HLComms.c
	PE-/PVM Comm.	PEs 1-n/ PVM-PEs	PVM-Messages	LLComms.c

Fig. 2. Two Layers Model of the Communication modules

The message buffering system is implemented in a changed module *LLComms* which provides a slightly modified interface. All methods in the interface of *LLComms* do not access the MP-System itself any more, but new internal functions. The former behaviour remains unchanged to keep the new layer transparent with respect to communication routines, which are shared with the GUM system. Message buffering could be used for GUM without any changes, but data is only sent on demand in GUM, and GpH does not use any concepts comparable to the stream channel communication in Eden, so there is no need for dynamic chunking in GUM at all. On the contrary, Hammond/Loidl dismiss message buffering for GUM entirely in [11], since each additional latency would definitely lead to a slow-down.

Functionality provided by *HLComms* sends and processes messages according to the Eden protocol and uses the MP-System only by the interface of *LLComms*. Therefore, only a slight modification was necessary to force urgent messages immediately.

Receiving message packets only requires changes in internal methods of *LLComms* in order to work on the receive buffers instead of with the MP-System. Modifications have been made to the receiving routine, to the selector for sender and message type and to the unpacking method. Furthermore, we had to implement a separate method for a blocking receive. *Sending* messages has been discussed above in the concept. As explained, we need an additional method to force packet sending and have to administer the send buffers in the low-level module.

We shall not digress too much on this low-level implementation, but need to say some words about the startup and system messages, which, of course, must be adapted to message buffering, too. The system messages (**Ready, Finish, Task-Ids**) are sent and processed by the same methods as data messages, but the startup messages should be sent and processed *before* the other message buffers are completely set up. Furthermore, message buffering allows us to receive system messages from *SysMan* with priority. We therefore introduced a special handling for system messages to and from *SysMan*.

As already mentioned, the message buffering mechanism introduces a middle-layer into the communication subsystem of GUM and Eden. This also has an architectural aspect: Fig. 3 shows the modified communication system, which now abstracts from the concrete underlying message-passing system (currently PVM[2]). Internal functions inside *LLComms* still use PVM, but they have been moved away from the interface and concentrated in the internals. As they are completely independent of the GHC runtime system as a whole, this section can easily be placed in an additional module, thereby facilitating the port to other MP-systems.

	Layer	Communication	Messages	Module
System-Communication (Start, Stop)	Process-Comm.	Process::Outport ↓↑ Process::Inport	Process instantiation, Data, Termination	HLComms.c
	PE-Comm.	(virtual) PEs 1-n	Eden-Packets	LLComms.c
PVM-Comm.		(virtual/real) PEs (pvm_tids)	Simple data (C-Integer)	LLComms.c

Fig. 3. New Three-Layer-Model of Communication in the runtime system

The dependencies in every other file than *LLComms* have been eliminated, so that the layer concept in the shared communication system is consequently

[2] GUM has recently been ported to MPI, but in a different version and manner than what we describe here[19].

implemented. The system management by *SysMan* is an exception, since it uses more specific functionality from the MP system, e.g. notification of errors on child PEs and PE placement on the physical machines. It is reasonable to keep the system management closely associated with the concrete MP-system, since a generic version could never anticipate needed functions for particular platforms and will always provide only a reduced functionality.

4 Results

4.1 Expected Effects of Dynamic Chunking

As already motivated in Section 3.1, we can expect considerably decreased runtime particularly when the measured program sends small elements of a long list. The effect depends on the size of the sent data and might be consumed by the additional overhead for the buffer administration. In the following simplifying model, we estimate the runtime change for message buffering in dependency of the global data size and number of messages.

As explained in Section 3.1, all sending operations of a parallel algorithm (put together in one formula) require the constant cost λ per message (assume N_{msg} messages) and a factor β for cost related to the total data *size* (taken over all messages, number of copy operations not taken into account). Without message buffering, we get the following estimation:

$$T_{unbuf.d} = \lambda \cdot N_{msg} + \beta \cdot size \tag{1}$$

By dynamic chunking, we collect the messages in N_{packet} packets and send those packets instead of single messages. This requires an additional copy operation when sending a packet (depending roughly on *size*, since the amount of data is the same, but the operation copies much more data in one call). Simplifying in this way, we get:

$$T_{buf.d} = \lambda \cdot N_{packet} + (\beta + T_{copy}) \cdot size + Overhead \tag{2}$$

where the *Overhead* describes additional actions required for our buffering mechanism. It consists of a constant part for the administration of the buffers on startup and variable cost for preparing the buffer and checking it every time we add a new message to it. The check as well as the preparation are very simple and will be estimated by an upper bound. As we cannot argue about the scheduling loop, where this check is also needed, we postulate a constant number of passes per message (since sending must always be preceded by an evaluation). In all, we get:

$$\begin{aligned} Overhead \;&\leq\; \overbrace{N_{PEs} \cdot T_{alloc}}^{\text{startup}} + N_{msg} \cdot \overbrace{(const' \cdot T_{check} + T_{prepare})}^{\text{Buffer operations}} && (3)\\ &=\; const + N_{msg} \cdot T_{var} && (4) \end{aligned}$$

where the additional cost T_{var} indicates variable cost per message.

Assuming we save $N = N_{msg} - N_{packet}$ messages by dynamic chunking, we decrease the time for sending messages by:

$$T_{unbuf.d} - T_{buf.d} \geq \lambda \cdot N - (T_{var} \cdot N_{msg} + T_{copy} \cdot size + const)$$

or, considering $N = N_{msg} - N_{packet}$:

$$T_{unbuf.d} - T_{buf.d} \geq (\lambda - T_{var}) \cdot N - T_{var} \cdot N_{packet} - T_{copy} \cdot size - const \quad (5)$$

We see that savings of $(\lambda - T_{var}) \cdot N$ stand vis-à-vis to additional costs which depend on the amount of data (*size*) and the number of packets (N_{packet}) (neglecting the constant). As per definition, $T_{var} \cdot N_{packet}$ decreases in the same way as N increases. Postulating an optimal use of the message buffers, the remaining cost is $size \cdot T_{copy}$. It must be said that T_{copy} only gives a rough estimation of the real cost, since the number of copy operations is considerably smaller than for the unbuffered variant, at least reduced by $N = N_{msg} - N_{packet}$. Therefore, we can expect that dynamic chunking has a strong impact when messages are small enough and the program optimally synchronised, while for bigger messages, runtime will increase in a moderate way with bigger amounts of data.

Determining N in practice is a different matter, since it does not only depend on known factors as buffer size and timeout, but also on the global synchronisation, i.e. data dependencies in the computation and speed differences between different machines. And it is even harder to talk about the overhead introduced for *receiving* packets.

4.2 Measurements

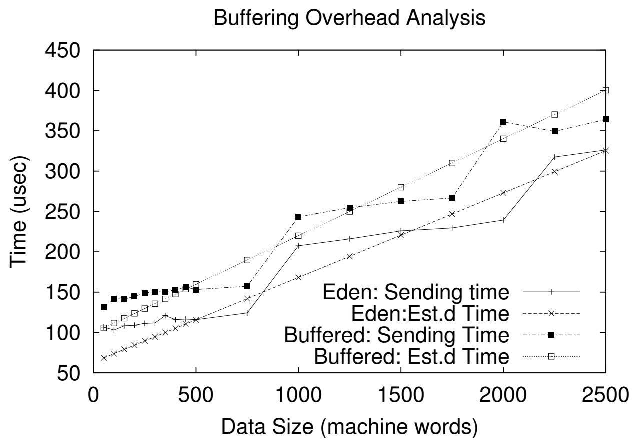


Fig. 4. Overhead measurement corresponding to Fig. 1

Overhead test Fig. 4 shows results of the test setup described in Section 3.1, applied to the Eden implementation which uses dynamic chunking. In this test, the sending operations simulate single-value transmission. Messages are never buffered, but sent immediately using the implemented buffering mechanism, so we get a good estimate of the overhead variables in 5. A linear regression yields a constant $\lambda' = 99.6\ \mu sec$ and a variable $\beta' = 0.12\ \mu sec$, resulting in $T_{copy} = 0.02\ \mu sec$ and $T_{var} = 36.26\ \mu sec$ for the cost model we sketched.

System test The pure effect of dynamic chunking can be observed with a simple system test which does not perform *any* remote evaluation, but only echoes its input list.

```
echo :: Trans a => Process [a] [a]
echo = process id
```

Fig. 5 shows the runtime for echoing a list of 10000 items of the determined size (in machine words, 32 bit). As expected, runtime is much faster with small messages, but not excessively longer for big ones. A characteristic value for this system setup and program is around 300 words per message, where overhead and savings are equal. This size may vary, according to different machine and network setup and to the time spent on computation (zero for `echo`).

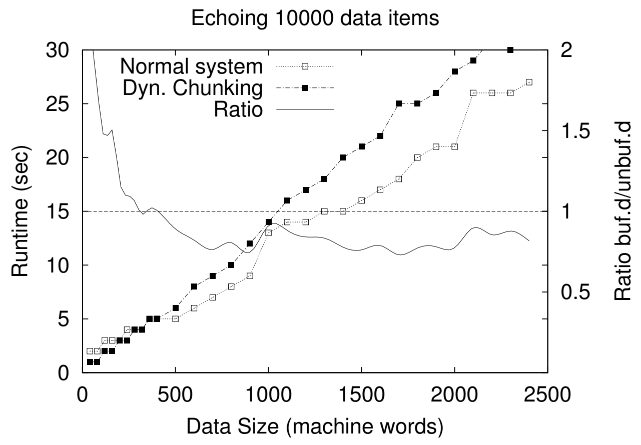


Fig. 5. Measurement with echo process, 10000 messages of variable data size (2 non-dedicated Linux PCs, PC-Pool, Univ. of Marburg)

A variable parameter, besides buffer size, is the buffer timeout for packets, which can be adjusted by a runtime parameter. The system test with `echo` cannot give results for this parameter, since it does not perform any computation on child side and only depends on network speed and synchronisation effects. In all, it is clear that a high buffer timeout may increase the charge of the packets, but it introduces an additional message latency. Only by experiments, the complex dependencies between process synchronisation and message latency can be optimised using this parameter. The implemented runtime statistics on dynamic chunking can help to find the right runtime parameters.

Benchmark programs Dynamic chunking has been tested with the simple sorting functions shown in Section 2.2, as well as with different other benchmark programs: a simple ray-tracer and a Mandelbrot Set visualisation. We used up to eight nodes of a Beowulf Cluster connected through 100MBit Ethernet.

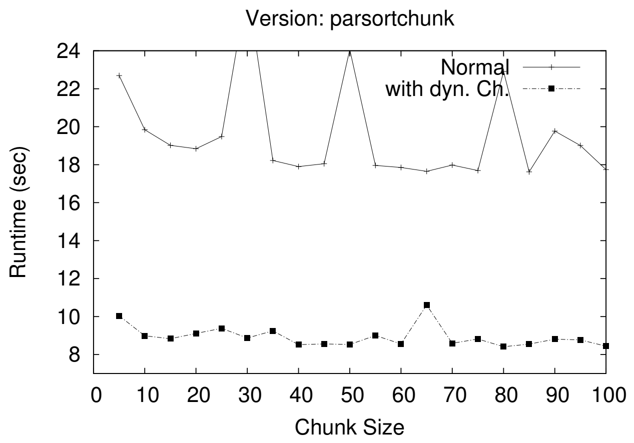
Program	**Problem size**	**Normal**	**Dyn. Chunking**
parsort (8 PEs)	100K Integers	37.9 sec	13.7 sec
mandelbrot (8 PEs)	300x300 pixels	38.3 sec	20.3 sec
raytracer (8 PEs)	huge scene	27.3 sec	26.9 sec

The differences between dynamic chunking and the previous runtime environment are evident. Dynamic chunking applied to a straight-forwardly expressed parallel algorithm, as e.g. the `parsort` program, can speed up runtimes massively (e.g by factor 3 for the sorting program with 100K Integers as input). The Mandelbrot Set visualisation runs up to 40 % faster with dynamic chunking, while the ray-tracer, which is highly hand-optimised and already chunks pixels to lines, has nearly equivalent runtimes.

Unsurprisingly, the sorting program exhibits a rather poor speedup curve, which is due to the sequential start and end phase of the algorithm and to the fast sequential merge sort used. A slower sorting function such as `insertion sort` would show better speedups by totally degrading the overall performance. The Mandelbrot Set visualisation and the raytracer both use predefined implementation skeletons for `map` and show better speedups.

For a fair comparison, we also have to consider hand-tuned variants with static chunking. Fig. 6 shows the impact of these modifications for selected chunk sizes. Apparently, the program runs much faster with the appropriate chunking parameter. Additionally, since the hand-tuning has modified (and improved) the structure of the parallel algorithm, the version `parsortchunk2` outperforms the other one by far.

Measuring programs with static chunking and a runtime system which supports dynamic chunking mixes two effects. The results show that the dynamic chunking mechanism does not replace explicit optimisations, but it does not disturb them either. Hand-tuned programs may run faster with dynamic chunking if their parameters are not optimally chosen. Another interesting point is that the version `parsortchunk`, where only the input to child processes is chunked, exposes rather unstable behaviour when run without dynamic chunking. Since the input arrives much faster, the child processes start working earlier and flood the caller with their results (each element in single message) resulting in high network traffic and affecting other actions on the network. Dynamic chunking



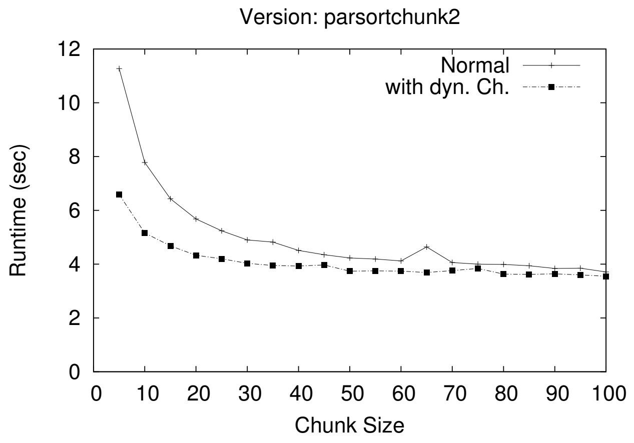


Fig. 6. Influence of static chunking on runtimes in the sorting program (Beowulf Cluster, Heriot Watt University, Edinburgh, 8 nodes)

reduces the number of replies and leads to a much better synchronisation, which is why the runtime is considerably smaller, whereas for the second version, dynamic chunking has almost no effects when bigger static chunk sizes are chosen.

5 Related Work and Conclusion

We have introduced a message buffering mechanism into the runtime environment for the language Eden, which performs list chunking dynamically and adapts itself to the respective program behaviour. Although dynamic chunking is tailored precisely to a specific property of Eden's semantics, the buffering mechanism is generic and can easily be exploited for other parallel Haskell dialects.

Coordination languages and implementations which would profit from using message buffering are those in which PEs communicate much, but often exchange only small data. Necessarily, the impact is limited to languages with (at least partly) explicit communication. When data is only transmitted on demand and separated from evaluation, the implementation can freely choose different strategies, and performance will degrade by message buffering, since it introduces an artificial latency into the communication subsystem. The problem of dynamic chunking for Eden is rather specific and message buffering is usually a concern for more basic software such as message-passing middleware and alike [15, 12, 7], where it is commonly used with success. We are not aware of comparable work in parallel functional languages, but partly related topics are the discussion of different data fetching strategies for GUM in [11] and the SCL subsystem for the data-parallel Nepal [5, 4], which implements a customised library for generic space-efficient vector transmission.

Our measurements show that dynamic buffering massively improves straight-forward parallelised Eden programs, while hand-optimised programs do not profit as much, due to the (nevertheless acceptable) overhead of buffer administration. For programs which are already highly hand-optimised, dynamic chunking affects program performance only in a moderate way. In particular, dynamic chunking performs well for parallel computations with few processors, where the administrative overhead is smaller. The measurements show that dynamic chunking does not completely replace optimisation by static chunking on the language level, but it produces much better results for intuitive straight-forward parallelisations. Programs can remain unchanged, and runtime statistics can give the programmer hints to suitable chunking parameters.

An additional advantage of message buffering in the Eden runtime environment is the new modularity of the communication subsystem. This independence of the concrete MP-system should be exploited for future ports to new platforms and advanced middleware, as well as the general idea of modularity and aspect orientation in the parallel runtime system can be extended to the design of a generic and flexible platform for parallel languages using Haskell as a sequential base.

Acknowledgements. We would like to thank the colleagues from Heriot-Watt University, Edinburgh, for the opportunity to use their Beowulf Cluster and especially Phil Trinder for reviewing a document on the same subject. We also thank Rita Loogen and the IFL workshop attendants for fruitful discussions and suggestions.

References

1. J. Berthold, U. Klusik, R. Loogen, S. Priebe, and N. Weskamp. High-level Process Control in Eden. In H. K. et al., editor, *EuroPar 2003 — Intl. Conf. on Parallel and Distributed Computing*, volume 2790 of *LNCS*, Klagenfurt, Austria, 2003.

2. S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *PLILP'98*, LNCS 1490, pages 318–334. Springer, 1998.
3. S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Progr. Models*, pages 120–124. IEEE Comp. Science Press, 1997.
4. M. Chakravarty and G. Keller. How Portable is Nested Data Parallelism ? In W. Cheng and A. Sajeev, editors, *PART'99*, Melbourne, Australia, 1999. RMIT University, Springer-Verlag. Available at *http://www.cse.unsw.edu.au/˜chak/papers/CK99.html*.
5. M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal — Nested Data-Parallelism in Haskell. Technical report, University of New South Wales, 2000. *http://www.cse.unsw.edu.au/˜chak/papers/ndp-haskell.ps.gz*.
6. I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. *http://www.mcs.anl.gov/dbpp/*.
7. G. Geist, J. Kohl, and P. Papadopoulos. PVM and MPI: a Comparison of Features. *Calculateurs Paralleles Vol. 8 No. 2 (1996)*, 8(2), May 1996.
8. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
9. U. Klusik, R. Loogen, and S. Priebe. Controlling Parallelism and Data Distribution in Eden. In *SFP'00*, Trends in Functional Programming, pages 53–64, Univ of St. Andrews, Scotland, July 2000. Intellect.
10. U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden — Low-Effort Parallel Programming. In *IFL'00*, volume 2011 of *LNCS*, pages 71–88, Aachen, Germany, Sept. 2000. Springer.
11. H.-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96*, volume 1268 of *LNCS*, pages 184–199, Bad Godesberg, Germany, September 1996. Springer. *http://www.cee.hw.ac.uk/˜dsg/gph/papers/ps/packet.ps.gz*.
12. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.
13. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *JFIT'93*, pages 249–257, March 1993.
14. S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at *http://www.haskell.org/*.
15. *Parallel Virtual Machine Reference Manual, Version 3.2*. University of Tennessee, August 1993.
16. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, 1998.
17. P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*, pages 78–88. ACM Press, May 1996.
18. P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *J. of Functional Programming*, 12(4&5):469–510, 2002.
19. A. A. Zain. Heriot-Watt University, Edinburgh. personal contact, July 2003.