CHEMNITZ UNIVERSITY OF TECHNOLOGY

Faculty of Computer Science

Computer Architecture Group

Student Research Project

ESPGOAL
A Dependency Driven Communication Framework

Timo Schneider, Sven Eckelmann

Chemnitz, January 6, 2011

Superadvisor: Prof. Dr.-Ing. Wolfgang Rehm
Advisor: Dipl.-Inf. Nico Mittenzwey

Timo Schneider, Sven Eckelmann
ESPGOAL
Student Research Project, Faculty of Computer Science
Chemnitz University of Technology, 2011

Contents

1 Introduction

The demand for parallel computing continues to rise as a lot of research areas in Biology, Chemistry and Engineering depend on fine-grained simulations or the solution of large systems of differential equations. The hardware available to perform such computations is becoming faster and cheaper. However, increasing the clock frequency of processors is limited by constraints such as power dissipation. So systems are not getting much faster but more parallel since many years.

Many high performance computing (HPC) clusters available today consist of a very large number of cores. These cores are relatively slow compared to modern desktop CPUs and have low-latency access to a high-bandwidth interconnect [AAA+02, ABB+09]. Utilizing the huge number of cores in message passing frameworks such as MPI [Hem94] has been shown to be difficult when point to point messaging is used for communication. It obfuscates the communication pattern, complicates the design and implementation of parallel programs and hinders performance in some cases [Gor04].

Therefore, to develop scalable scientific applications, collective communication should be used instead. One of the merits of MPI is that it includes support for most of the collective communication schemes used in practice. The optimization of MPI collectives is an active research area since many years [AHA+05, TRG05].

Collectives in MPI-2 are blocking, which means communication can not be overlapped by computation. This is expected to change in the future, as non-blocking implementations of MPI collectives, such as LibNBC [HL06] have been shown to improve the performance of scientific applications [HZ07, HGLR07].

Two main problems have been identified when using non-blocking MPI collectives. The first problem is progression: When a non-blocking MPI collective is executed and its execution is overlapped with computation there must be a way to ensure that the communication can progress. For user level MPI libraries this can be done in two ways: either the computation has to be interleaved with calls to the MPI library such as MPI_Test(), or the communication has to be performed in a separate thread. Both options have drawbacks [HL08]: The optimal frequency for the polling with MPI_Test() is hard to determine and depends on machine and network parameters such as interrupt and network latency.

Also it can be impossible to split the computation into smaller pieces to interleave it with MPI_Test() calls, for example when the computation step is a call to an external library.

The progression problem could be solved completely by making the network interface hardware or a part of the software network stack aware of the abstract communication pattern that should be executed so that progression does not have to be ensured by the application. ESPGOAL is implemented as a Linux kernel module, as such it can implement asynchronous progression without polling as it can receive interrupts from the Linux kernel network stack whenever new data has arrived.

Another problem with MPI collectives is that the application can not define new collectives dynamically. ESPGOAL solves this problem as it can execute arbitrary dependency graphs.

The basic idea behind ESPGOAL is to express the role of each node in a collective communication primitive, for example a barrier, as a dependency graph between non-blocking send and receive operations and local transformations on data (similar to MPIs reduction operations). Figure 1.1 gives an example for the data movement pattern in a scatter oper-

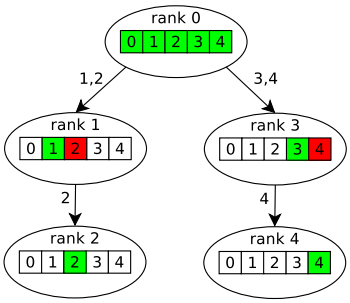


Figure 1.1: Data movements for a tree based scatter operation with five nodes

ation across five nodes which is implemented in a tree based manner. Data items in green can be stored in the buffers that would be passed to the collective function by the application, if we assume MPI semantics. Red data items have to be stored in a temporary buffer until the collective operation is finished for the respective rank.

To carry out this communication pattern each rank has to perform a distinct set of communication tasks. If we assume each data item is a four bytes in size rank number ones tasks can be expressed as

1. Receive 8 B from rank 0 into temp buffer at offset 0

2. Copy 4 B from temp buffer offset 0 into recv buffer offset 0

3. Send 4 B from temp buffer offset 4 to rank 2

Care must be taken to prevent the execution of tasks two and three before task number one is finished. We can represent the task list as a dependency graph as shown in Figure 1.2. The nodes in the dependency graph consist of primitive operations such as non-blocking sends and receives as well as simple transformations on data in local buffers. An edge $u \rightarrow v$ in the graph means that there is a dependency on v which is resolved after the task represented by u is completed. An operation may only be started if it has no incoming edges or if all operations on the tail of its incoming edges are already finished. At any point in time, operations that do not have unmet dependencies should be able to progress independently and in parallel. Therefore all send and recv operations are non blocking. A non-blocking operation is considered to be finished after it can be guaranteed that all buffers used by it can be overwritten by other operations.

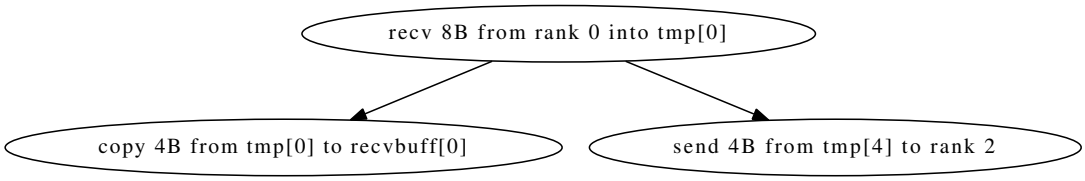


Figure 1.2: Local task graph for rank one (cf. Figure 1.1)

Dependency graphs as the one shown in Figure 1.2 will be compiled into a space efficient binary object so that it can be copied into the kernel memory space with a single `copy_from_user()` call. The application can instruct the ESPGOAL kernel module to execute the dependency graph, once it is compiled to a binary object, called *schedule* from now on. The idea to express collective communication tasks as dependency graphs was published by Höfler et al. in [HSL09b].

To carry out the send and receive operations specified in the schedule, our kernel leverages a modified version of the Ethernet Stream Protocol (*ESP*). The original version of ESP is explained in detail in [HRM+06], later, flow control was added to ESP. The flow control mechanism is documented in [Tre07]. ESP is a reliable, low overhead protocol for communication over Ethernet. It was chosen to be the transport layer for this project because we were not aware of a Ethernet communication protocol which offered a non-blocking kernel API and one of the authors, already gained enough inside into ESP in prior projects to be able to implement such an API on top of the existing ESP codebase.

1.1 Related Work

The ESP protocol used as a basis for this work is a low overhead protocol for generic Ethernet clusters. As such it bears resemblance to a number of projects with a similar goal like GAMMA [Cia03], EMP [SWP01] and Open-MX [Gog08].

EMP and GAMMA modify the firmware of network adapters or rely on device specific hardware features to, for example, query the NIC in polling mode rather then waiting for an interrupt on packet arrival to reduce the point to point latency or to be able to perform zero-copy communication over generic Ethernet hardware. As such these projects are very different from ESP which does not depend on specific low level NIC features. However, the results obtained by the respective research groups are remarkable, GAMMA claims to achieve an end to end latency of about half of what we measured with ESP on our testbed.

Open-MX is a low overhead Ethernet protocol implementation which is API compatible to Myrinet MX [Geo04]. Contrary to EMP and GAMMA, Open-MX is not depending on low-level hardware features. It tries to achieve good performance by avoiding memory copies where possible. Incoming data is written directly to application buffers in most cases. For this to work the application buffers have to be pinned, i.e. the operating system is not allowed to swap the memory pages associated with these buffers. When the NIC receives new data an event is generated by the receive hook associated with the ethertype used for Open-MX packets. This event is put in a queue which resides in a pinned memory region shared between the Open-MX kernel module and its userspace library. Open-MX requires the application to call into its userspace library often enough so that this queue does not overflow. As such it is impossible to directly use the Open-MX module to implement a GOAL scheduler as a kernel module, as Open-MX does not define a kernel API which offers similar features as the userspace library. However, as Open-MX shows much better performance in latency benchmarks than ESP it could be a possible work item for the future to derive such an API from the existing Open-MX protocol and replace the ESP transport layer.

The only userspace implementation of non-blocking collectives known to the authors, LibNBC [HLR07, HL06], naturally can not offer true asynchronous progress, as it can not rely on network interrupts for progress, so progress has to be ensured by frequent polling by the application or by having a separate progression thread. Both approaches increase the host overhead. The collective functions offered by LibNBC have the same syntax and semantics (where applicable) as the collectives defined by MPI [Hem94]. The way to express collective operations in LibNBC is equivalent to that of GOAL in its expressiveness, however, in LibNBC dependencies can only be defined between “rounds” instead of single operations but since there are no restriction on how many operation a round has to con-

tain it is essentially the same. Schedules in LibNBC are also compiled into cache-friendly binary objects prior to execution.

While this work was in progress, another research group published a paper [NI10] which describes a kernel module which also enables userspace applications to define collective communication schedules which are then progressed in kernel context. Unfortunately their implementation, named KACC, is not publicly available as of now. The userspace API is semantically similar to the one offered by GOAL. The main difference between ESPGOAL and KACC is that KACC executes the *Progress Engine* (*PE*) in the SoftIRQ context inside of a Linux kernel *Tasklet*, where ESPGOAL uses a *Workqueue* to implement the same functionality. While the use of Tasklets might lead to a lower scheduling latency (the difference estimated by some microbenchmarks later in this work could be up to 1μs), executing the progress engine, or *scheduler* in ESPGOAL terms, also has certain disadvantages. For example it is impossible to use Linux kernel sockets as the transport layer in a Tasklet based implementation, as certain operations on sockets can sleep and it is illegal to call such functions in a Tasklet. That means the underlying communication layer has to be implemented directly on top of the device driver. This does not make the implementation unportable, as this part of the networking API is already device independent in Linux, however, as this layer offers none of the abstractions known from socket programming to the developer, implementing a communication layer on top of it is cumbersome. The authors claim to use TCP as their transport layer, however, for the reasons explained above, we can only speculate that they reimplemented at least parts of the TCP networking stack already in the Linux kernel. Another implication of the usage of Tasklets instead of Workqueues is that all memory that is accessed from inside the Tasklet has to be kernel memory or pinned memory.

2 The GOAL API

2.1 API Conventions

All GOAL API functions and datatypes visible to the user should have the Prefix "GOAL_". Those not visible to the user should have the prefix "_GOAL_".

2.2 Basic GOAL Functionality

2.2.1 Initialization

Before any other GOAL operation is called the user has to initialize the GOAL communication subsystem by calling

```
1 int GOAL_Init()
```

This function may only be called once by every process. To check if it was already called before the user can call the function

```
1 int GOAL_IsInitialized()
```

which will return 1 if GOAL was already initialized for the calling process, 0 otherwise. Note that this variant of the GOAL API is not thread-safe, as we do not pass any state information to GOAL. For the thread-safe API see its separate documentation.

After GOAL is initialized we can find out how many endpoints are participating in our communication group with the function

```
1 int GOAL_GroupSize()
```

it will return the number of endpoints in our communication group. Note that GOAL does not have communicator support as known from MPI — a communication group can only be defined at the start of a GOAL application.

GOAL will enumerate all endpoints with numbers from 0 to GOAL_GroupSize()-1. To find out the number of our endpoint we can call

```
1 int GOAL_MyRank()
```

Note that neither the group size nor the local rank can change during a GOAL applications life cycle, so it is unnecessary to call these functions more than once.

To free all resources allocated by GOAL_Init() an application has to call the function

```
1 int GOAL_Finalize()
```

After that no other GOAL functions can be called, except GOAL_IsFinalized(). Similar to GOAL_IsInitialized() that function can be used to check if GOAL was already finalized.

```
1 int GOAL_IsFinalized()
```

which will return 1 if GOAL is already finalized or not initialized yet, 0 otherwise.

2.2.2 Graph Creation

The basic idea behind GOAL is to describe the dependencies among a series of communication and computation operations, so that the GOAL interpreter can execute these in any order which satisfies those dependencies. Dependencies can be described with directed acyclic graphs (DAGs). Such DAGs are the basic building blocks for GOAL.

So the first function which must be called to use GOAL creates a new GOAL_Graph object:

```
1 GOAL_Graph GOAL_CreateGraph()
```

Once we are finished with a GOAL_Graph (i.e. after it has been successfully compiled to a binary schedule) we can free all resources used by that graph with the function

```
1 int GOAL_FreeGraph(GOAL_Graph g)
```

2.2.3 Adding Operations

Now we can add operations (send, receive and computations). Those will be represented as vertices in the graph, so all vertex-adding functions return an GOAL_Vertex object, which is a handle to the newly added vertex. Of course each operation needs some parameters to specify its task. All calls to vertex-creating functions have the same first parameter, the GOAL_Graph Object which they should be added to.

In case of the send operation the user needs to specify the sendbuffer, the number of bytes to be sent and the destination rank.

```
1 GOAL_Vertex GOAL_Send(GOAL_Graph graph, void* buf, int count, int
      dest, int tag=0, GOAL_MemType mem=GOAL_USERSPACE)
```

The last parameter of this function is common for all buffers in ESPGOAL: Buffers can be pointers to memory allocated by the program or byte offsets in the scratchpad memory region of that schedule. The buffer argument will be interpreted as a normal pointer if the "mem" argument is set to GOAL_USERSPACE (the default). We will explain the creation and usage of scratchpad memory in detail in Section 2.2.5.

To receive data GOAL provides a receive operation, which can be added to the graph with the GOAL_Recv() function. The meaning of its parameters is analogue to GOAL_Send().

```
1 GOAL_Vertex GOAL_Recv(GOAL_Graph graph, void* buf, int count, int
      source, int tag=0, GOAL_MemType mem=GOAL_USERSPACE)
```

GOAL also supports sending and receiving multiple pieces of (possibly discontinuous) data in a single operation. This is done by giving a list of pointers to the beginning of the individual pieces of data as well as a list of lengths that describes how many bytes should be copied from each specified location to the GOAL_SendVec() function. These parameters are given to GOAL by-reference, however, the contents of these lists are immediately copied into the GOAL graph by the GOAL_SendVec() function, they can be freed or overwritten after that function returned. The actual data is of course (as in the case of GOAL_Send()) red during the schedule execution. The interface of the GOAL_SendVec() function is:

```
1 GOAL_Vertex GOAL_SendVec(GOAL_Graph graph, void** offsets, int*
    lengths, int num_elements, int dest, int tag=0, GOAL_MemType
    mem=GOAL_USERSPACE)
```

The first parameter *graph* specifies to which GOAL_Graph object the newly created operation will be added to. The *offsets* parameter is (in the default case, see below) an array of void pointers, this array has to be at least of size *num_elements* and it contains the start addresses of the data blocks that have to be sent. The *lengths* parameter is an array of integers that specify how many bytes each data block consists of. This array has to contain at least num_elements entries. Suppose offsets[i] has the value of 0x82234242 and lengths[i] is 32. That means that the i-th data block starts at address 0x82234242 and consists of 32 bytes. Note that GOAL will send all blocks as a contiguous piece of data without padding, starting with block 0 to block num_elements-1. The rank who will receive that data is given in the parameter *dest*.

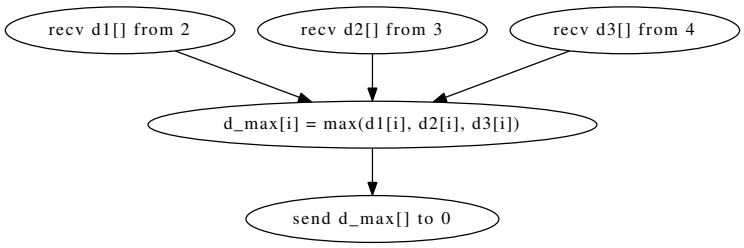
To receive data and store it in a non-contiguous manner the function GOAL_RecvVec() has to be used. Its prototype is analogue to GOAL_SendVec():

```
1 GOAL_Vertex GOAL_RecvVec(GOAL_Graph graph, void** offsets, int*
    lengths, int num_elements, int src, int tag=0, GOAL_MemType
    mem=GOAL_USERSPACE)
```

Of course it does not have a destination parameter but a *src* parameter in its place to specify from which rank we want to receive from. Note that the contents of the arrays *offsets* and *lengths* do not have to be the same for a vector-send and the corresponding receive to match. Messages are matched only on the overall size of the of the data. This makes it possible, for example, to send an array stored in row-major order and store it in column-major order on the receiving rank.

The vector-send and the vector-receive function have a common parameter *mem*. This parameter specifies if the contents of the *offsets* parameter will be interpreted as void pointers to location in userspace memory (which is the case when mem=GOAL_USERSPACE, the default) or as byte offsets in the scratchpad memory space (the allocation and usage of scratchpad space is described below). In the scratchpad-case, when mem=GOAL_SCRATCHPAD the void pointers in *offsets* will be casted to ints.

The third type of operations that can be used in GOAL (besides sending and receiving of data) are so called *local operations*. A local operation does not involve any communication, it is executed on the local rank (the one executing the corresponding schedule) only. The purpose of local operations (also called localops) is to enable the dependency based execution of simple arithmetic operations inside of GOAL. For example when implementing a collective with similar semantics as MPI_Reduce() the user might want to define a dependency graph similar to this one:



That can be accomplished in GOAL by implementing the mathematical operation $d[i] = max(...)$ as a local operation. Doing so gives the GOAL interpreter the freedom of executing the local operation as soon as all the data dependencies are fulfilled and wherever it is appropriate, for example directly on the NIC if it the GOAL interpreter has the capability to do so. However, to achieve this kind of freedom it must be ensured that local operations are "simple enough" to be executed by the GOAL interpreter. Therefore only predefined local operations are possible in the Basic GOAL API. The predefined operations available in GOAL are:

Datatype	GOAL_SINT					GOAL_UINT					GOAL_FLOAT				
Width	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64
GOAL_MAX															
GOAL_MIN															
GOAL_ADD															
GOAL_SUB															
GOAL_DIV															
GOAL_MUL															
GOAL_COPY															
GOAL_AND															
GOAL_OR															
GOAL_XOR															
GOAL_WTIME	no type checking/conversion, puts timestamp [in s] in buf3 as 64bit float														

Local operations can be added to a GOAL graph with the function:

```
1 int GOAL_LocalOp(GOAL_Graph graph, void *b1, void *b2, void *bres
    , GOAL_Op op, GOAL_DataType dt, int element_width, int
    num_elements, GOAL_MemType b1_mem=GOAL_USERSPACE, GOAL_MemType
     b2_mem=GOAL_USERSPACE, GOAL_MemType b3_mem=GOAL_USERSPACE)
```

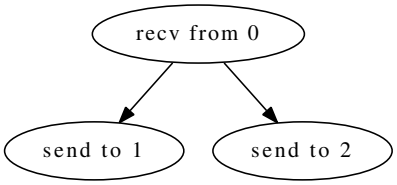
Upon execution the buffers *b1*, *b2* and *bres* will be interpreted as arrays of the type given in the *dt* parameter, with each element consisting of *element_width* bytes. For example a type parameter with a value GOAL_UINT and an element width of 16 means that the buffers will be interpreted as arrays of type uint16_t containing *num_elements* entries. The operation that will be carried out is specified by the *op* parameter. All valid values for the parameters *dt*, *element_width* and *num_elements* are given in the above table. A green rectangle means this combination of parameters is allowed, a red one means this combination is illegal. Operations that take two parameters (all, except GOAL_COPY) have the following semantics: $\forall i : b_3[i] = b_1[i] \otimes b_2[i]$ where $\otimes$ is the infix operator for the specified operation, for example "$-$" in the case of GOAL_SUB. When the memtype arguments are GOAL_USERSPACE the buffers will be assumed to be normal pointers in userspace, if the memtype parameters have the value GOAL_SCRATCHPAD the corresponding buffer arguments will be casted to ints and interpreted as offsets in the scratchpad buffer.

There is also a vector variant of GOAL_LocalOp with similar semantics which has the following interface:

```
1 GOAL_Vertex GOAL_LocalOpVec(GOAL_Graph graph, void** buf1, int*
    lengths1, int num_elements1, void** buf2, int* lengths2, int
    num_elements2, void** buf3, int* lengths3, int num_elements3,
    char datatype, char datatype_width, char op, GOAL_MemType
    b1_temp=GOAL_USERSPACE, GOAL_MemType b2_temp=GOAL_USERSPACE,
    GOAL_MemType b3_temp=GOAL_USERSPACE)
```

2.2.4 Adding Dependencies

As we mentioned earlier the execution of the specified operations will be dependency-driven. To add dependencies between operations the GOAL_Requires() function must be used. Each of the functions that will add an operation to the graph will return a GOAL_Vertex object which identifies that operation in the graph. These identifiers can be passed to GOAL_Requires() to link them together: Suppose we want to receive some data from rank 0 and upon reception we want to send this data to rank 2 and 3. The dependency graph for this case would look like this:



Using the GOAL_Requires() function

```
1 GOAL_Requires(GOAL_Graph g, GOAL_Vertex prereq, GOAL_Vertex
    target)
```

this can be expressed with the following piece of code:

```
1 GOAL_Vertex recv, send1, send2;
  recv = GOAL_Recv(g, &buf, 2, 0, GOAL_USERSPACE);
3 send1 = GOAL_Send(g, &buf, 2, 1, GOAL_USERSPACE);
  send2 = GOAL_Send(g, &buf, 2, 2, GOAL_USERSPACE);
5 GOAL_Requires(g, recv, send1);
  GOAL_Requires(g, recv, send2);
```

2.2.5 Scratchpad Buffer

Since the operations in a GOAL graph can be executed at any time after its definition by the user, possibly also multiple times, allocating and freeing temporary buffers is difficult for the user. One might be inclined to do something like this to implement a tree based scatter:

```
  buf = malloc(64);
2 recv = GOAL_Recv(g, buf, 64, 1, GOAL_USERSPACE);
  send1 = GOAL_Send(g, buf, 32, 2, GOAL_USERSPACE);
4 send2 = GOAL_Send(g, buf+32, 32, 3, GOAL_USERSPACE);
  GOAL_Requires(g, recv, send1);
6 GOAL_Requires(g, recv, send2);
  free(buf);
```

However, the problem with this approach is that the buffer is freed in line 7 before the schedules execution is finished (in this example we did not even compile or start the schedule execution).

If we omitted free() in the last line this example would work fine, however we would have produced a memory leak as we never free buf now. One way around this would be to manually track the life cycle of the schedule resulting from the Graph g and free buf when the schedule is finished and will not be executed again. This is inconvenient and error-prone, especially in scenarios where we want to hide the actual collectives (i.e. scatter) implementation in a function call.

Therefore GOAL supports a very minimal memory management subsystem, the scratchpad buffer. We can inform GOAL that to execute the operations defined in "graph" we need at most "bytes" bytes of temporary buffer space with the function

```
1 int GOAL_AllocateScratchpad(GOAL_Graph graph, size_t bytes)
```

If we do that the GOAL interpreter will allocate a buffer of this size immediately after the schedule corresponding to "graph" is executed and that buffer will be destroyed as soon as the schedules execution is finished. To use that buffer we have to supply offsets instead of pointers to any function that takes an argument of type GOAL_MemType. For the memtype argument we have to supply GOAL_SCRATCHPAD instead of GOAL_USERSPACE, which has to be used for standard memory accesses. The supplied offsets are relative to the start of the scratchpad buffer. Note that it is not possible to have more than one scratchpad buffer. If several buffers are needed the user has to allocate a single buffer large enough and perform the memory management himself.

The correct version of the example given above would look like:

```
1 GOAL_AllocateScratchpad(g, 64);
  recv = GOAL_Recv(g, 0, 64, 1, GOAL_SCRATCHPAD);
3 send1 = GOAL_Send(g, 0, 32, 2, GOAL_SCRATCHPAD);
  send2 = GOAL_Send(g, 32, 32, 3, GOAL_SCRATCHPAD);
5 GOAL_Requires(g, recv, send1);
  GOAL_Requires(g, recv, send2);
```

To copy data to/from the scratchpad memory we can use a local operation of the type GOAL_COPY. See the documentation on local operations for more information.

2.2.6 Schedule Compilation

After we added all operations and dependencies to a GOAL_Graph we can transform it into a GOAL_Schedule. Such a schedule is a compact binary representation of the corresponding GOAL_Graph, optimized for the processing by a GOAL interpreter. A schedule can not be changed any more by the user. To compile a GOAL_Graph the function

```
GOAL_Schedule GOAL_Compile(GOAL_Graph graph)
```

has to be used. Note that after a successful compilation the GOAL_Graph is no longer needed unless we want to generate another schedule with more operations or dependencies.

A GOAL interpreter can invalidate a GOAL schedule during execution. To find out if a GOAL_Schedule is valid the function

```
1 int GOAL_IsScheduleValid(GOAL_Schedule sched)
```

can be used. It will return 1 if a schedule is valid (i.e., it can be executed by GOAL_Run) and 0 otherwise. Schedule invalidation happens if the GOAL interpreter directly operates on the schedule in userspace. During execution the dependency counters will be decremented. After execution such a schedule would contain the same operations as before but no dependencies between them.

If we want to reuse (i.e., execute it multiple times) a schedule and we are dealing with a schedule-invalidating GOAL implementation, we can tell GOAL not to invalidate that particular schedule (i.e. by copying it internally if necessary). For this purpose use the function

```
1 void GOAL_MakeScheduleReusable(GOAL_Schedule sched)
```

that should always be used were applicable to write portable GOAL programs.

Once we are done with a schedule (i.e. its execution is finished and we do not want to execute it again) we can free all resources used by it using

```
1 int GOAL_FreeSchedule(GOAL_Schedule sched)
```

to find out if a schedule is finished the user has to keep track of all the GOAL_Handles associated with that schedule.

2.2.7 Schedule Execution

To start the execution of a compiled schedule the user has to call

```
1 GOAL_Handle GOAL_Run(GOAL_Schedule sched)
```

be aware that a schedule *can* be invalidated by the interpreter, depending on its implementation, during execution. To prevent that one can use the GOAL_MakeScheduleReuseable() function, as explained before.

To check if a schedules execution is finished, GOAL provides two different functions:

```
1 int GOAL_Test(GOAL_Handle handle)
```

will return 1 if the corresponding schedule (the one for which handle was returned when it was started with GOAL_Run) is finished executing, otherwise 0. It will not block

```
1 GOAL_Wait(GOAL_Handle handle)
```

on the other hand will block until the corresponding schedule is finished. A schedule can be freed with GOAL_FreeSchedule() when all its copies are finished. If a schedule is freed while it is still in execution the results are undefined.

2.3 GOAL-Extensions

Beyond the basic functionality described above several extensions could be useful. For example it would be desirable to support user defined local operations. With user defined local operations it would be possible to implement something semantically similar to the active messages paradigm [PSZ92] without actually sending around function pointers like originally suggested. However implementing support for user defined local operations in a GOAL interpreter can be hard. Imagine a GOAL interpreter that runs on the NIC itself. The code the actual operation consists of must be compiled for the processor architecture used by the NIC instead of the hosts architecture, for which the rest of the application was compiled for. Memory accesses must be trapped and deferred to the same address space the main application is running in.

A possible option would be to use runtime-loadable objects produced with a separate tool chain (cf. Cell BE). Here the user must have full knowledge of the target architecture and the target environment. Another option could be to specify a simple language which can be interpreted at runtime by an extended GOAL interpreter. However we did not consider such extensions in this work.

A possible way to provide different extensions in different GOAL interpreters would be to have a function

```
1 int GOAL_IsExtensionAvailable(GOAL_ExtensionId ext)
```

which returns 1 if the specified extension is provided by the currently used GOAL interpreter. Each extension can add a set of functions of the type GOAL_* but may not alter the semantics of the functions described above.

3 ESP Transport Layer

Usually systems using IEEE 802.3 Ethernet as data link layer use TCP/IP as their network and transport layer. These protocols are designed for robustness in wide area network environments and add an additional overhead during a transfer. This additional processing is needed to handle routing, fragmentation and reordering. But it was shown that switched clusters can benefit from a simplified protocol for both connectionless and connection-oriented communication [HRM+06].

ESP/EDP, which was developed at TU Chemnitz [Tre07], was chosen to integrate a GOAL interpreter. It provides a general solution for switched Ethernet networks based on Linux without the dependency on special hardware or driver like U-Net, VIA, EMP or Gamma. It can be used through sockets like UDP/IP or TCP/IP, but works with a slightly different address format and a new protocol family `PF_ENET`. Transfers/connections are identified using Ethernet MAC addresses for communication partners and 16 bit ports.

```
1  int fd;
   struct sockaddr_en my_addr;
3
   if ((fd = socket(AF_ENET, SOCK_STREAM, 0)) == -1) {
5    perror("socket"); exit(1);
   }
7
   memset(&my_addr, 0, sizeof(my_addr));
9  my_addr.sen_family = AF_ENET;
   my_addr.sen_addrlen = ETH_ALEN;
11 my_addr.sen_port = htons(PORT);

13 if (bind(fd, (struct sockaddr*)&my_addr, sizeof my_addr) == -1) {
     perror("bind"); exit(1);
15 }
   if (listen(fd, 0) == -1) {
17   perror("listen"); exit(1);
   }
19
   close(fd);
```

The Ethernet Datagram Protocol (EDP) provides a solution for connectionless communication based on top of raw Ethernet messages. It can only multiplex transfers using ports and can handle different control messages to get ICMP-like communication between nodes. It cannot provide any sort of reliable communication or congestion control which makes it inappropriate to exchange large amount of data. In its current form only the ICMP like functionality is used by the MPI Ethernet BTL to detect which network interface card provides a link-local connection to another host.

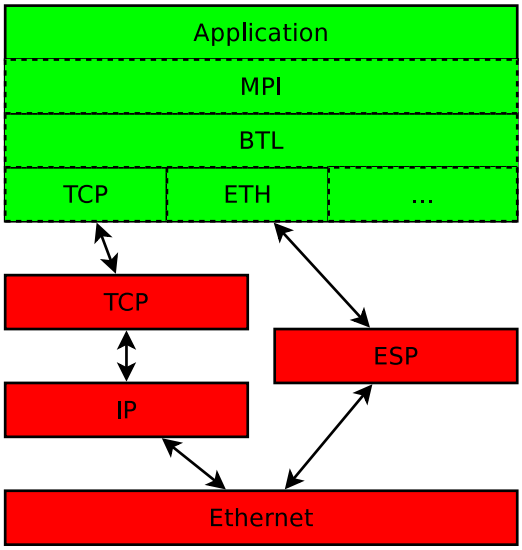


Figure 3.1: Standard integration of ESP in MPI — tasks in red are carried out by the kernel module, green ones are performed in userspace.

ESP was implemented in the `PF_ENET` protocol family as extension for this unreliable, connectionless protocol. It is a connection-oriented, port multiplexed and reliable protocol on top of Ethernet and supports hand optimized congestion control for static, switched networks. It can be used through standard sockets and is implemented in a single Linux kernel module. This makes it possible to add own functionality and to integrate it in a kernel based version of the GOAL interpreter with nearly no adjustments to the transport protocol application programming interface.

Figure 3.1 shows that the current integration in a message passing library is done similar to TCP. Sockets provided by the Linux kernel are used together with the new protocol family. Unlike OpenMX, no special userspace library is needed to communicate with the transport layer. This makes it easier to implement a communication management layer inside kernelland since the kernel itself manages all socket operations and provides an abstract in-kernel sockets API with the release of Linux v2.6.19.

3.1 Receive Handling

When a packet gets received by a network interface card, an interrupt is generated to inform the kernel about the new data it can now process. The kernel normally tries to stop the interrupts for that card, gets all needed data, reactivates interrupts and then tries to process as much data as possible in a SoftIRQ context. Processing means that the data is prepared for the next layer inside the network stack. It depends on the actual card what this could mean.

A card could for example send all its data using DMA to a socket buffer created earlier[1] and the processing routing has now to dequeue that buffer from a data structures which holds all buffers the NIC is allowed to use[2]. Not all NICs are able to that. A different way to handle the receiving interrupt for a driver is to create a socket buffer and fetch the data from dedicated memory[3]. The socket buffer will be send to the next layer for processing using specialised receiving functions like `netif_rx()`[4] or `netif_receive_skb()`[5].

In each layer the socket buffer gets now prepared for the next higher layer and protocol dependent information is stored in a private data region of the socket buffer, so it can be used without the information were it came from. For example the Ethernet frame contains a field for the EtherType in the MAC header. The `PF_ENET` protocol family registered its subprotocols using a constant integer equal to the EtherType in the MAC header and a receiving hook which gets called by the underlying layer.

After ESP was notified about the reception of an Ethernet frame it has to process the data further to ensure that enough data is available for at least its header. If the received frame is not too small then the receiving socket can be determined from it by using the MAC header (source and destination MAC address) and information from the ESP header (source and destination port) as seen in Figure 3.2. This is enough information to decide if we have a valid connection established and the data can be processed further when the user is not currently accessing the socket. Otherwise a private per socket queue is used to create a backlog that is processed after the user is not holding the socket anymore.

In case there is no socket available were we can add a backlog then there is either no user available which can hold the socket or the received packet can be identified as invalid transfer and must be dropped.

[1] drivers/net/r8169.c line 487 in Linux v2.6.36
[2] drivers/net/r8169.c line 4591 in Linux v2.6.36
[3] drivers/net/xilinx_emaclite.c line 606 in Linux v2.6.36
[4] net/core/dev.c line 2489 in Linux v2.6.36
[5] net/core/dev.c line 2945 in Linux v2.6.36

ESP has to ensure that all packets it received are in correct order and no data was lost during the transfer. Afterwards it has to finish the bookkeeping and maybe acknowledge some messages.

In this situation, a userspace application could start to read new data from the socket buffer into its data buffer and may finish a message. The application has to call either a blocking receive function for the data or periodically a non-blocking test function. Otherwise it would never know that the transfer has finished.

This means for non-blocking collectives that the application has to call a test operation provided by the implementation of the non-blocking collective until the complete data which belongs to it is processed. A different approach is to use dedicated threads which have to handle the progression of the collective [HL08].

An alternative way of progression handling for non-blocking transfers will be introduced later in form of the ESPGOAL communication management layer.

3.2 Transfer Management

It can happen that packets were dropped during the transfer. The cause could be errors during the transmission or congestion at the receiving end or during the transmission, for example in a switch. In case of congestion, it is most of the time not the right decision to send even more data in form of retransmissions.

In ESP two classes of packets are defined to help to avoid congestions: One class are packets which are used to transfer data and the other class are packets used to signal changes in the state of a socket. The latter one must always be transferred to keep the state of a socket in sync, but the first class can be paused to avoid further congestion.

Figure 3.2 shows that, next to the flags SYN, ACK, FIN and RST, we also have RRQ, TXS, TXF and TXR which are new compared to flags available in TCP. These are needed to implement the new congestion aware retransmission scheme. TXR is actually an unused flag in the ESP implementation, but can be used later in ESPGOAL.

A single transfer consists of two separated stages. In the initial stage only the sender has to retransmit packets until it can be sure that the receiver is also aware that a transmission started. The second stage starts when the receiver acknowledges the packet which was marked using the TXS flag. After that only the receiver has a timeout running which forces a re-request of a specific amount of packets which has not reached him. This makes

it easier for the receiver to decide if it can or cannot handle more data right now or if the sender should stop the transmission until the data in the socket buffers was processed.

A transmission is automatically stopped when the receiver gets a packet marked as “transmission finished” with the TXF flag. This flag does not mean that a single write to the socket was finished, but that the sender did not know about more data it should send at the point when it did send the packet the first time.

This means that multiple writes to a socket can be coalesced into a single transfer by removing the TXF flag of the last transmission and the TXS flag of the new transmission if they are still in the write queue and therefore not send yet.

As special behavior added in the development of ESPGOAL, the TXF flag always adds the TXR flag, but during a merge of two transfers the TXR will not be removed.

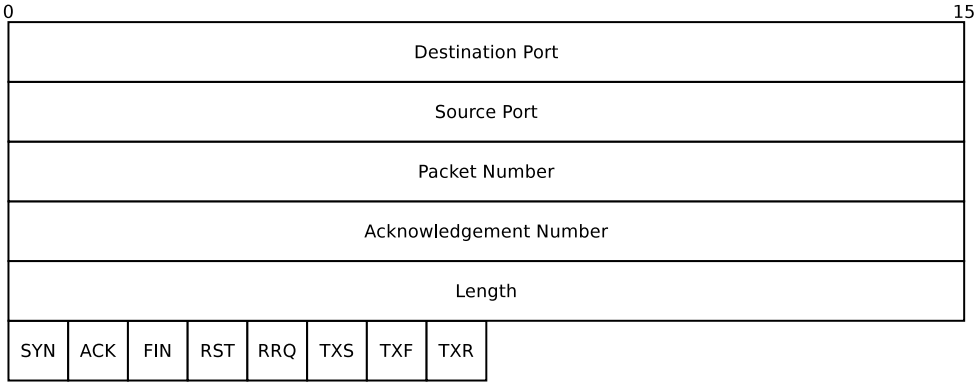


Figure 3.2: ESP packet header with added TXR flag

The merging reduces the amount of time the sender is in the initial stage where he has to actively send data to the receving host and thus has a higher probability to create congestion during that stage. [Tre07]

3.2.1 Known Problems

During the development of ESPGOAL different implementation details had to be corrected to prevent various hangs and race conditions during transfers. All those changes were send to the ESP repository[6] and are not differences between ESP and ESPGOAL.

[6] http://www.unixer.de/research/commopt/

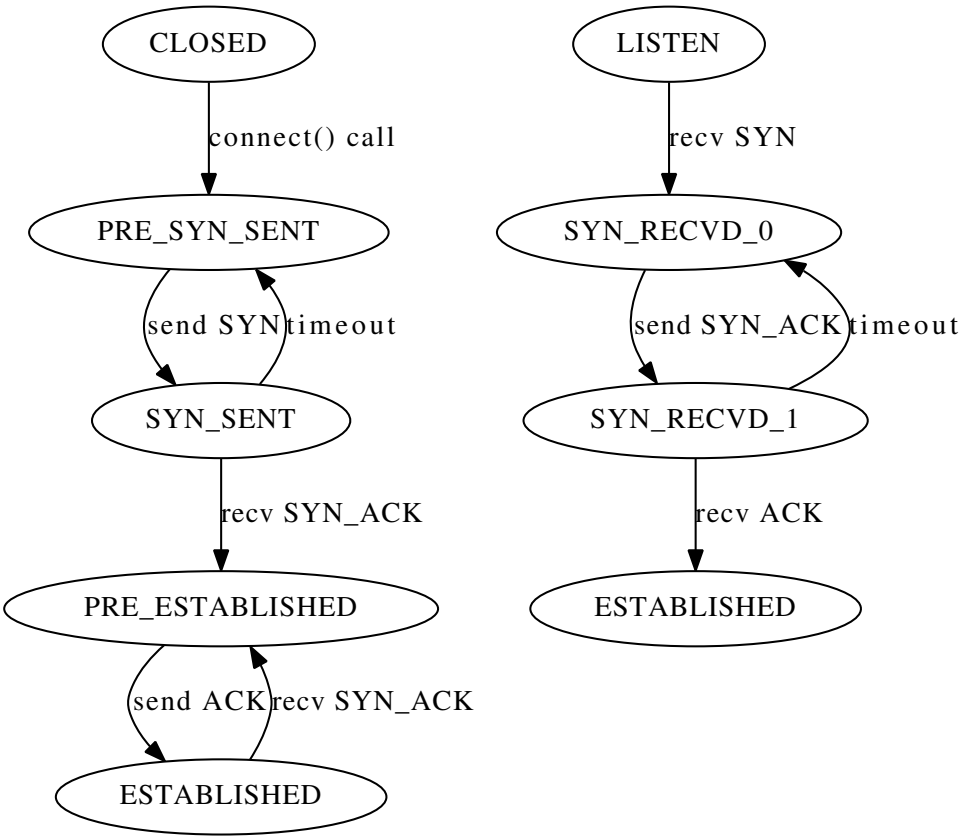


Figure 3.3: New ESP connection setup state machine for connect() (left) and bind() (right)

One of the most important change is the redesign of the connection setup. The original design was made for systems were we can be sure that no packets will be dropped during the handshake and no packets will be repeated. This assumption is not true in many situations. We could for example create a situation were either the connecting or the listening host receives a lot of packets and either a switch or the receiving network card has to drop the data. Each type of packet (SYN, SYN+ACK, ACK) can now be dropped and one host may be in a different state compared to the connection partner. In many situations, this will result in a hang due to the fact that not both partners have the state ESTABLISHED and thus will drop packets which announce new transfers. There is also the situation that the caller of connect is in the state SYN_SENT and receives a SYN+ACK packet. That implied that it knows that his connection is established and his connection partner only waits for an ACK packet. If that ACK packet is dropped then only one side knows about the finished connection handshake and even a hypothetical resend of the SYN+ACK packet would have caused a connection reset.

A new connection handshake similar to TCP was implemented as shown in Figure 3.3. Now all states will be able to accept packets with SYN flag set and will try to retransmit their handshake packet in case either their own packet or the ACK packet was dropped or delayed.

A similar problem exists with the acknowledgment of TXF flagged packets. Usually the receiver is forced to send an ACK to allow the sender to start a new transmission. The sender is only transferring `initial_burst_length` packets before it hands over the complete control over the transmission handling to the receiver, but the receiver only sends RRQ in case it wants more packets. The receiving node stops to ask for more packets when a TXF flagged packet was received and no new TXS packet was detected. It is possible that an ACK packet for a TXF packet is dropped. In that situation the sender does not know that all packets including the TXF flagged packet are acked and the receiver assumes that it is no longer in control over the transmission. The only reason that this transmission does not stop is that `initial_burst_length` in many situations is larger than the difference of the sequence numbers between the last acked packet and the new TXS flagged packet. The next TXS packet can be send as part of the `initial_burst_length` window of sequence numbers and the sender ensures through retransmissions of this packet that the receiver is aware of his role as controller of the transmission.

The ESP protocol currently does not assure that the amount of packets it can send after the last acked packet is smaller than `initial_burst_length`. This may lead to the situation that the difference between the sequence number of the last known acked packet and the next TXS packet on the sender side is larger than `initial_burst_length`. The next TXS packet will not be send because the sender still waits for the ACK of the last TXF packet.

There are different ideas on how to work around such a situation. TXS flagged packets could be added to the filter `ESP_CONNECTION_FLAGS`, excluding special flags from being stopped by the congestion control. This can lead to a problem when many small transfers will be send and the congestion control would need to wait for the acknowledgments according to the original design. The flooding prevention would not work anymore for small packets. A different idea is to ensure that `initial_burst_length` is always larger than `burst_length` to be able to send at least the TXS flagged packet when exactly `burst_length` packets were not acked yet.

A more complex way to deal with that problem is a tree-way handshake with an acknowledgment of TXF flagged packets. This may lead to a lower bandwidth and seems to be too complex for that situation.

No idea was implemented yet and further analysis has to be done.

4 The Architecture of ESPGOAL

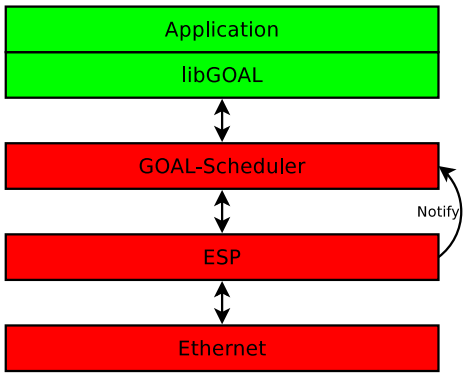


Figure 4.1: ESPGOAL kernelland integration — red are carried out by the kernel module, green ones are performed in userspace.

As shown in Figure 4.1, the implementation of ESPGOAL consists of a userspace library called libGOAL and a kernel module handling the schedule execution and communication with userspace processes and hosts connected over Ethernet. The userspace library implements the previously defined GOAL API which handles the creation of schedules, peer management and interaction with the actual scheduler. The scheduler itself is implemented as kernel module together with a fork of ESP which provides different notifications for the communication management of the ESPGOAL scheduler.

Beside those notification hooks and ethertype, we can consider the ESP transport layer unchanged in comparison to the latest development version of the EDP/ESP kernel module.

4.1 Control Flow

Figure 4.2 outlines the proposed control flow for the ESPGOAL module. Everything in green happens in user space, red items happen in kernel space.

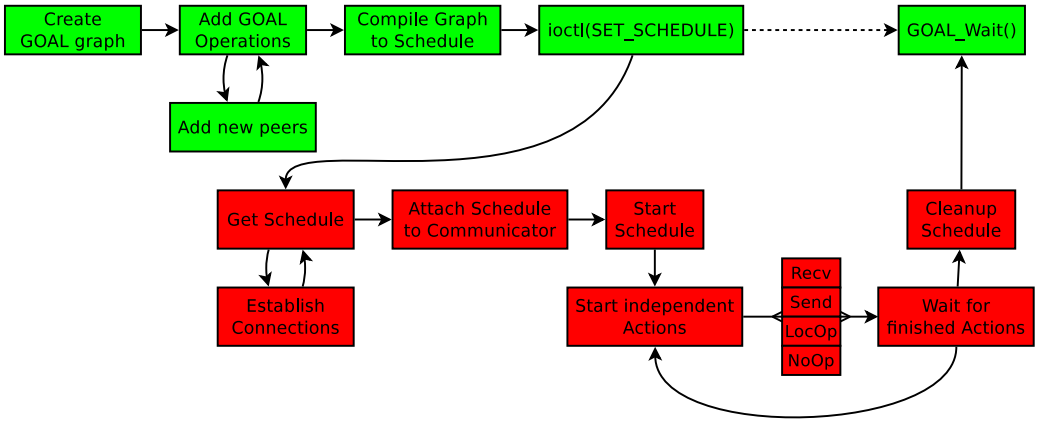


Figure 4.2: Control flow of ESPGOAL. Tasks in red are carried out by the kernel module, green ones are performed in userspace.

4.1.1 Loading the Kernel Module

The scheduler and the communication management are implemented using a kernel module which consists of the ESPGOAL part and ESP as transport layer. Each of those parts has to initialize its own datastructures and add special hooks inside the kernel when the module gets loaded. An example would be the `packet_type` of ESP which is needed to get informed about new arriving packets[1] or `net_proto_family` which is used to add new types of sockets[2]. These two hooks are enough to connect ESP to the network drivers and to provide an interface for user and kernel processes to communicate with it.

There is no general framework for communication schedulers to provide such an interface for the process that wants to start schedules or to get information about the state of the scheduler. A new way of communication between libGOAL and the kernel scheduler had to be developed on top of existing technologies. Possibilities were for example a new syscall or special files in a pseudo file system like `sysfs` or `/dev`. The latter version was chosen because of the relative easy creation of character devices with ioctl handlers in comparison with the manipulation of the syscall tables[3] which would need a larger modification of the kernel itself and could not be done in a separate kernel module.

Beside the interface for user processes, also some mechanisms had to be initialized to be able to use the socket interface inside the kernel. The most important part is the master socket which listens on a predefined ESP port due to the fact that we do not have a sideband

[1] include/Linux/netdevice.h line 1217 and net/core/dev.c line 374 in Linux v2.6.36
[2] include/Linux/net.h line 212 and net/socket.c line 3109 in Linux v2.6.36
[3] arch/x86/include/asm/unistd_64.h and arch/x86/kernel/syscall_64.c in Linux v2.6.36 for amd64

protocol to exchange information about the ports that a specific process uses to start the peer connections. This single master socket needs some extra communication to identify to which communicator/process this new established socket belongs, but solves the problem of the missing sideband protocol quite well for us.

Another big problem is the blocking, non-atomic behavior of the socket API. Even if all operations inside the protocol implementation are not sleeping at all and thus never would start a reschedule, the socket API still enforces that a function from the socket API is never called in an atomic context[4]. This means that everything which gets received by the ESP `net_proto_family` receive hook is in an atomic context as the receiving of data is done in bottom-halves which are currently implemented in Tasklets and thus run in a SoftIRQ context. It is not allowed to call blocking, non-atomic functions inside this context.

It is still possible to create bottom-halves which have a larger execution time. The currently most used API for those asynchronous process execution contexts are Workqueues[5]. We need to create them to start a special worker thread including the work queue which gathers the work items. The worker thread itself provides the schedulable context for all our work items which were enqueued from our atomic context. The worker thread will automatically start to process them when it idles or when it finished another working item.

4.1.2 Adding a Communicator

It must be possible to distinguish different communicators before a schedule can be started. For this purpose `GOAL_Init` will automatically call the ioctl `IOCTL_GOAL_INIT` and provide a communicator identifier which is not already used inside the kernel module. This identifier could be created by an global communicator manager (cf. orted in Open MPI) which is not part of this work.

4.1.3 Starting a Schedule

When a user process calls the library function `GOAL_Run`, it will implicitly perform an ioctl `IOCTL_SET_SCHEDULE`. Figure 4.2 shows that this schedule should be created using libGOAL by adding new operations to an empty GOAL graph. The graph has to

[4] net/core/sock.c line 2001 and include/Linux/hardirq.h line 13-110 in Linux v2.6.36
[5] Documentation/workqueue.txt in Linux v2.6.36

be compiled to its binary representation. It consists of the raw schedule and some extra header information needed for the in-kernel scheduler:

1. size of the schedule
2. communicator identificator
3. schedule number
4. own rank inside communicator
5. amount of peers to add
6. pointer to memory region with peers
7. size of extra temporary buffer for schedules
8. Pointer to memory region with schedule
9. Memory region for notification bit
10. pointer to extra data for the scheduler
11. size of the extra data for the scheduler

The most important extra information is the pointer to peer information. Each added schedule may have peers which were not already registered as communication partners. We must be able to connect to those new peers using a combination of machine identificator and port. ESP uses MAC addresses as machine identificator and our predefined port for the master socket is 80. The MAC addresses were previously identified using libGOAL in userspace and are stored inside the peer information datastructure of our communicator.

Each node may be part of multiple communicators. It must be possible to decide to which communicator an incoming connection belongs. For this purpose, the communicator id has to be send in the first packet after the connection handshake is finished. The receiving node is able to find its communicator with that information and can decide if a connection is still valid or if it has to postpone the connection for later usage.

All further information can be copied from userspace and attached to a new allocated schedule. Also a special memory region is allocated to allow in kernel storage of messages which do not need to be copied to userspace. The kernel now tries to start all independent actions before it return back to the userspace program.

The calling program can either process independent data or call `GOAL_Wait` to block until the kernel module informed the userspace program through the pointer to the notification byte which was copied to the kernel module.

4.1.4 Schedule Progression

The GOAL scheduler starts all currently independent actions. This is done by giving the independent action either to a LocalOp layer which will only return when his computations are finished or to the communication layer that enqueues management data and tries to start a transfer. It does not mean that a transfer is finished when the call to the communication layer functions returns. Instead, the layer itself will explicitly call the scheduler to inform him about a finished transfer.

The scheduler can now remove all outgoing dependency edges from the finished action and start all new independent actions.

4.1.5 Progression by ESP

The ESP sockets inform the ESPGOAL module about either finished transfers or about a discontinued transfer due to full receiving buffers. Both would start the `msg_scheduler` that runs inside a workqueue, if it is not currently running and enqueue a task to get as much data from the receiving buffer as possible. `msg_scheduler` will be explained in detail in Chapter 4.2.

ESP will automatically resume the transfer when the receiving buffer has enough room to receive more packets. This ensures that the progression does not need to be continued by explicit calls to a test function and removes the blocking behavior of sending and receiving functions.

4.1.6 Unloading the Kernel Module

The Linux kernel module handling includes a special mechanism which disallows modules which are currently used from being unloaded[6]. This means that it is impossible to unload the module due to the fact that the module itself opens the listen socket by default which must be closed first to reduce the reference count of the ESP socket layer to zero. A special ioctl instruction was implemented to allow this. It only works when all communicators are freed and all connections to other hosts were closed either properly or by a timeout.

[6] kernel/module.c line 786 in Linux v2.6.36

4.2 Data Structures

4.2.1 Starting a Schedule

During the module initialization, the `master_socket` is created and defines an entry point for new connections. They will be stored automatically by ESP in a connection backlog of the master socket and ESPGOAL can take one connection from this queue at a time. `kernel_accept_correct` in `espgoal_datastructures.c` will ensure that this new connection is the connection we are waiting for or store it in `postponed_socket_list`. It will try to search this list of postponed sockets first to ensure that connection request from the same host do not get processed in the wrong order.

The connecting host will send a new message containing the connection id to ensure that this new connection really belongs to the the current communicator and not to a different one which was not yet created on the current host. This case is currently unimplemented due to the missing communicator identification server (cf. orted in Open MPI).

Afterwards the new socket can be changed to an internal state which allows the ESP layer to call the receiving hook to inform the `msg_scheduler` that it can now read new messages from the receiving buffer. If we would not wait for the state change then it is quite likely that the `msg_scheduler` tries to receive the communicator identificator as a normal message before the socket was verified and attached to the communicator.

The peers itself are managed by `peer_list_t` which is attached to `communicator_t` as shown in Figure 4.3. Each `peer_list_entry_t` represents a single host with all needed information to open a connection to this host and the socket itself when the connection was already established.

All information about a peer is copied from userspace when we receive the `schedule_trans_t` through the `IOCTL_SET_SCHEDULE` handler in `espgoal_devhandler.c` to extend the information stored inside the communicator. The amount of new peers is indicated by `num_peers_in_list` inside this structure.

A new kernel representation `schedule_t` of the userspace schedule will be attached to the communicator identified by `communicator_id` to be able to add more information needed to process the schedule. All other information will be copied from the userspace schedule as seen in Figure 4.3, but memory in userspace has to be copied to the kernel

Figure 4.3: Data structures to manage a communicator with attached schedules and peers used inside the schedules

memory region to be able to access it independent from the running context and without going through the Linux memory manager.

As additional information every `schedule_t` has following elements:

schedule_id Identificator of the schedule inside the kernel which may be different to the `schedule_num`

scratchpad_buffer buffer for temporary GOAL operation results

task Pointer to the task which emitted the ioctl [7]. Needed to access the memory regions without switching the context to this task [8]

pending_actions Amount of started independent actions which are not finished yet

finishing_lock Amount of actions which finished and are eligible to destroy the kernel representation of the schedule and inform the user. Only the last action which accesses the schedule is allowed to do that.

[7] include/asm-generic/current.h in Linux v2.6.36
[8] kernel/sched.c line 2839 in Linux v2.6.36

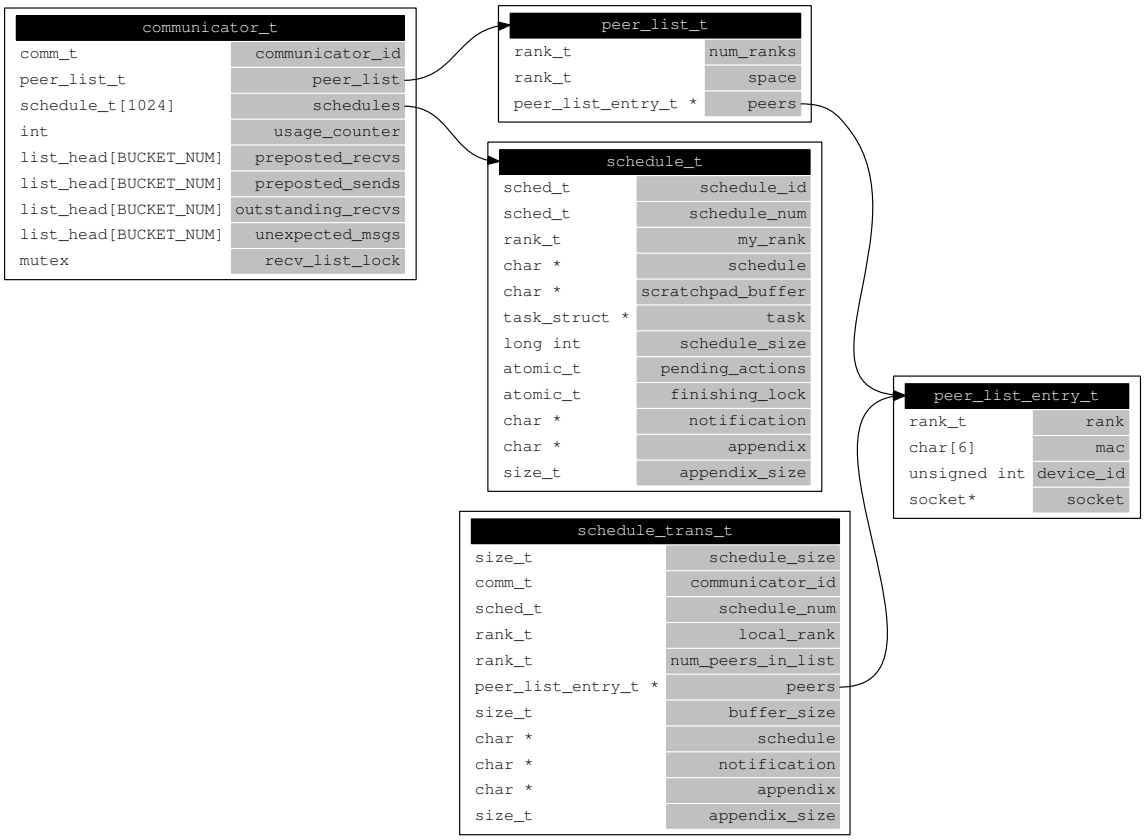


Figure 4.4: Connection management data structures — Binary schedule is inserted into the kernel schedule datastructure and new peers added to the communicator

4.2.2 Transfer Management

ESPGOAL has two separate types of queues to manage currently ongoing non-blocking transfers. Both are implemented in `espgoal_transfer.c`.

The first type handles the transfer of messages to another rank and the second one handles the receiving and matching of incoming transfers. All incoming transfers need extra attention in situations where the receiver has not yet enqueued all information about it. This could happen when not all dependencies for a GOAL receive are finished and thus the GOAL receive is not known to the communication management layer. The ESP transport protocol will automatically inform the GOAL communication management layer about finished messages and full socket buffers. A rendezvous protocol has been implemented for large transfers to reduce the amount of receives without a preposted GOAL receive operation.

Send Management

`send_mgm_task_list` is a global list of messages which have to be transferred or where the scheduler has to be informed that the whole message was passed to the ESP transport layer. Items can be removed and added at different points. Accesses to this list must be serialized using `send_mgm_task_lock`.

New `send_mgm_task_t` items will be created through `_esp_nonblock_send` which is indirectly called by the GOAL scheduler. It holds all data necessary to identify the current status of the transfer and the datastructures which are needed during the transfer as seen in Figure 4.5. Many parts can either be copied from the call to `_esp_nonblock_send` like `commid`, `sched_num`, `req`, `memtype`, `sender`, `receiver` and `tag` or calculated using already known information. For example `sched_id` and `schedptr` can be calculated through functions in `espgoal_datastructures.c`, but are stored inside this structure for easier and faster access.

The storage of the pointer to the userspace or scratchpad buffer depends on the way this function was called. There is a wrapper for sends of a simple memory region called `esp_nonblock_send` and a wrapper `esp_nonblock_sendvec` to send multiple memory regions as a single transfer. Both versions will call the `_esp_nonblock_send` function, but the vector version only receives relative addresses to the extra buffer of the schedule which was called *appendix*. These addresses have to be transformed to global

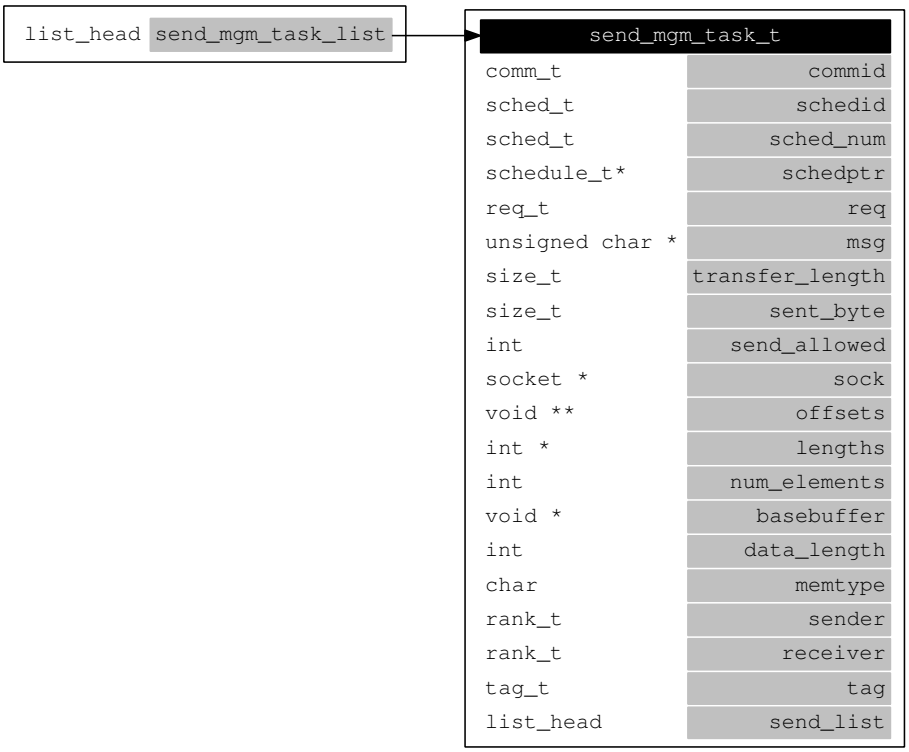


Figure 4.5: Send control data structures — Sends will be progressed until all data is send

addresses inside the kernelland address space before the `_esp_nonblock_send` can work transparently with them.

The biggest problem for the non-vector send function is that either the send functionality had to be duplicated or these `offsets` and `lengths` also have to be used for the non-vector transfer. For that purpose `basebuffer` and `data_length` are used to store the values we got from the GOAL scheduler, `num_elements` is set to 1 and `offsets` and `lengths` are used as simple pointers to `basebuffer` and `data_length`.

To differentiate between local transfers and transfers to other sockets, a look up inside the `peer_list_t` will show us if a socket exists or the destination rank is on the same host. In case we have found a valid socket then we must store a pointer to the socket to start or continue the transfer at a different point.

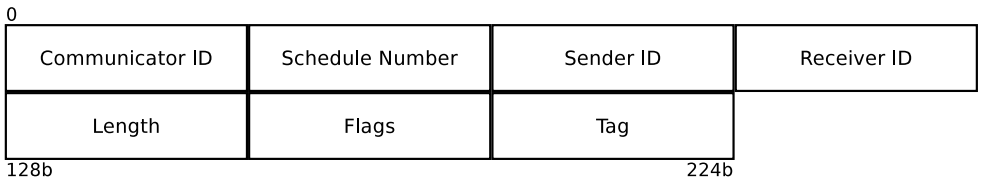


Figure 4.6: GOAL transfer header

The message itself will be attached as `msg`. It consists of a header as shown in Figure 4.6 and the data buffer we should transfer. The header itself provides enough information to identify the correct receive operations on the receiver side. The message chunk is limited to 128 KiB, but larger messages can be transferred by generating multiple 128 KiB chunks which are automatically loaded when the previous chunk was successfully copied to the ESP socket sendbuffer. The amount of sent bytes will be stored as `sent_byte` and after each 128 KiB boundary, the next chunk is loaded from the scratchpad or userspace buffer.

This copying is done without being in the context of the original process that started the schedule. Two special copying functions `copy_to_user_task` and `copy_from_user_task` were implemented in `espgoal_copyuser.c` which work similar like the non-`*_task` versions normally provided by the kernel[9], but takes an additional parameter to define the task which is used to access the correct pages.

[9] arch/x86/lib/usercopy_32.c line 717 in Linux v2.6.36

The scheduler can be informed when $sent\_byte = sizeof(msg\_header) + \sum_{i=0}^{num\_elements-1} lengths_i$. This test must be made quite often and is costly for complex vector sends. Therefore the sum is precalculated when we initialize the `send_mgm_task_t` and store it as `transfer_length`.

We must ensure that no other transfer is currently running on the same socket before the transfer can be started. Otherwise we would interlace two or more different transfers and neither ESP nor our `msg_header` provides enough information to allow multiplexing on top of a single socket. `send_allowed` stores a special state which is `SEND_DISALLOWED` by default. Only when no other `send_mgm_task_t` on the same socket is currently in a different state then it is changed to one of two states which allows sending on this socket. Normally it will be `SEND_ALLOWED` which signals all components that the transfer can be continued. Otherwise it will be `SEND_IN_PROGRESS` to inform other components that it is currently using the socket and has not yet updated `send_mgm_task_t` with the results of the transfer. This allows us to start the transfer directly after we got informed about the send action by the scheduler without waiting for the `msg_scheduler` to find this send task inside the `send_mgm_task_list`.

Receive Management

The second type of transfer management lists are the receiving lists. The most important subtype is the list of sockets which emitted a signal through `ghandle_receive` after they have finished a transfer. At any time two lists of that type exists. One is called `recv_socket_write` and the other one `recv_socket_read`. The write socket queue is only accessed by `ghandle_receive` to enqueue new sockets. The read socket queue is only accessed by the message scheduler to test for new messages in the socket buffer. These lists will be exchanged when the read socket buffer is empty. This makes it possible that the write socket queue must only be protected from multiple concurrent accesses by `ghandle_receive` and from the swapping of the pointers to the read and write queue.

We can see in Figure 4.7 that all transfers will be stored as `unmachined_msg_t` in `unmachined_msg_list`. This list is independent from any communicator or schedule and will get all messages which are detected, but not yet finished. The `msg_scheduler` has to look through the list if there is already an unfinished message on the socket where a transfer was finished or not. At the beginning no unfinished message should be available. We have to peek inside the socket receive buffer to test if enough data (28 Byte) are available for the header or we have to postpone the socket for now. Otherwise it is not possible to add a correctly initialized `unmachined_msg_t` structure to the list. This usually does not happen because the probability that the 28 Byte constituting the message

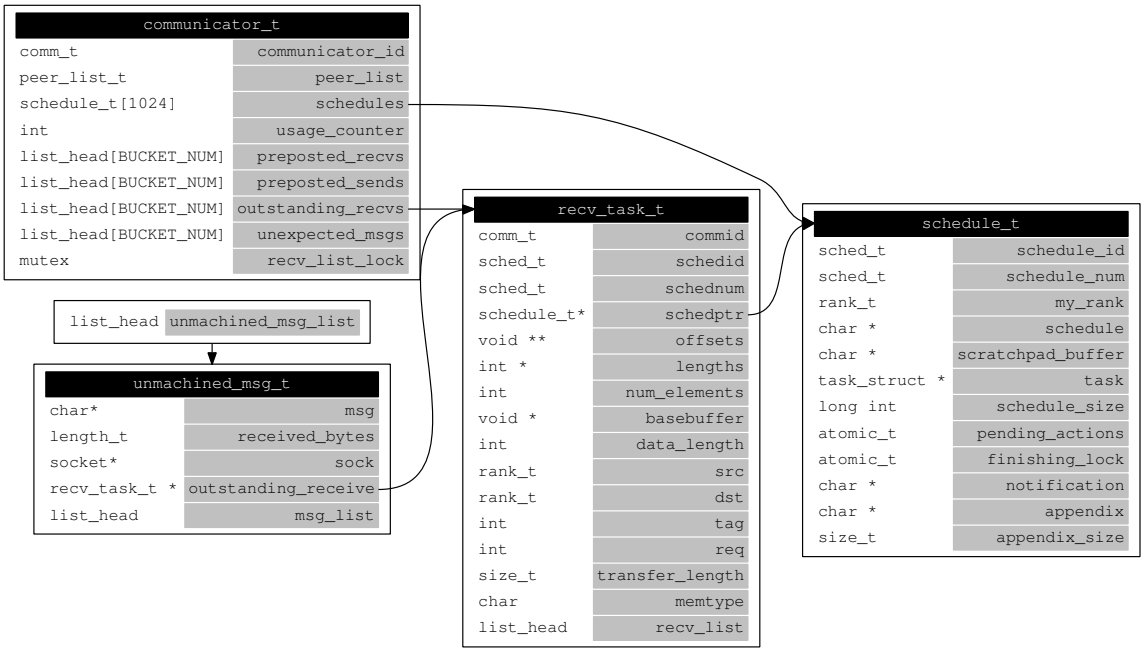


Figure 4.7: Receive control data structures — Message gets received, matched and processed

header are fragmented is small with a standard MTU of 1500 Byte. So normally we can create a message buffer with the size $\min(header_{length}, 128\ KiB)$. Receives bigger than 128 KiB are handled similar to sends larger than 128 KiB. The only difference is that we must assume that the receive was already started by the GOAL scheduler to know where to copy each chunk we received. Otherwise we would need to allocate a new data buffer for each chunk.

The transferred data will be copied to the receive buffer until the amount of received bytes stored in `received_bytes` is equal to $header_{length}$. This makes it possible to interrupt the receive when not enough data is available, but continue the receive when the transfer or part of the transfer was finished.

A finished message cannot be treated as a guarantee that the receive buffer of the socket is now empty. We still need to check if there is at least a header for the next message available. The next socket in the `recv_socket_read` list should only be checked when either not enough data is available to start a new message or to continue the current message. This should keep the amount of data inside the receive buffer low to allow the sender to continue his transfers as fast as possible.

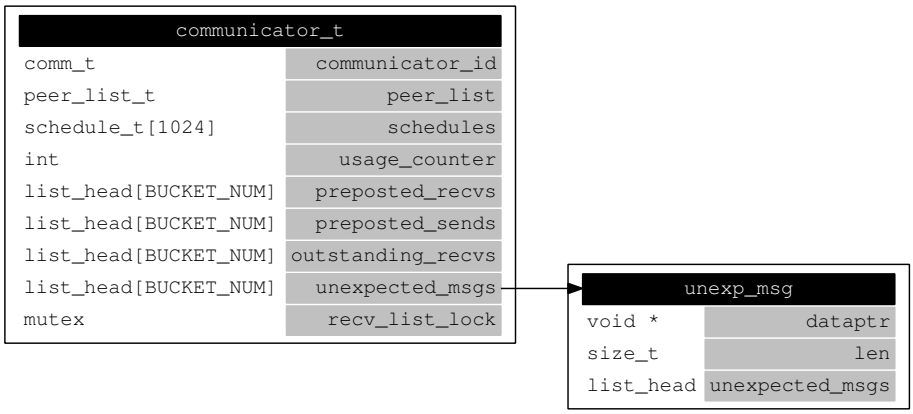


Figure 4.8: Unexpected messages data structures — Messages for not existing schedules/receives will be stored

A finished message has to be matched against all waiting receives attached to the `outstanding_recvs` list inside the communicator. It is not attached to the schedule because a message for a schedule can already reach the receiver before it added the schedule to the communicator. In that situation, it must still be possible to store those messages in a special list called `unexp_msg` as seen in Figure 4.8. If the message was successfully matched against the correct receive then the GOAL scheduler can be informed that his receive was finished and that the new independent actions can be started.

A special exception for this matching rule are transfers bigger than 128 KiB. The message `unmachined_message_t` does not store the header in an extra field. Let us assume for now that we can always find a corresponding entry for the message in the `outstanding_recvs` list and that it will never be an unexpected message. It will be shown subsequently that this is always true for message over the rendevous size limit. It is possible to override the current message chunk after we matched the header against an `outstanding_recvs` when we store a pointer to it inside the `unmachined_message_t`. It is not allowed to tell the scheduler that we already finished the message after we found the `outstanding_recv` because their might be implicit data dependencies between the buffers used in the different actions. The found outstanding receive is still needed to find the receive buffer and it must be used when the message is really copied to the destination buffer to inform the scheduler about the finished action.

The `outstanding_recvs` are created similar to the non-blocking sends, but during the scheduler call to `esp_nonblock_recv` and `esp_nonblock_recvvec`. The data structure must not be created when there is already a matching `unmachined_msg_t`

inside the unexp_msg list. _esp_nonblock_recv can copy the data to the intended memory region, free the unmachined_msg_t and inform the scheduler about the finished action it just started.

When comparing recv_task_t in Figure 4.7 with send_mgm_task_t in Figure 4.5, it is easy to see that both are quite similar and are created using the same information. The biggest difference is the absence of a message buffer and related management information like sent_byte, send_allowed and sock because they are already created as part of unmachined_msg_t and must be known before the GOAL scheduler started the receive. The rest of the information inside recv_task_t is only useful for multiple chunk messages were we assumed that it was possible that we could create a link between those two data structures.

Rendezvous Transfers

We assumed before that the receive was already started when a message over 128 KiB was received by the msg_scheduler. This may not be true with the currently explained handling of messages. A special packet similar to the already known message header in Figure 4.6 is introduced. The flags field will have the bit MSG_PREPOSTED_RECEIVE set and no additional data will follow the header. This message is automatically generated and send when the GOAL scheduler starts a receive larger than 128 KiB. The receiver of this message can now either start the related send or save it until the GOAL scheduler is ready to start it.

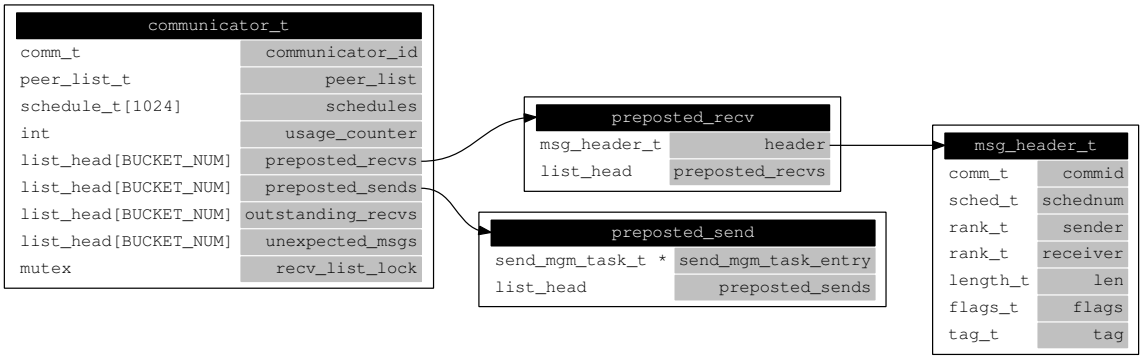


Figure 4.9: Communicator data structures — Large sends will be postponed until the receiver announced that he waits for it

In Figure 4.9 is shown that the sender will store such a header in the list preposted_recv attached to communicator_t which the

_esp_nonblock_send function has to traverse when it is asked to transfer more than 128 KiB large messages. The matching header can be freed and the send_mgm_task_t can be enqueued for transmission. Otherwise the new send_mgm_task_t will be enqueued in preposted_send and the msg_scheduler has to search that list to enqueue the found send_mgm_task_t for transmission.

4.2.3 Stack Overflow Avoidance

The kernel stack attached to processes/threads is currently limited to 8 KiB and can also be reduced to 4 KiB on 32-bit x86 systems [10]. This makes it impossible to have large recursions, but it would be easy to construct them when we assume that the GOAL scheduler would call esp_nonblock_recv, _esp_nonblock_recv can find a matching unexp_msg and it indirectly calls the GOAL scheduler to inform it that a receive was finished.

A schedule could be generated which has a large amount n of receives arranged as chain where the receive r_i depends on the r_{i-1}. The receive r_0 has to depend on a special r_n which is tagged differently in comparison to all other receives. The sender had to create a schedule with the matching sends, but the matching send s_n, for r_n, has to depend on all sends $s_0, .., s_{n-1}$. This would lead to a stack overflow on the receiver side for large n.

To avoid a stack overflow, an extra dynamic queue progress_list was implemented in espgoal_schedule.c to save information about recently finished actions. progress_schedule_on_finished_action will allocate an item which saves communicator id, schedule id and the request id on top of the queue. The counter progress_task_filled is set from -1 to 0 when it tried to save the first item in an empty queue. All other callers will increase the counter to a non-zero value and decrease it again without any further actions beside the enqueuing of their progression item.

That ensures that only the first caller in a call chain dequeues items from progress_list. After this first caller was identified, it has to enter progress_scheduler and take each item out of the queue to inform the scheduler using _progress_schedule_on_finished_action. The GOAL scheduler can now find independent actions and start them. These actions may also finish immediately, but they will also call progress_schedule_on_finished_action, enqueue their progression item and check if they are marked to call _progress_schedule_on_finished_action. They can return safely until the dequeuing loop in _progress_schedule_on_finished_action is reached

[10] arch/x86/Kconfig.debug line 124 in Linux v2.6.36

again. The new progression item will be dequeued and the GOAL scheduler can continue to start new actions. This will continue until enough actions do not finish immediately and the `progress_list` gets empty.

When an action finishes with an empty `progress_list`, the caller of `progress_schedule_on_finished_action` will automatically be selected to handle items in the queue. This should be enough to keep the size of the used stack low and still allows to handle the progression of all finished actions.

4.3 Interpreting a GOAL Schedule

If the application calls the `GOAL_Run()` function to start the execution of a schedule, the ESPGOAL library tries to open a device file associated with the ESPGOAL kernel module. Our kernel module keeps track of the state of the associated device file — if it is already open, any subsequent call to open will fail until the device file is closed again. We do this to ensure that only one process tries to communicate with the kernel module at any given time. If the device file could be opened successfully the ESPGOAL library will transfer the binary schedule and other information associated with it to the kernel module with an ioctl. The argument of the ioctl is a pointer to a datastructure shown in Figure 4.10. After the ioctl is finished the device file is closed again.

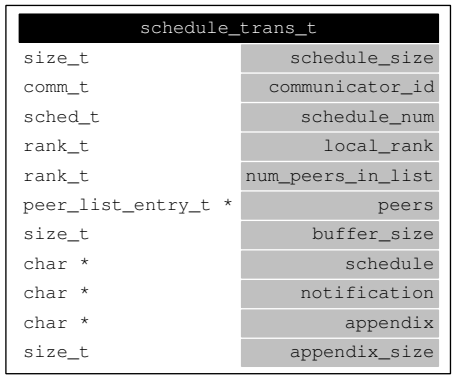


Figure 4.10: The data structure that is transferred from user- to kernelland to start a new schedule

The data which is now accessible by the kernel module contains a list of all peers that the local rank will communicate with in the referenced schedule. The kernel module will check the list of opened connections registered with the referenced communicator and open new connections to those nodes that did not communicate with the local node (in the

context of the referenced communicator) before. The schedule is copied from user- into kernel memory to allow the application to execute it again, possibly while another copy of it is still being executed. The copy operation is necessary as the schedule will be altered during execution as explained later. The datastructure reference by the ioctl also contains a pointer to userspace memory. The memory location referenced by the "notification" pointer is set to a non-zero value as soon as the schedule execution is finished. This enables the ESPGOAL library to efficiently check if a schedule is finished, by simply reading from memory.

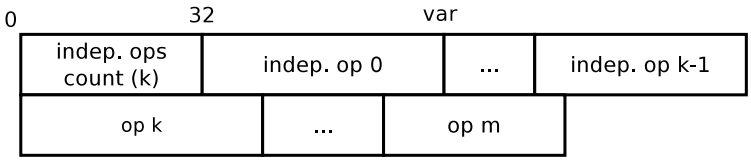


Figure 4.11: Description of the binary schedule representation

If the userspace library indicated that the transferred schedule needs a temporary buffer of a certain size, this buffer is also allocated and registered with the schedule. After that, schedule execution will begin. The transferred schedule is expected to have the form depicted in Figure 4.11. Each operation is encoded differently. Figure 4.12 shows how send operations are represented in the binary schedule.

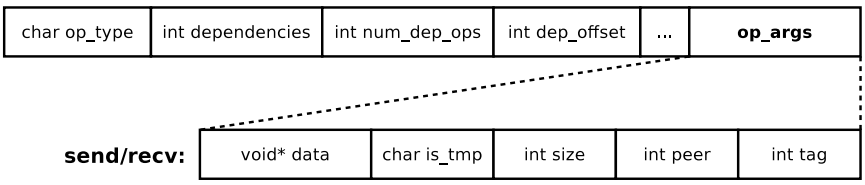


Figure 4.12: Binary encoding of send and recv operations

Each operation has a counter (*dependencies*) that indicates, at any point in time during schedule execution, how many operations have yet to be finished until this operation can be started. This counter initially holds the number of incoming edges of the corresponding node in the dependency DAG. If an operation is finished the dependency counter of each operation referenced in the dependent operations list for the particular operation is decremented by one. If a dependency counter reaches zero the (now startable) operation is enqueued for execution. Operations are indexed by their byte-offset inside the schedule. The beginning of the schedule contains all operations that have no dependencies (their dependency counter is zero), as well as the number of such operations.

If ESPGOAL has finished an operation it calls the scheduler function and passes the binary offset of the operation that was finished. The scheduler will then iterate over the list of

num_dep_ops dependent operations and decrease their dependency counters by one. If a dependency counter is zero, the corresponding operation will be executed.

5 Implementing Collectives in GOAL

In this chapter we will introduce some standard collective functions (i.e., collectives implemented by MPI). We will describe in detail how Open MPI 1.4.2 implements these collectives and show how they can be implemented in GOAL.

Most collective implementations are built upon only few communication schemes. Therefore we will first introduce these abstract schemes. Understanding those is vital for implementing collective functions.

5.1 Recursive Doubling

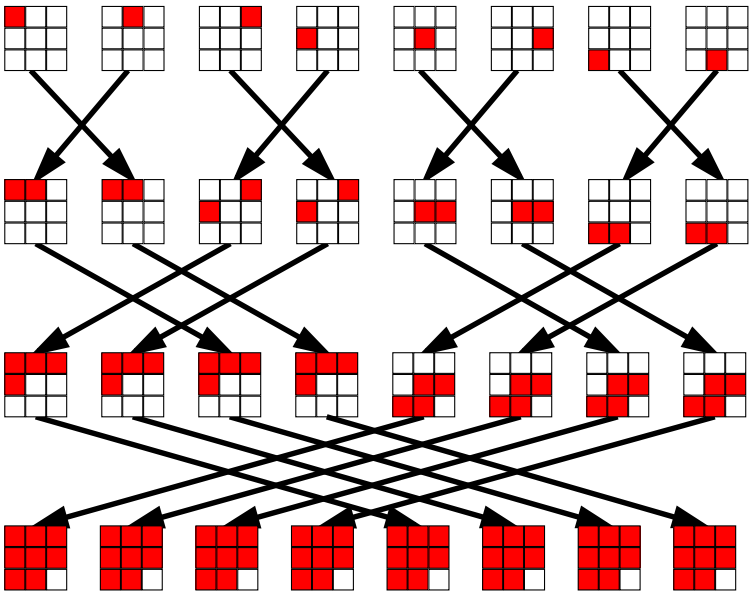


Figure 5.1: Recursive doubling communication scheme. The x-Axis of this picture corresponds to the rank number, the y Axis represents the steps of this algorithm. Red squares represent data that is exchanged.

Recursive doubling is a communication scheme for processor counts that are a power of two. It consists of $\log_2(N)$ steps s. In each step processors i and $i \oplus 2^s$ (where $\oplus$ is the binary exclusive or operation), exchange data. Figure 5.1 is a graphical representation of this communication scheme.

5.2 Bruck's Algorithm

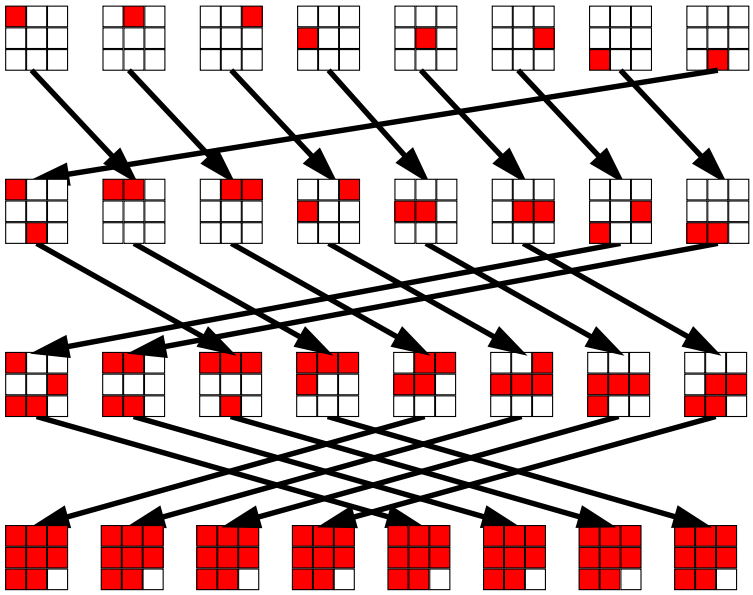


Figure 5.2: Bruck's communication scheme. The x-Axis of this picture corresponds to the rank number, the y Axis represents the steps of this algorithm. Red squares represent data that is exchanged.

Recursive doubling seems to be a good communication scheme for operations like Barrier or Broadcast. Theoretical optimality of course depends on parameters like network latency, bandwidth, routing, switching, congestion control, etc. However, one major drawback of recursive doubling is the "extra step" which is necessary in cases where the processor count is not a power of two. Bruck mitigated this problem by proposing a communication scheme [BHK+02] where every processor sends and receives a message to/from a processor at a specific distance in every communication round. The distance doubles in every round.

In each of the $\log_2 N$ step processors $i - 2^s \mod N$ and $i + 2^s \mod N$ exchange data (in this example $a \mod b$ shall denote the number $x : x \in [0, b-1] \wedge x \equiv a \mod b$. Figure 5.2 is a graphical representation of this communication scheme.

5.3 Binomial Trees

When data has to be broadcasted or gathered from/to one node the naive approach would require $P - 1$ send operation, one for each data transfer from the root to each other node. Unfortunately most networks can not deliver full bandwidth for small message sizes. This is also true for Ethernet networks. One of the reasons is the packet size for the used protocol: For Gigabit Ethernet a frame has to be at least 520 Bytes in length. If the amount of data to be transmitted is smaller, the frame will be padded with zeroes. [Spe08]

So it can be more efficient to coalesce the data for some nodes together, send this data to another node and let him distribute the chunks of data. One way to do this is the usage of binomial trees. Binomial trees can be constructed by the following rules:

- A binomial tree of order 0 (B_0) is a single node
- A binomial tree of order k (with $k > 0$) is a node connected to the roots of the binomial trees $B_{k-1}, B_{k-2}, \ldots, B_0$

If we want to map rank numbers to a binomial tree it is important that we do this in such a way that we do not have to splice the data received from our children if we sent something up the tree — the m children of rank r should have the ranks $r+1, \ldots, r+m$. A binomial tree whose node numbers meet this condition is called an *Inorder Binomial Tree*. An example for such a tree is given in Figure 5.3.

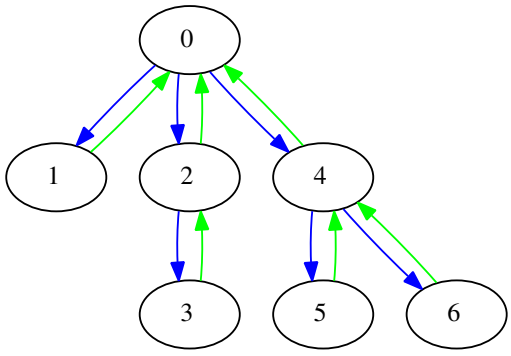


Figure 5.3: Inorder Binomial Tree with 7 nodes and root 0. Blue arrows point to child nodes, green arrows point to father nodes.

To implement collectives with inorder binomial trees it is unnecessary to create a binomial tree in memory — we are only interested in the ranks of our children and our father respectively. These ranks are calculated with the two functions given below.

```
int inorder_bmtree_child(int rank, int size, int num_child) {
  int childs = 0;
  int mask = 1;
  while (mask < size) {
    int remote = rank ^ mask;
    if (remote < rank) break;
    else if (remote < size) {
      if (childs==num_child) return remote;
      childs++;
    }
    mask <<= 1;
  }
  return -1;
}
```

This functions returns the *num_child*th child of rank *rank* in an inorder binomial tree of size *size* with root 0. If no *num_child*th child exists, the return value is negative.

```
int inorder_bmtree_father(int rank, int size) {
  int mask = 1;
  if (0 == rank) return 0;
  while (mask < size) {
    int remote = rank ^ mask;
    if (remote < rank) return remote;
    mask <<= 1;
  }
  return -1;
}
```

This function returns the father node of rank *rank* in an inorder binomial tree of size *size* with root 0.

5.4 MPI_Barrier

Open MPI 1.4.2 by default uses two different barrier implementations. A barrier built upon a recursive doubling communication scheme is used if the communicator size is a power of two. Otherwise a Bruck-based barrier implementation is used.

The recursive doubling communication scheme has been described in Section 5.1 for communicator sizes that are a power of two. For a barrier implementation in GOAL we transmit a one-byte message in each round - instead of exchanging data we are propagating the information on which nodes are already in the barrier. That means that a node may not send a message in each round before it did not receive the message of the previous round. In the cases where the communicator size is not a power of two we determine the smallest power of two which is smaller than the communicator size p, let this number be 2^k. Then we perform the recursive doubling algorithm for the nodes $0 \ldots 2^k$. To ensure that the last $p - 2^k$ nodes also completed the barrier, each of the nodes $0 \ldots p - 2^k - 1$ exchanges a message with one of the nodes $2^k \ldots p - 1$. Care must be taken to ensure that those "additional" messages are not confused with the messages of the recursive doubling algorithm, so we use a different tag for them.

Altogether a GOAL schedule for a recursive doubling based barrier can be created as shown in the listing below:

```
#define GSP GOAL_SCRATCHPAD
int commsizepo2 = pow(2,FLOORLOG(2, commsize));
GOAL_AllocateScratchpad(g, 1);
GOAL_Vertex recv = INVALID_VERTEX;
GOAL_Vertex sendf = INVALID_VERTEX;
GOAL_Vertex sendn, recvn, send;
if (myrank < commsizepo2) {
  uint32_t mask = 0x1;
  while ( mask < commsizepo2 ) {
    uint32_t remote = myrank ^ mask;
    mask <<= 1;
    if (remote >= commsizepo2) continue;
    sendn = GOAL_Send(g, 0, 1, remote, 0, GSP);
    recvn = GOAL_Recv(g, 0, 1, remote, 0, GSP);
    if (recv != INVALID_VERTEX) {
      GOAL_Requires(g, recv, sendn);
      GOAL_requires(g, recv, recvn);
    }
    if (sendf == INVALID_VERTEX) sendf = sendn;
    recv = recvn;
    send = sendn;
  }
}

if (commsize != commsizepo2) {
  GOAL_Vertex sendx, recvx;
  if (myrank < (commsize - commsizepo2)) {
    uint32_t remote = myrank + commsizepo2;
```

```
    sendx = GOAL_Send(g, 0, 1, remote, 1, GSP);
30  recvx = GOAL_Recv(g, 0, 1, remote, 1, GSP);
  }
32 else if (myrank >= commsizepo2) {
    uint32_t remote = myrank - commsizepo2;
34  sendx = GOAL_Send(g, 0, 1, remote, 1, GSP);
    recvx = GOAL_Recv(g, 0, 1, remote, 1, GSP);
36 }
  GOAL_Requires(g, recvx, sendf);
38}
GOAL_Schedule sched = GOAL_Compile(g);
```

A Bruck barrier has also been implemented in GOAL. The Bruck communication scheme is described in Section 5.2. Here we also exchange one byte messages in each round. The GOAL implementation of the Bruck barrier is show in the following listing.

```
1 int myrank = GOAL_MyRank();
  int commsize = GOAL_GroupSize();
3 GOAL_Graph g = GOAL_CreateGraph();
  GOAL_AllocateScratchpad(g, 1);
5
  GOAL_Vertex recvo = GOAL_INVALID_VERTEX;
7 GOAL_Vertex sendo = GOAL_INVALID_VERTEX;

9 for (int distance = 1; distance < commsize; distance <<= 1) {
    uint32_t from = MYMOD(myrank - distance + commsize, commsize);
11    uint32_t to   = MYMOD(myrank + distance, commsize);
    GOAL_Vertex sendn = GOAL_Send(g, 0, 1, to, 0, GOAL_SCRATCHPAD);
13  GOAL_Vertex recvn = GOAL_Recv(g, 0, 1, from, 0, GOAL_SCRATCHPAD
        );
    if (recvo != GOAL_INVALID_VERTEX) GOAL_Requires(g, recvo, sendn
        );
15  if (recvo != GOAL_INVALID_VERTEX) GOAL_Requires(g, recvo, recvn
        );
    recvo = recvn;
17  sendo = sendn;
  }
19
  GOAL_Schedule sched = GOAL_Compile(g);
```

5.5 MPI_Gather

Open MPI 1.4.2 by default uses four different gather implementations. For large data sizes (where the amount of data to be received by root is greater than 6KB) a communication pattern called *linear_sync* is used. This pattern works in the following way:

When the root process enters the gather collective it sends a zero byte synchronization message to all other processes. The other ranks do not send any data before they receive this synchronization message. This technique prohibits "flooding" the root with data, which could lead to back pressure and in turn yield a small bandwidth.

After the processes know that root is ready to receive data they send the first segment of data. The size of this segment depends on the overall amount of data to be received. For datasizes of 92.16 KB and more this first segment is 32.768 KB, for datasizes between 6 KB and 92.16 KB this segment is of size 1024 Byte. After that the rest of the data is sent.

Root will post an MPI_Irecv()[1] for the first segment before it sends the synchronization message to ensure that the data can be immediately processed upon arrival. After the synchronization message is sent, root will issue the MPI_Irecvs() for the second segments and waits for all the first segments to arrive completely with MPI_Wait(). After that root copies his local data from his send to his receive buffer. And issues an MPI_Waitall() to perform a blocking wait on the second segments.

A linear_sync communication scheme has been implemented in GOAL as shown in the following listing:

```
GOAL_Schedule GOAL_linear_gather_sync(void *sbuf, int scount,
    void *rbuf, int rcount, int root, int segsize) {
  int myrank = GOAL_MyRank();
  int commsize = GOAL_GroupSize();
  GOAL_Graph g = GOAL_CreateGraph();

  if (myrank != root) {
    GOAL_Vertex syncmsg = GOAL_Recv(g, ((char*)rbuf), 1, root);
    GOAL_Vertex firstseg = GOAL_Send(g, sbuf, segsize, root);
    GOAL_Vertex rest = GOAL_Send(g, ((char *)sbuf+segsize),
        scount-segsize, root);
    GOAL_Requires(g, firstseg, syncmsg);
```

[1] In the actual implementation of collectives in OpenMPI no MPI_ functions are used because that would break the profiling interface, we list the MPI functions which are equivalent to the internal calls here for clarity

```
    GOAL_Requires(g, rest, firstseg);
  }
  else {
    std::vector<GOAL_Vertex> reqs(commsize-1, GOAL_INVALID_VERTEX);
    for(int p=0; p<commsize; p++) {
      if (p==root) continue;
      reqs[p] = GOAL_Recv(g, ((char*) rbuf)+p*rcount, segsize, p);
      GOAL_Vertex syncmessage = GOAL_Send(g, ((char*) sbuf), 1, p);
      GOAL_Vertex rest = GOAL_Recv(g, ((char*) rbuf)+(p*rcount)+segsize, rcount-segsize, p);
      GOAL_Requires(g, rest, syncmessage);
    }
    GOAL_Vertex cplcl = GOAL_LocalOp(g, sbuf, NULL, rbuf, GOAL_COPY, GOAL_UINT, 8, rcount);
    for (int p=0; p<commsize-1; p++)
      GOAL_Requires(g, cplcl, reqs[p]);
  }

  GOAL_Schedule sched = GOAL_Compile(g);
  GOAL_FreeGraph(g);
  return sched;
}
```

Note that the waitall on the second segments is implicit in GOAL, the schedule will not be marked as finished before all receive operations have completed.

Open MPI uses two variants of the linear_sync pattern: if the amount of data to be gathered on the root is bigger than 90 KiB it uses a initial segment size of 32 KiB. If the datasize is smaller than 90 KiB but exceeds 6 KB the linear_sync pattern is used with a initial segment size of 1 KiB. If the datasize is also smaller than 6 KB Open MPI uses a binomial_gather scheme if more than 60 ranks participate in the gather operation or if the datasize is smaller than 1 KiB and more than 10 ranks participate in the collective. In all other cases a simple linear_gather is used.

The linear gather in OpenMPI is implemented in the following way: All ranks except root send their data to root in a loop, root receives the data and also copies its local data into its receive buffer. We implemented a similar communication scheme in GOAL as shown below:

```
GOAL_Schedule GOAL_linear_gather(void *sbuf, int scount, void *
    rbuf, int rcount, int root) {
  int myrank = GOAL_MyRank();
  int commsize = GOAL_GroupSize();
  GOAL_Graph g = GOAL_CreateGraph();
  /* Everyone sends data (root too because we
   have to copy from send to receive buffer) */
  GOAL_Send(g, sbuf, scount, root);
  if (myrank == root) {
    /* receive from everyone */
    for(int p=0; p<commsize; p++) {
      GOAL_Recv(g, ((char*) rbuf)+p*rcount, rcount, p);
    }
  }
  GOAL_Schedule sched = GOAL_Compile(g);
  GOAL_FreeGraph(g);
  return sched;
}
```

The binomial gather scheme in Open MPI works in the following way: A ordered binomial tree of the same size as the number of ranks in the communicator is constructed (cf. Section 5.3). All leaf nodes send the contents of their send buffer to their predecessor ("father") in the binomial tree. All non-root nodes which have successors ("children") copy their local data into a temporary buffer and also receive the data that their children are sending them into that buffer. This buffer now contains the sendbuffers of all childs. Due to the properties of a *inorder* binomial tree, all childs have consecutive rank numbers. After all receives are finished the temporary buffer is sent to the predecessor in the binomial tree. The root node can receive the data from its children directly into the recv buffer. A similar communication scheme was implemented in GOAL as shown below:

```
GOAL_Schedule GOAL_binomial_tree_gather(void *sbuf, int scount,
    void *rbuf, int rcount, int root) {
  int myrank = GOAL_MyRank();
  int commsize = GOAL_GroupSize();
  int total_recv = 0;
  GOAL_Graph g = GOAL_CreateGraph();
  if (myrank == root) {
    GOAL_Send(g, sbuf, scount, root, 1);
    GOAL_Recv(g, rbuf, rcount, root, 1);
  }
  if (inorder_binomial_tree_child(myrank, commsize, 0, root) !=
      -1) {
    int child;
```

```
    int childnum = 0;
13  std::vector<GOAL_Vertex> reqs;
    if (myrank != root) {
15    GOAL_Send(g, sbuf, scount, myrank, 1);
      reqs.push_back(GOAL_Recv(g, 0, scount, myrank, 1));
17  }
    while ((child = inorder_binomial_tree_child(myrank, commsize,
        childnum, root)) != -1) {
19    int mycount = child - myrank;
      if (mycount > (commsize - child))
21      mycount = commsize - child;
      mycount *= rcount;
23    if (myrank != root) {
        reqs.push_back(GOAL_Recv(g, (void*) (total_recv+scount),
            mycount, child, 2, GOAL_SCRATCHPAD));
25    }
      else {
27      reqs.push_back(GOAL_Recv(g, ((char*)rbuf)+total_recv+
            scount, mycount, child, 2, GOAL_USERSPACE));
      }
29    total_recv += mycount;
      childnum++;
31  }
    if (myrank != root) {
33    int father = inorder_binomial_tree_father(myrank, commsize,
          root);
      GOAL_Vertex send = GOAL_Send(g, 0, total_recv+scount,
          father, 2, GOAL_SCRATCHPAD);
35    for (int i=0; i<reqs.size(); i++) {
        GOAL_Requires(g, reqs[i], send);
37    }
      GOAL_AllocateScratchpad(g, total_recv+scount);
39  }
  }
41  else if (myrank != root) {
    int father = inorder_binomial_tree_father(myrank, commsize,
        root);
43  GOAL_Send(g, sbuf, scount, father, 2);
  }
45  GOAL_Schedule sched = GOAL_Compile(g);
  GOAL_FreeGraph(g);
47  return sched;
}
```

6 Benchmarks

Our GOAL implementation allows us to perform timing measurements in three different ways:

- with GOALs built in GOAL_WTIME operation we can obtain a timestamp during schedule execution
- with the PERFTRACE API built into ESPGOAL it is possible to instrument our code to obtain timestamps at predefined points during the execution of the GOAL interpreter in the kernel and the ESPGOAL communication layer
- using MPIs Wtime() function we can time the userspace parts of our GOAL implementation

In the following sections we will use these possibilities to show that collectives implemented in ESPGOAL have a comparable latency to those implemented in Open MPI when using the ESP BTL. We also demonstrate that simple transformations on local data can be efficiently carried out with ESPGOAL. This is an important feature for the implementation of reductions. We will compare the host overhead of ESPGOAL to that of LibNBC, the only freely available non-blocking collective implementation for generic Ethernet networks known to the authors.

6.1 Testbed

The test system that was used to carry out all benchmarks mentioned in this chapter consisted of 32 nodes of the CHiC cluster [MMHR07].

Each node consists of:

- Two Dual-Core AMD Opteron 2218 Processors

- Two Tigon3 BCM95704A6 rev 2100 network cards
- 4 GiB of RAM

The nodes are connected to a 48 port Gigabit Ethernet Switch (SMC 8848M). These switches theoretically support Jumbo Frames [Dyk99] up to an MTU of 9000, but only if no VLAN is in use. As the CHiC network is configured with VLANs we could not take advantage of this feature and had to set our MTU to 1500. Therefore Open MX has to be configured with the flag `--with-mtu=1500` as it uses a default MTU of 9000 otherwise.

Important software packages used for our benchmarks:

- Linux Kernel 2.6.36
- GNU C Compiler gcc 4.3.2
- Open MPI 1.4.2-1
- Open MX 1.3.2
- NetPIPE 3.7.1
- LibNBC 1.0.1

6.2 Interrupt coalescing parameters

Normally every Ethernet frame that enters or leaves the NIC will notify the CPU / the kernel via an interrupt. This is desirable for small messages, as it ensures messages will propagate as quickly as possible through the network stack and thus exhibit small latencies. However, for large messages that behavior will increase the host-overhead o in the LogGPs model [HSL09a] and decrease the bandwidth and also the overlap potential for large messages.

Interrupt coalescing is a feature of most modern Ethernet NICs [PJD04, Int07]. It allows fine grained configuration of the constrains that have to be met so that the network card will raise an interrupt upon an incoming or outgoing packet. Low overhead Ethernet communication protocols such as ESP or Open MX are especially sensitive towards these pa-

rameters. The Open MX FAQ [1] states in regard to the question how coalescing parameters should be set up:

> **What is the interrupt coalescing impact on Open-MX' performance?**
>
> Most Ethernet drivers use interrupt coalescing to avoid interrupting the host once per incoming packet. While this is good for the throughput, it increase the latency a lot, up to several dozens of microseconds.
>
> To get the best latency for Open-MX, interrupt coalescing should be reduced. The easiest way to do so is to disable it completely.
>
> ```
> $ ethtool -C eth2 rx-usecs 0
> ```
>
> However, it is often better to set it close to the latency so that the observed latency is as optimal while there is still a bit of coalescing for consecutive packets. So, assuming that you observe a N usecs latency with Open-MX when interrupt coalescing is disabled, a nice configuration would to set coalescing to N or N-1 usecs:
>
> ```
> $ ethtool -C eth2 rx-usecs <N-1>
> ```

Our test system, which uses Tigon3 partno(BCM95704A6) rev 2100 cards became unstable when we set rx-usecs to zero as advised in the FAQ. Also the second paragraph of the FAQ seemed dubious — the latency for a message that fits into one frame can not be optimal unless an interrupt is generated immediately after the frames arrival and setting rx-usecs to anything other than zero seems like trading latency for bandwidth, so why should N (or $N-1$) be a good value and not, say, N^2? Also the tg3 driver has more coalescing parameters than just rx-usecs, altogether there are 8 usable parameters: rx-frames, rx-usecs, rx-frames-irq, rx-usecs-irq, tx-frames, tx-usecs, tx-frames-irq and tx-usecs-irq. For an explanation of these parameters consult Figure 6.1.

Note that an interrupt is raised whenever one of the set conditions is met. So if rx-frames is set to 1 the setting of rx-usecs becomes irrelevant for example.

To be able to set these parameters to "good" values to conduct meaningful benchmarks we have to understand how these parameters interact with latency and bandwidth. Note that it is not the goal of this section to find a good latency/bandwidth tradeoff, we solely want to isolate which parameters interact with latency and bandwidth and in what way — the mentioned tradeoffs are likely to be application specific and are only of marginal interest

[1] `http://open-mx.gforge.inria.fr/FAQ/\#perf-intrcoal` on 27. 09. 2010

Name	Range	Description
rx-frames	1-255	how many frames are received before an interrupt is generated
rx-usecs	1-1023	after a frame is received, how many microseconds will be waited before an interrupt is generated
rx-frames-irq	1-255	how many frames are received before the status is updated if interrupts are disabled
rx-usecs-irq	1-1023	after a frame is received, how many microseconds will be waited before the status is updated if interrupts are disabled
tx-frames	1-255	how many frames are transmitted before an interrupt is generated
tx-usecs	1-1023	after a frame is transmitted, how many microseconds will be waited before an interrupt is generated
tx-frames-irq	1-255	how many frames are transmitted before the status is updated if interrupts are disabled
tx-usecs-irq	1-1023	after a frame is transmitted, how many microseconds will be waited before the status is updated if interrupts are disabled

Figure 6.1: Usable interrupt coalescing parameters for the Tigeon 3 Network cards in our test system

when comparing collective implementations that are based on similar protocols with one another.

To measure the impact of the interrupt coalescing parameters on latency and bandwidth we used the ping-pong microbenchmark shipped with Open MX. We wrote a simple client-server test script that selects a random set of parameters on the server, where each parameter is a set to a random number, the random numbers are distributed independent and uniform across the whole value range of each parameter. After that the server sends the parameters for the next run to the client through a TCP socket. Then the chosen parameters are applied on both nodes with the Linux tool `ethtool` and the Open MX ping-pong is initiated. The ping-pong conducts two different measurements: one for a 0 byte message and one for a 1 MB message. Each measurement is repeated 1000 times by mx_pingpong to minimize measurement errors.

To get a visual overview of the data we plotted the results of 250 of such test runs in the following manner: Each parameter is graphed against the 0B and the 10MB latency. This produces the 16 plots shown in Figure 6.2.

In that figure we can identify four plots which exhibit some kind of structure visible by the “naked eye”: For the latency of small messages the rx-usecs parameter seems to be the

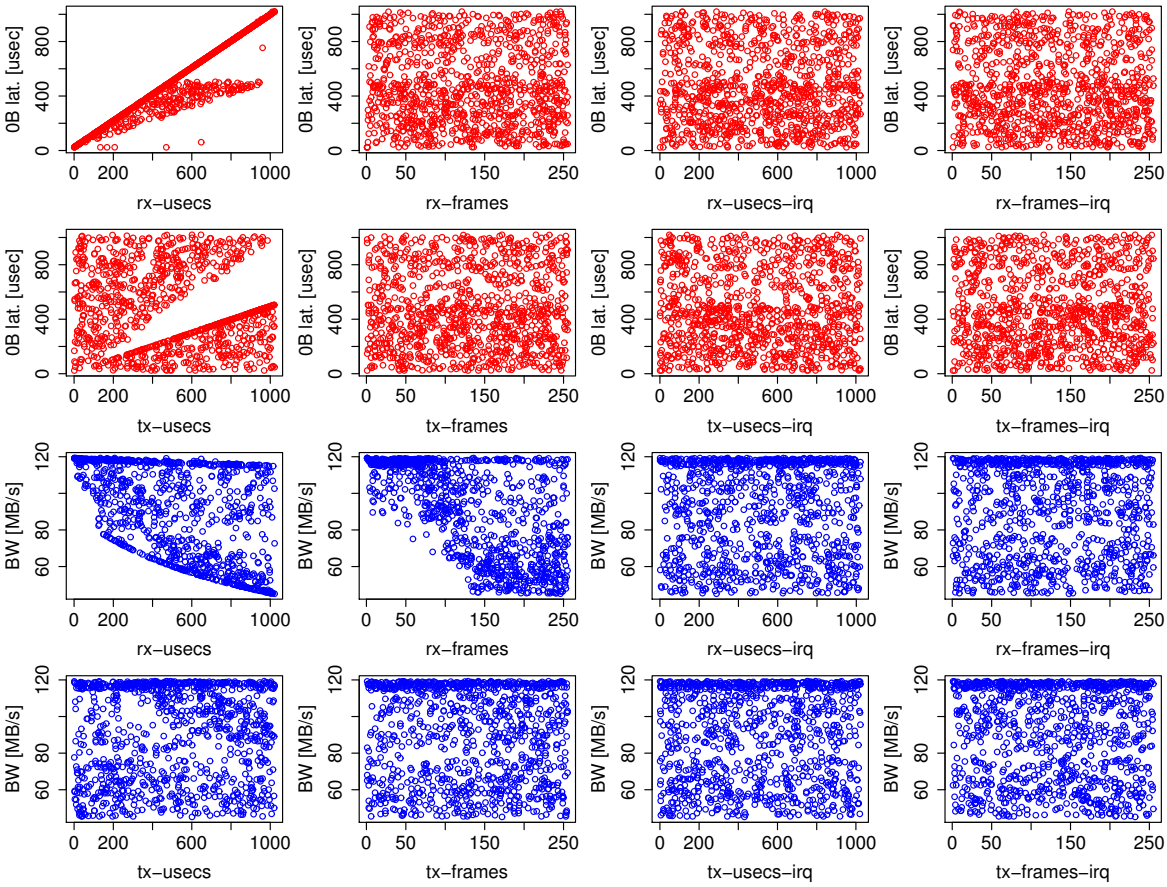


Figure 6.2: Influence of the different interrupt coalescing parameters on latency and bandwidth. Some parameters, such as rx-usecs, obviously have a big impact on the network performance, while others seem completely unrelated.

most important one, its value correlates almost perfectly with the latency. This is what we expected: if rx-usecs is set to X microseconds on both hosts and we send a packet back and forth the measured round-trip time will be $rtt \geq 2 * t_{net-latency} + 2 * X$ as both hosts have to wait X microseconds before the NIC will notify the kernel for the incoming packet. The only other parameter that could mitigate this is rx-frames if we are lucky enough that the packet we are waiting on is the Y'th packet since the last interrupt and rx-frames is set to Y. But the probability for this to happen is rather small, especially on both hosts simultaneously. However this effect could be an explanation for the few dots below the diagonal in the upper left plot.

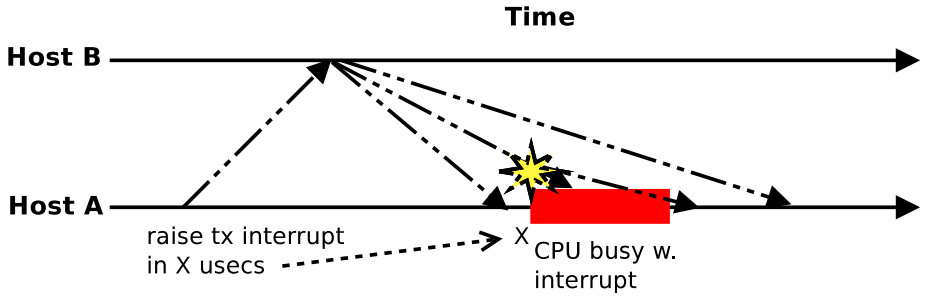


Figure 6.3: Explanation for the gap in the tx-usecs vs. latency plot: For any value of tx-usecs we can not measure round trip times in that range because the CPU will be busy servicing the interrupt

A plausible explanation for "gap" in the tx-usecs / 0-byte-latency plot is depicted in Figure 6.3: If we set tx-usecs to the value X and send a ping from A to B, A can not receive the message in the time from X to $X + t_{int}$, where t_{int} is the time needed to service the raised interrupt. So if the round trip time was in that range A will measure a $RTT \geq X + t_{int}$ and therefore a higher latency will be reported. Messages that arrive before or after that interval can be processed normally.

For the bandwidth of 10 MB messages we can see that, contrary to the Open MX FAQ, a small rx-usecs value does not have any negative effects on the bandwidth (it might have a negative effect on the overlap potential of collective operations but that is not the focus of this chapter). Also, other than for the zero byte case, a small value for rx-frames now also helps with achieving a good bandwidth.

From our visualization in Figure 6.2 we can not obtain any more information about the other coalescing parameters. Therefore we tried to model the influence of all the coalescing parameters on latency and bandwidth with a linear model [WR73]

$$Y = \beta_0 + \sum_{i=1}^{n} \beta_i X_i + \epsilon$$

where Y is a *response variable*, i.e., latency in our case. The X_i are the available *predictors* whose effect on the response we want to assess in terms of the corresponding β_i coefficients. If we fit this model to our data using the least-squares method we get the following models:

$Y =$	**0B latency**	**1MB bandwidth**
β_0	71.3273936	1.370e+02
$\beta_{rx-usecs}$	0.9170643	-4.591e-02
$\beta_{rx-frames}$	0.0148644	-2.168e-01
$\beta_{rx-usecs-irq}$	0.0092267	-1.064e-03
$\beta_{rx-frames-irq}$	-0.0138306	1.056e-02
$\beta_{tx-usecs}$	-0.1213577	5.595e-03
$\beta_{tx-frames}$	-0.0057134	-5.291e-03
$\beta_{tx-usecs-irq}$	0.0003548	3.660e-03
$\beta_{tx-frames-irq}$	-0.0199030	-6.843e-04
Adjusted R^2	0.9109	0.7019

The R^2 value is an indicator for how well the model fits the data. A R^2 value of 1 means that all data points lie on the curve implied by the model. The green color in the table means that for these values we can say with certainty (significance-level $\alpha = 0.001$) that these values have to be different from 0 and thus the respective parameter must have a significant impact on the latency/bandwidth.

Our analysis so far suggests to select a small value for rx-usecs and rx-frames and a large value for tx-usecs. But we can not make a good assumption about the other parameters. Either because they have no measurable effect on the latency and bandwidth or because their interactions with each other are too complicated for our linear model. To assert that the latter is not the case we tried heuristic searches in the parameter space, namely Hill-Climbing and Genetic Optimization [Sch77].

Hill-Climbing did not yield useful results in our case, because our search space is "flat" in some regions, all neighbor configurations will result in measurements that differ only within the error margin of the measurements. In such cases Hill-Climbing algorithms wander through the search space aimlessly.

Our genetic algorithm initializes the starting population with 500 random parameter sets where each parameter is randomly chosen from the allowed value range with uniform probability. After that we are selecting a pair of individuals (parameter sets) from the population. For that we use roulette wheel selection

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$$

where the probability p_i for the selection of the individual i is proportional to its fitness f_i, relative to the rest of the population. The fitness of a individual is the inverse of the latency when we optimize for latency or the bandwidth if we optimize for bandwidth. Each selected pair is merged into a new individual with uniform crossover. After crossover we apply a (uniform) random value $\in [-50, 50]$ to each parameter with the mutation probability $m = 0.15$.

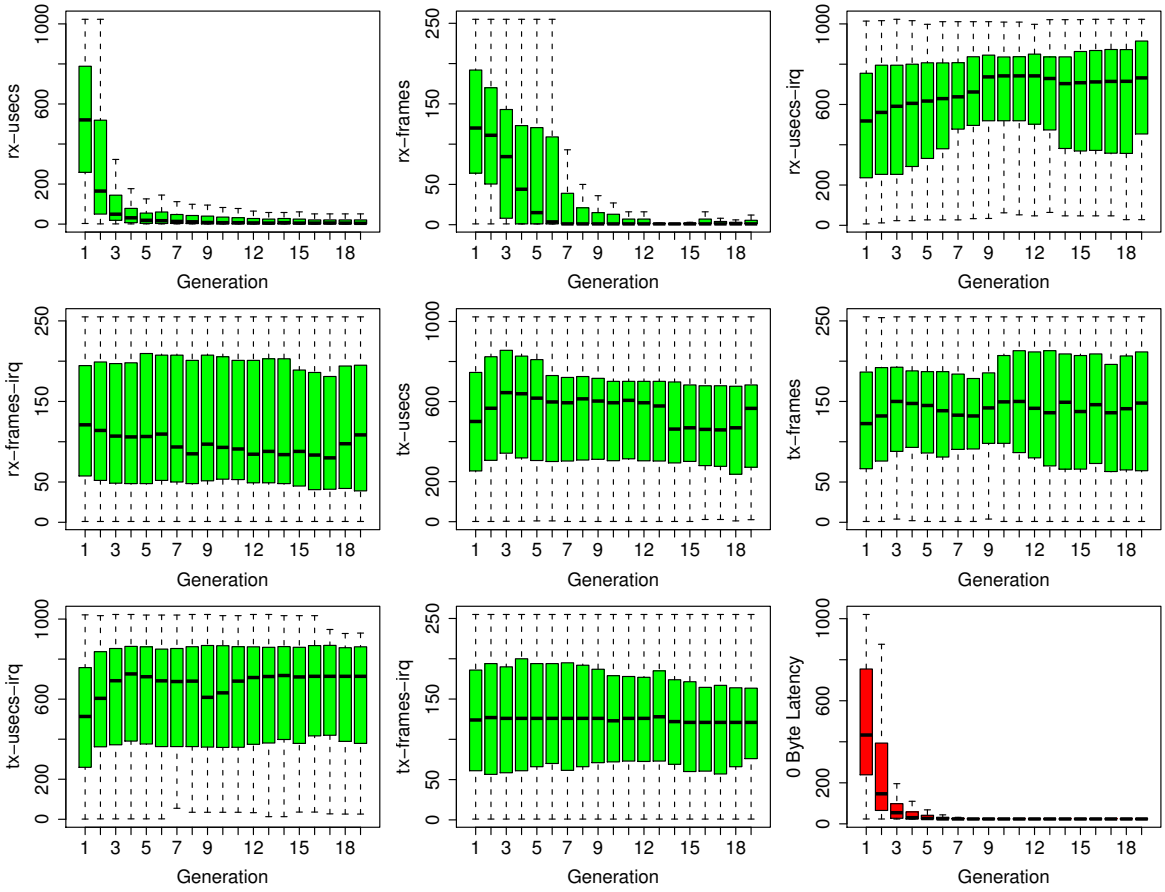


Figure 6.4: Boxplots for the coalescing parameters and the resulting latency over the first 30 generations produced by our genetic algorithm

As shown in Figure 6.4 the latency converges quickly towards a good value. The **best** parameter set in the last of the simulated 50 generations gives a 0b-latency of 22.930 µs with the following parameters:

rx-usecs	**rx-frames**	**rx-usecs-irq**	**rx-frames-irq**
1	1	996	95
tx-usecs	**tx-frames**	**tx-usecs-irq**	**tx-frames-irq**
32	94	724	128

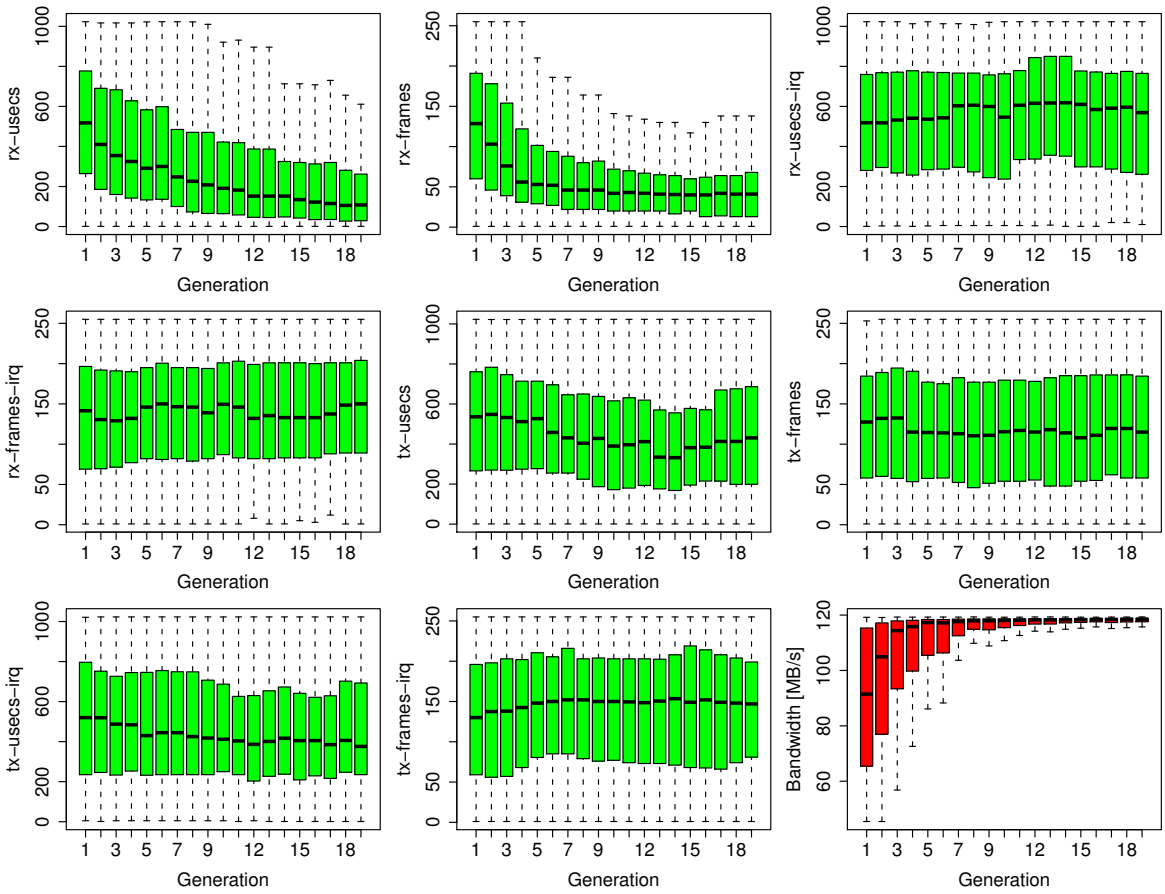


Figure 6.5: Boxplots for the coalescing parameters and the resulting bandwidth for a 10 MByte message over the first 30 generations produced by our genetic algorithm

As shown in Figure 6.5 the bandwidth converges quickly towards a good value. The best parameter set in the last of the simulated 50 generations gives a bandwidth of 119.35 MB/s with the following parameters:

rx-usecs	**rx-frames**	**rx-usecs-irq**	**rx-frames-irq**
47	1	256	243
tx-usecs	**tx-frames**	**tx-usecs-irq**	**tx-frames-irq**
9	52	640	112

Unsurprisingly the parameters resulting from the genetic optimization for latency differ from the optimization for bandwidth. Luckily the bandwidth is nearly the same (statistically the difference is insignificant) for both parameter sets, as shown in Figure 6.6. Therefore we will use the parameter set which is optimal for the latency in all the following benchmarks.

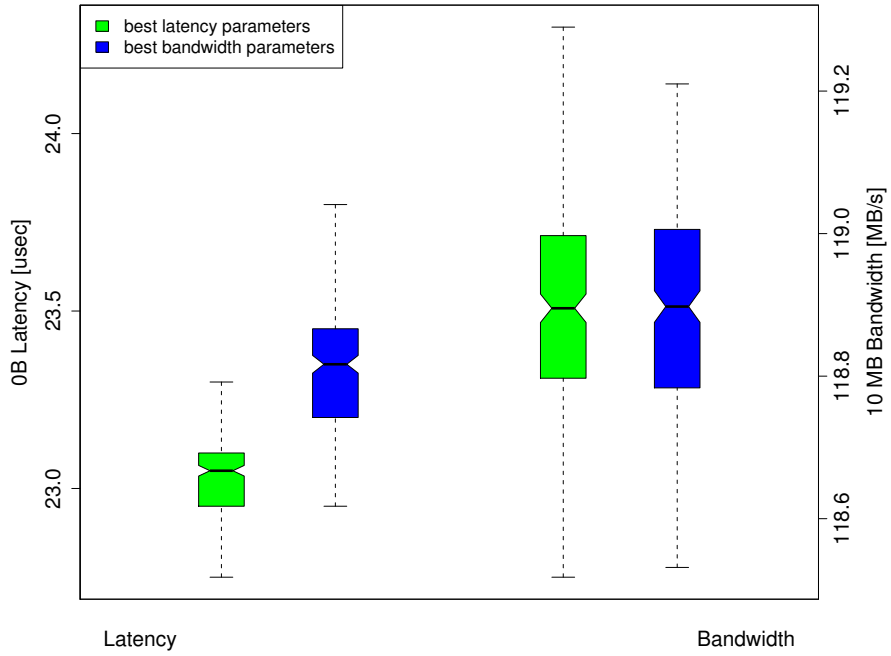


Figure 6.6: Boxplots of 250 measurements for the latency and bandwidth of both parameter sets given in the tables above, one resulting from the genetic optimization for bandwidth and the other for latency. Note that the bandwidth is nearly identical for both choices. This allows choosing the latency-optimal set for all following benchmarks.

6.3 Benchmarking Point to Point Latency

In the following sections we will benchmark collective communication implemented on top of various network protocols. To understand why certain implementations might be slower than others it is vital to know how the used protocols compare to each other in terms of point to point latency and bandwidth. To gather this information we used NetPIPE [SMG96] to compare the latency and bandwidth exhibited by the Open MPI Byte Transport Layer (*BTL*) for TCP, ESP and OpenMX. The latency results can be seen in Figure 6.7 and the results for the bandwidth comparison is graphed in Figure 6.8.

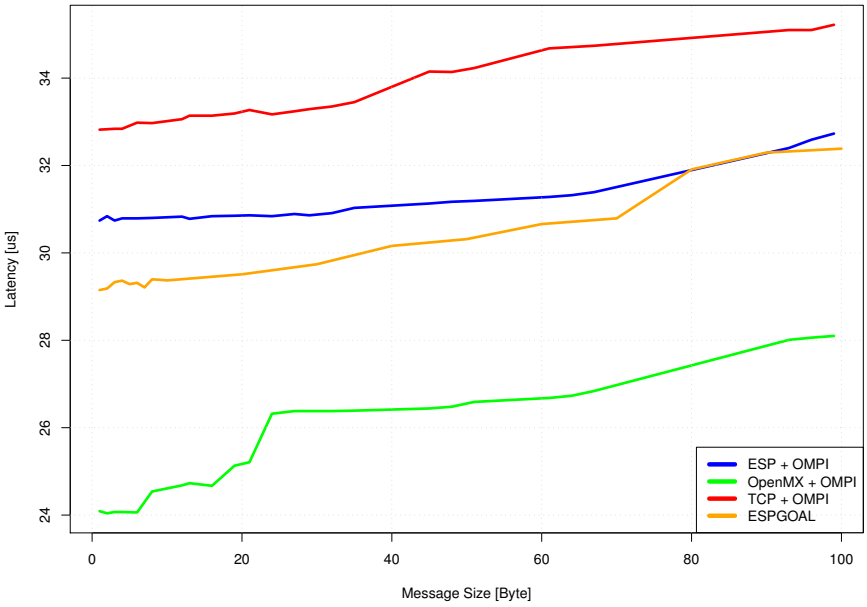


Figure 6.7: Comparison of different network protocols latency

Additionally to the MPI BTL performance data these graphs also show the latency and bandwidth of ESPGOAL. Since there is no MPI BTL for ESPGOAL (which would not make sense, since ESPGOAL was designed to implement collective communication, where MPI BTLs provide an abstraction for point to point messaging), ESPGOAL could not be benchmarked with NetPIPE or similar tools. Therefore we used the ESPGOAL to created a schedule which started and ended with a GOAL_WTIME operation and contained 100 ping pong rounds between two nodes in between. The GOAL_SEND in each ping pong round depends on the previous receive (or the first GOAL_WTIME if there was no previous GOAL_RECV). The second GOAL_WTIME operation depends on the receive operation in the last ping pong round.

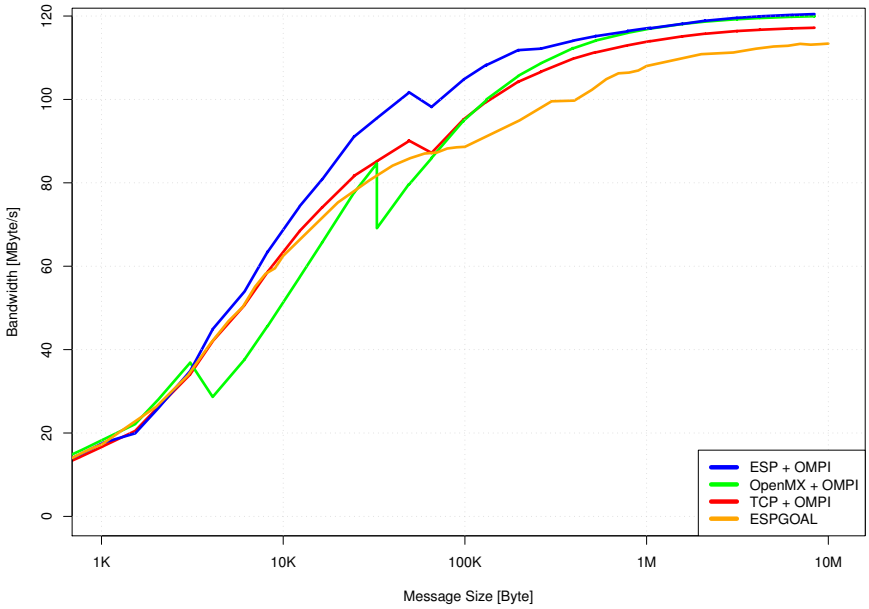


Figure 6.8: Comparison of different network protocols bandwidth

After the schedule is executed we can calculate the one way latency (under the assumption that we have symmetrical links) with $t_{lat} = \frac{t_{end} - t_{start}}{2 * rounds}$, where $rounds$ is the number of ping pong rounds performed in our benchmark (100 in our case).

6.4 Benchmarking Local Operations

To benchmark the performance of local operations which operate on userspace memory we implemented a set of test programs which generate a single schedule which performs a local operation on random data, run this schedule and wait for its completion. We measure the time it takes to execute GOAL_Run() and GOAL_Wait() with MPI_Wtime(). After that we perform the same operation in a single loop in the test program.

Since we are operating on data in userspace the GOAL interpreter has to copy the data in and out of kernelland in a blockwise fashion. These copy operations are not done for the "userspace" case in our benchmarks. Also we compiled the test program with -O3 optimization (because this resembles how a MPI program would probably be compiled), while our GOAL kernel module has to be compiled with -O2 (because -O3 could break compatibility with the rest of the kernel). Therefore it has to be expected that local operations are slower then equivalent computations in a userspace program. When adding

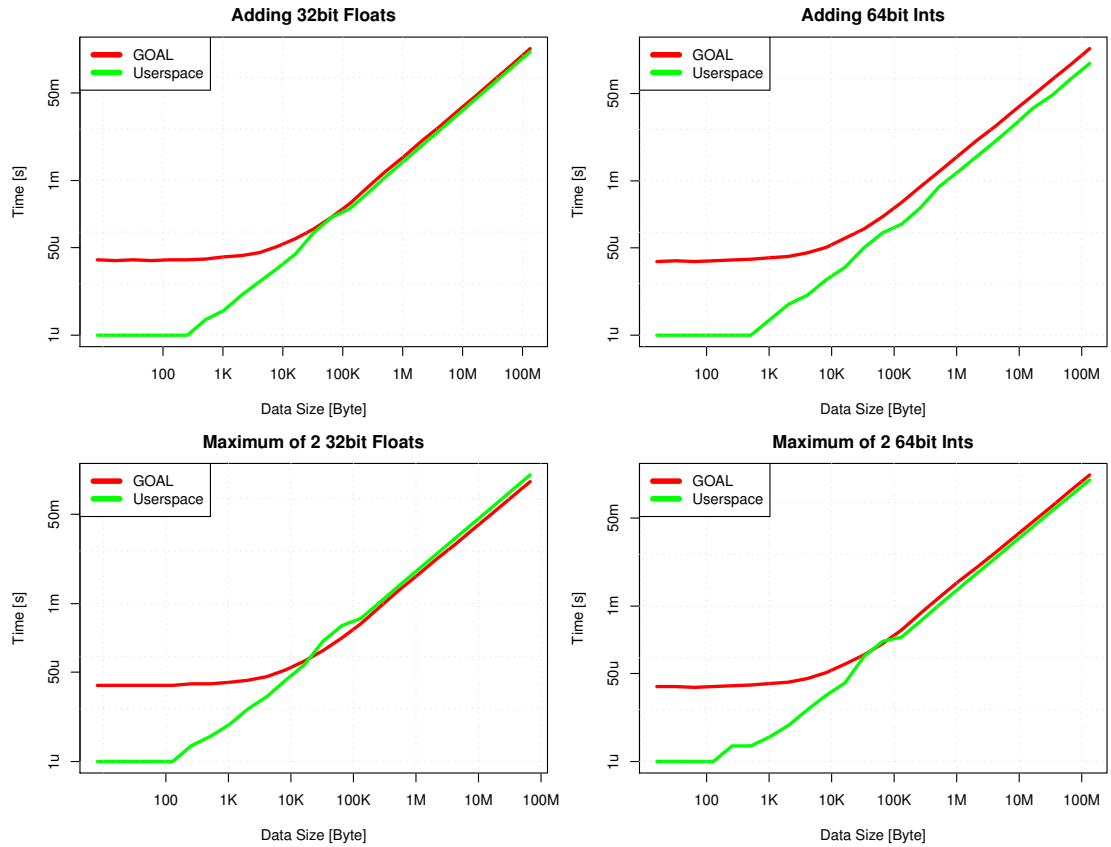


Figure 6.9: Speed comparison of the execution of a schedule performing only arithmetic operations and the same operations done in a normal C++ program

unsigned 64bit integers this performance difference becomes most apparent, and is surprisingly small when dealing with floating point numbers.

In all figures one can also observe the overhead of starting the GOAL schedule. Note that we measured the runtime of the schedule execution in the userspace program, so the starting overhead also includes the time needed by the Linux kernel to perform the context switch.

All benchmarks shown above operate on data in userspace memory. This is not representative for collective communication patterns, as for example for an allreduce operation, most data does not have to be written and operated on in userspace but can directly be received and reduced in the scratchpad buffer. Nevertheless, the benchmarks in Figure 6.9 show that it is possible to perform arithmetic operations in the kernel in an efficient manner.

6.5 Benchmarking Collective Communication Latency

In this section we will analyze the latency of some collective communication patterns. Benchmarking collective communication is a challenging task because a collective is (usually) a global operation across many nodes but in most cluster computers in use today there is no global clock available. That means we can only measure local time differences on each node. Also this makes it hard to synchronize all nodes so that they will start to execute the collective simultaneously.

These fact led to the development of many similar but subtly different benchmarking schemes. A good overview over the most common ones and also the most common mistakes and misconceptions in conjunction with benchmarking collective operations is provided in [HSL10] and [HSL08].

Our benchmarks were carried out in the following way:

```
...
// First create a schedule for the collective
// we want to benchmark

GOAL_Schedule sched = GOAL_bruck_barrier();

// Execute it numtest times and measure the time
// for GOAL_Run() + GOAL_Wait() on each node
```

```
10 for(cnt = 0; cnt < numtests; cnt++) {
     t1s = MPI_Wtime();
12   test = cnt;
     GOAL_Handle runh = GOAL_Run(sched);
14   GOAL_Wait(runh);
     t1e = MPI_Wtime();
16   goal_times[cnt] = t1e-t1s;
   }
18
   // Collect the timing from each node for output
20
   MPI_Gather(goal_times, ... MPI_COMM_WORLD);
22 ...
```

In the cases where we benchmarked collective that does not synchronize across all nodes, such as Gather, we performed an MPI_Barrier() before taking the timestamp `t1s` to minimize pipelining effects. We used the same scheme to benchmark MPI collectives. Since MPI collectives are blocking we did not have to call a Wait() function in that case, the MPI_* function was called instead of the GOAL_Run() function.

If we ran the benchmark shown above n times for a collective across p nodes we get $n \times p$ local time differences. First we calculate the sum of the p times obtained by each benchmark run and divide this value by p. In our graphs we always plotted the median value of the results. This is a meaningful value because it represents the average time per rank that was consumed by the collective communication.

Figure 6.10 shows the result of the described collective benchmark for the Barrier collective.

Unsurprisingly the Open MX BTL outperforms all other benchmarked BTLs and also ESPGOAL. ESPGOAL is only slightly faster than the ETH-BTL using ESP.

If we compare the latencies of binomial tree based gather implementations, as shown in Figure 6.11 we can see that the "startup overhead" that was already evident in the local operations benchmarks (cf. Section 6.4) limits ESPGOALs performance in situations where the majority of the communicating nodes only spend a small amount of time in the collective communication function.

A comparison between different implementations of the linear sync based gather with an initial segment size of 32 KiB and a message size of 500 KiB is shown in Figure 6.12. ESPGOAL outperforms all other transports in this case because we use non-blocking sends

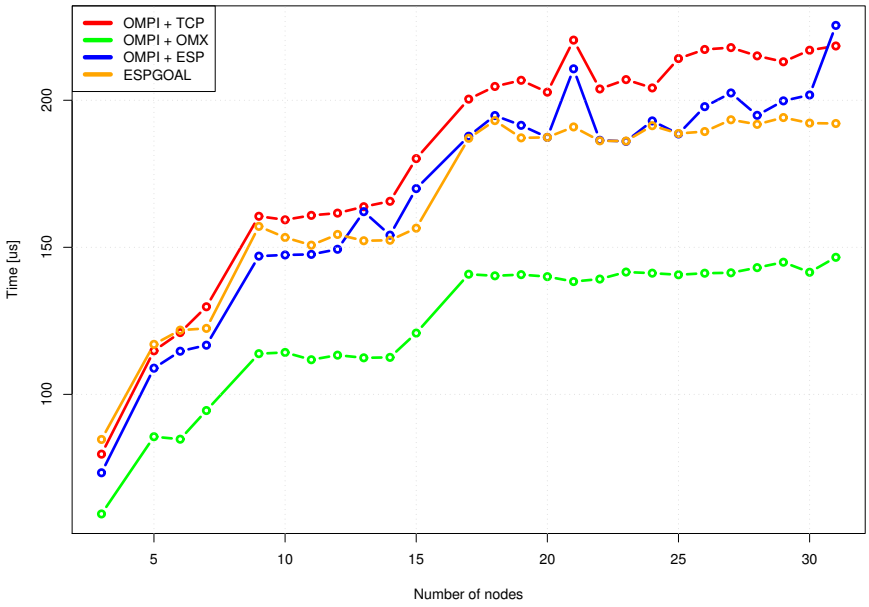


Figure 6.10: Bruck Barrier implemented in ESPGOAL vs. Open MPI barrier using different BTLs

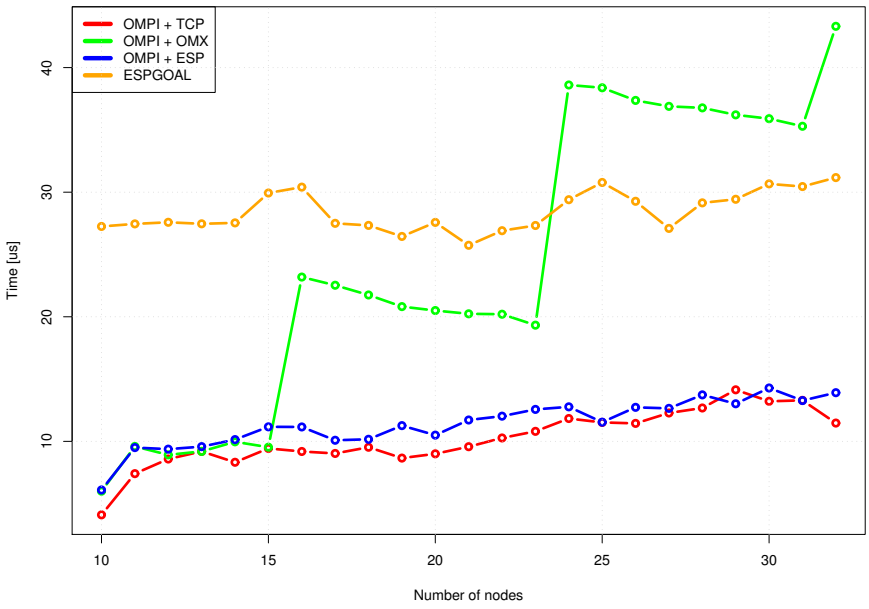


Figure 6.11: A comparison between different Implementations of the Binomial Tree based Gather with a message size of 512 B. The startup overhead dominates the runtime for ESPGOAL.

(we only block until we get the initial sync packet). Open MX was omitted from this plot due to a performance bug in conjunction with large data sizes.

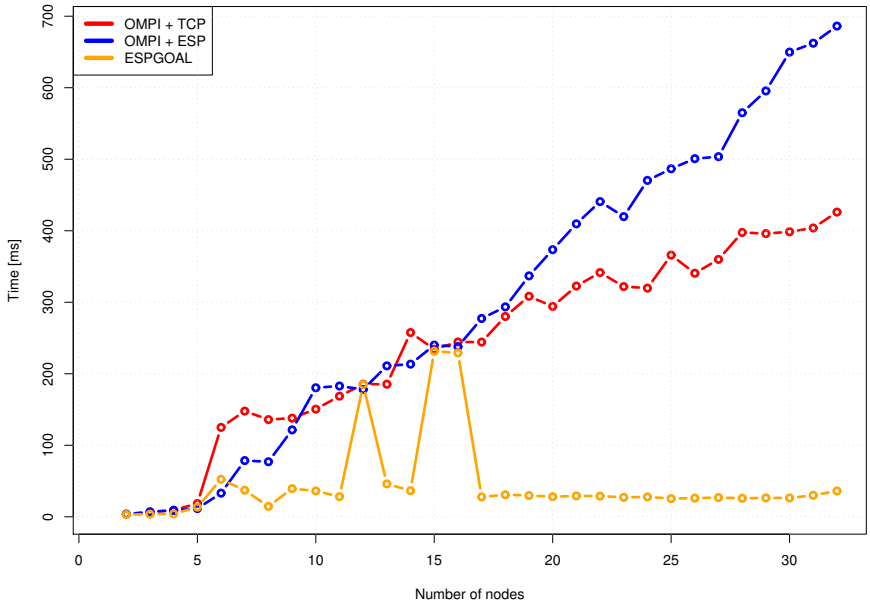


Figure 6.12: Linear sync gather implemented in ESPGOAL vs. Open MPI gather using different BTLs

6.6 Benchmarking Collective Communication Host Overhead

The purpose of non-blocking collectives is to overlap the latency of the collective communication with computation. Applications which do this are typically written in the form

```
app_buffer active, inactive;
communication_handle coll;
while (!solved) {
  coll = nonblock_comm(incative);
  do_computations(active);
  blocking_wait(comm);
  exchange active and inactive buffer;
}
```

In this scenario there are several important factors that influence the overall performance:

1. The computation has to be long enough to overlap the entire collective communication. Otherwise time is wasted in the blocking wait.

2. The collective communication library either needs to ensure implicit progress (the collective will continue to execute in the "background") or the do_computation() function has to make calls to a test/progress function provided by the non-blocking collective implementation to ensure progress. Otherwise the collective will be executed in the blocking_wait() call and therefore is not actually overlapped with computation. Note that doing calls to progression functions might not always be possible, for example if the computation consists of a single call to an external library.

3. The execution and progression of the collective should not use to many CPU cycles since that would delay the computation. For the same reason the execution and progression of the collective should keep cache pollution to a minimum.

To compare different non-blocking collective implementations with regards to these important factors we devised a benchmark that overlaps a collective communication with a $N \times N$ matrix multiplication. In the case where the collective communication implementation does not guarantee progress without regular calls to a test()-function (*implicit progress*) it is vital that the computation can be split into several, preferably equal, pieces of work. We chose a $N \times N$ matrix multiplication as a computation workload because it shows exactly that property: A multiplication of two $N \times N$ matrices can be decomposed into m^3 smaller $k \times k$ multiplication iff. $\exists m : N = mk$ [Str69].

After we chose a block-size k we can issue a call to a communication progression function. In our benchmarks we chose a block size of 32 and 4 Byte Floating point values as data elements. We store the matrices in the block data layout [PHP03] to achieve good cache locality, as we want to observe the impact of cache pollution. The outer loop of the matrix multiplication is parallelized with Open MP so that we can utilize all available cores in the system, otherwise the communication could use CPU cycles (from an idle core) but it would not have a measurable impact on the computation.

In our benchmark we measure the time in the matrix multiplication with and without a collective communication running. The start overhead of the collective function is added to this time. Each measurement where a collective is running is done twice, once with calls to the respective progress function after each $k \times k$ multiplication (explicit progress) and once without (implicit progress). After each matrix multiplication which was overlapped by a collective function we perform a blocking wait for the collective to finish. The time used for the blocking wait is also measured.

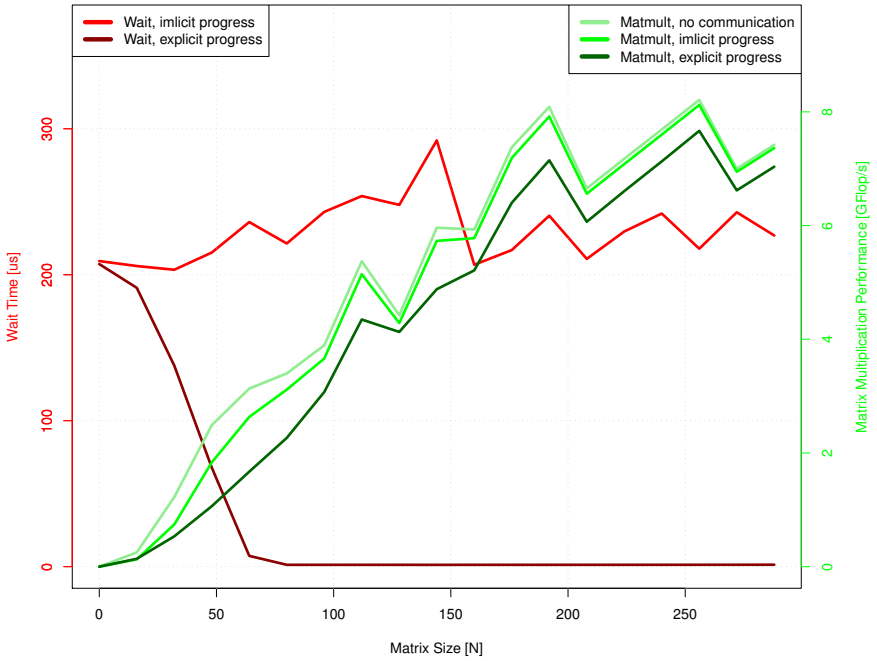


Figure 6.13: NBC Overhead Benchmark

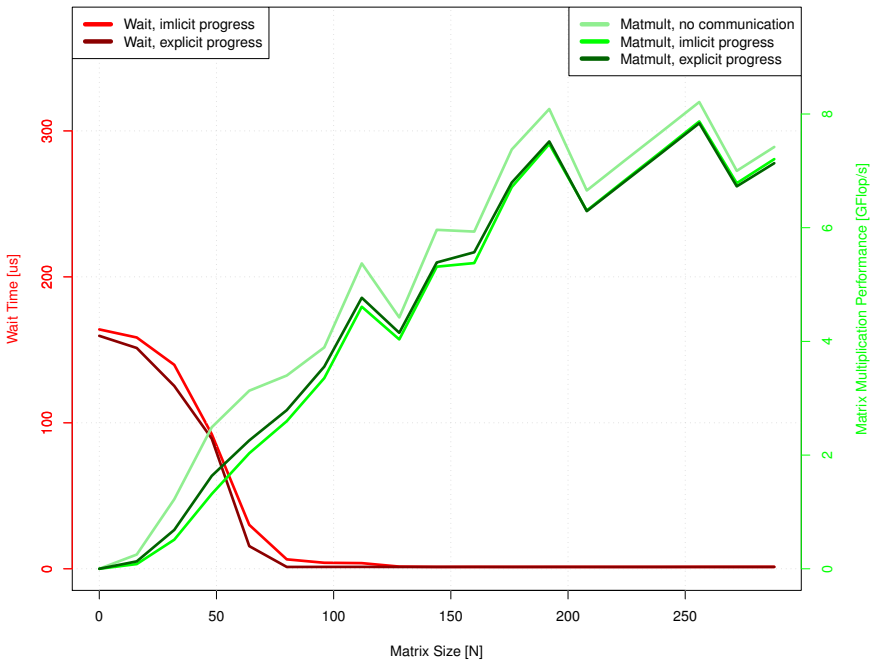


Figure 6.14: ESPGOAL Overhead Benchmark

The only publicly available non-blocking collective implementations known to the authors are LibNBC [HL06] and ESPGOAL. The results of the described benchmark with a Barrier with 32 participating nodes as the collective, a block size k of 16 and libNBC are shown in Figure 6.13. Apparently implicit progress does not work in LibNBC, as the time spent in the blocking wait never drops significantly when increasing the matrix size. LibNBC offers a configuration option `--with-thread` which should enable implicit progression, however, in our experiments the benchmark program terminated with a segmentation violation inside LibNBC when it was compiled with this option.

In Figure 6.14 we have performed the same benchmark with ESPGOAL. In ESPGOAL there is no real progression function, as all progress is asynchronous and driven by the interrupts generated by incoming packets. However, the small overhead of calling the GOAL_Test() function delays the matrix multiplication enough to make the lines for implicit and explicit progress in Figure 6.14 distinguishable.

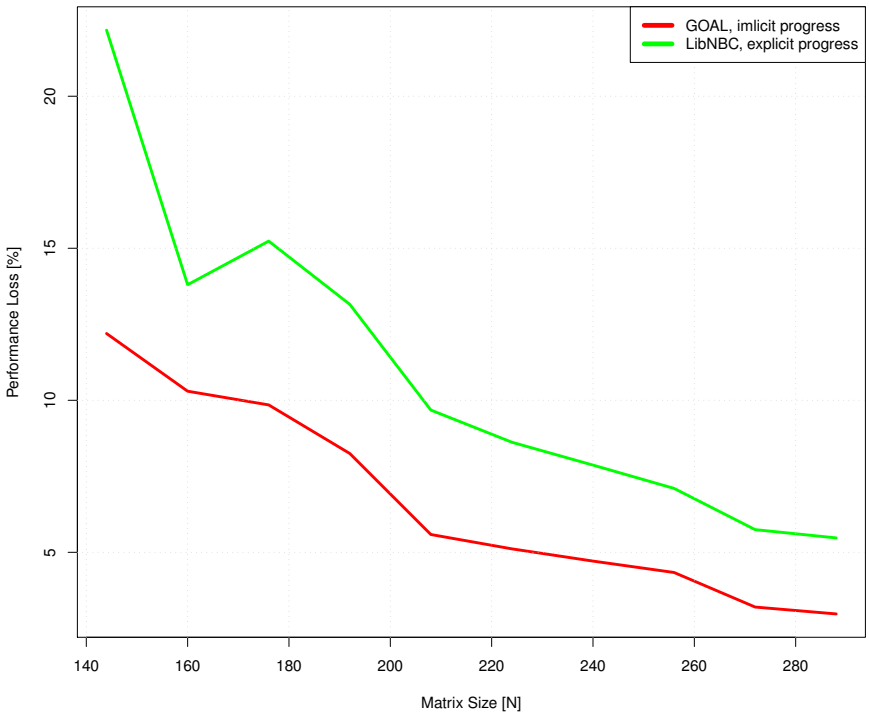


Figure 6.15: Performance loss due to overlapping the matrix multiplication with a single non-blocking barrier

In Figure 6.15 we compared the performance impact on the matrix multiplication induced by ESPGOAL with implicit progression and LibNBC with explicit progression. It makes no sense to examine LibNBC with implicit progression as it does not seem to offer this functionality, as indicated in Figure 6.13. For a similar reason we choose not to ana-

lyze ESPGOAL with explicit progression, as explicit progression is unnecessary in ESPGOAL.

Note that the overhead benchmark only issues a single non-blocking barrier, so as the matrix size increases the relative performance loss decreases. This is exactly what can be seen in Figure 6.15. We omitted the small matrix sizes from this plot where the waittime significantly different from zero, since comparing these results would not yield meaningful results because we can not guarantee that the amount of communication overlap was the same for LibNBC and ESPGOAL.

We can conclude that ESPGOAL offers true asynchronous progression and significant improvements in terms of host overhead compared with LibNBC.

6.7 Comparing Different Ways to use Ethernet NICs

The Linux kernel network stack consists of multiple layers, each try to provide a higher level of abstraction. The Linux network stack is shown in Figure 6.16.

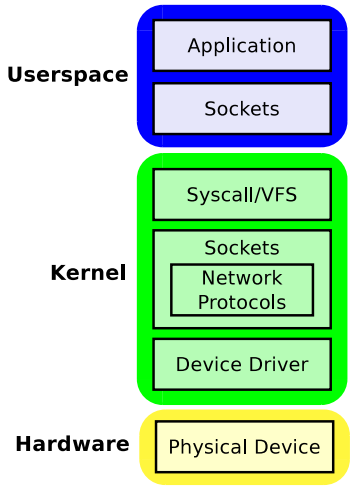


Figure 6.16: Linux Kernel Network Stack

A network interface card typically has a hardware buffer to temporarily store incoming network packets. When a new packet arrives the Linux kernel is notified about it with an interrupt from the network card. The device driver retrieves the newly received packets from the hardware buffer and stores them in so called socket buffers (*skbs*).

If we want to write kernel code that deals with incoming packets we can write a function that gets the packets it is interested in as an skb and register our function as a "receive hook". This is done with the Linux kernel function `dev_add_pack()` defined in `Linux/netdevice.h`. This function takes an argument that specified the receive hook function pointer and the protocol family (ethertype) we are interested in. To send data the developer has to set up an skb with the necessary information (receiver address, ethertype, etc.) and hand this skb over to the device driver by calling `dev_queue_xmit()`. This is the lowest layer of abstraction which is offered by the Linux kernel to send and receive data in a device independent manner. The benefit of this approach is that the functions mentioned above do not sleep and therefore they can be called in an irqhandler or Tasklet. The disadvantage is that, unlike sockets, it does not provide abstract concepts such as connections.

Another possibility how to implement network communication in the Linux kernel is to use the kernel socket API. Utilizing kernel sockets is very similar to userspace socket programming, however, in the kernel we have to ensure that whenever we want to do something with a socket that we have the appropriate lock to prevent race conditions. Most network protocols supported by the Linux kernel, such as TCP are implemented with the kernel socket API. The downside of the socket API is that certain functions, for example, sending data via `kernel_sendmsg()` is not possible in an interrupt handler or Tasklet. If such functionality is required it has to be implemented in a separate kernel thread or a workqueue element. This is the reason why ESPGOAL was implemented using Workqueues. Other network protocols such as TCP do not have to use Workqueues or an extra kernel thread as the problematic socket API function which might sleep are usually called from userspace.

A standard user application uses the userspace socket API declared in `sys/socket.h` to initialize and bind a socket. A socket is a file descriptor in Linux, so later calls to read() or write() pass through the virtual file system (*VFS*) layer.

As explained above it is currently unavoidable to use an extra kernel thread or Workqueues in ESPGOAL, since the ESP protocol is implemented using the Linux kernel socket API and some function calls in this API might sleep. This raises the question if the scheduling overhead implied by this has a negative impact on ESPGOALs performance, compared to the other possible approaches to send and receive data in the kernel. If the overhead required to start a new work item in a workqueue is significant it is desirable to have an upper bound on its performance impact so that we can decide if it would be useful to exchange the ESP protocol with something that directly utilizes the functions offered by the device driver to send and receive data in future work.

To enable us to answer these questions we implemented a simple ping pong microbenchmark. The results are shown in Figure 6.17. The benchmark consists of three different

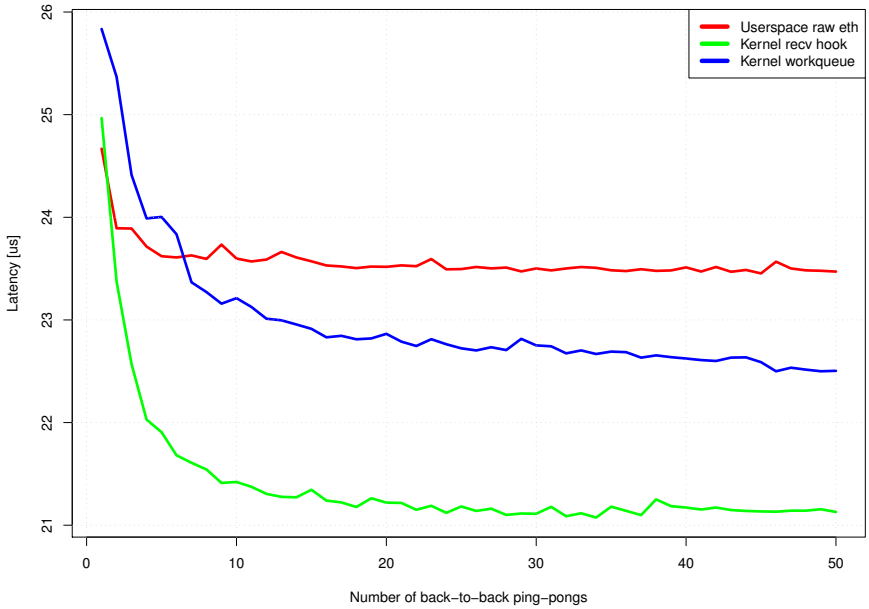


Figure 6.17: Comparison of different ways to communicate over Ethernet

implementations of a pingpong scheme, one is implemented in userspace using the raw socket API and the other two are implemented in kernelland. They all operate on the same principal: an Ethernet packet with a special ethertype is generated which contains four bytes of payload. The payload is interpreted as a signed integer. Upon reception of a Ethernet frame of the ethertype used by this benchmark the payload number is examined. If it is zero we take a timestamp t_e and start the next round of the benchmark. If it is larger than zero we send a new Ethernet packet to the host that we received the original Ethernet frame from with the payload from the original packet diminished by one. When we sent the first packet in each round we take the timestamp t_s for this round.

As a result we can measure how long it takes to perform n consecutive ping pongs with a specific communication API if we send an initial packet with the payload $2n - 1$ and subtract t_s from t_e. As we are interested in the one-way latency we divide the result by $2n$. Each measurement is repeated 10 times and the median value is shown in the graph.

The kernel based benchmarks are implemented as a kernel module with a special receive hook function registered. In the receive hook we either directly send the reply (shown as “kernel recv hook” in the graph), or we create a new workqueue item which contains a function that does this (shown as “kernel workqueue”). Both kernel benchmarks are started via the ioctl interface. The ioctl handler gets the initial payload information, the MAC address of the peer we want to use for this benchmark and the device index of the device that we want to use to send the ping pong packets as arguments for the ioctl. The timestamp t_s is saved directly after entering the ioctl.

The userspace benchmark opens a raw socket on both nodes and the server takes the start timestamp t_s and sends the initial packet for each round. After that server and client block until they receive a packet of the specified ethertype and "reply" if the payload is larger than zero, otherwise the timestamp t_e is taken.

We can observe that the overhead for inserting and scheduling the workqueue element adds about 1.6 μs of latency. Unfortunately this overhead can add up in every round of collective patterns such as a bruck barrier. However, if ESPGOALs latency is benchmarked in a similar fashion it shows a much higher latency of about 29 μs which can not be explained by the workqueue scheduling overhead.

7 Conclusions and Future Work

We implemented a dependency driven communication framework that offers true asynchronous progress without an extra progression thread. We defined an API to use such a framework that supports simple sends and receives as well as vector sends and receives as well and local operations. We have demonstrated that it is possible to implement MPI collective communication functions in that framework without loosing performance. Our framework shows significant improvements in terms of host overhead over existing userspace implementations of non-blocking collectives.

Since our ESPGOAL kernel module provides virtualization our work can serve as a basis for similar implementations and can be seen as an important step on the way to flexible hardware support for collective communication.

Our work shows that it is possible to implement dependency driven communication schemes as a Linux kernel module without placing constraints on the user. For example our GOAL scheduler does not require the user to pin the memory used for communication buffers.

In future work this implementation should be tuned further so that it can compete with state of the art low overhead Ethernet protocols such as Open MX. Especially the cause for the high startup overhead of about 30 μs should be identified and eliminated if possible. One possible way to tune ESPGOAL even further would be to replace the ESP protocol with another low overhead Ethernet protocol that shows better performance in point to point latency benchmarks, for example it could be investigated if ESP can be replaced with the kernel part of Open MX.

Also we did not investigate how large schedules can efficiently be handled by the schedule interpreter. Such considerations are to be seen as another step on the way to a hardware implementation for flexible collective offload to the network card.

8 Acknowledgments

The authors want to thank Nico Mittenzwey and Wolfgang Rehm for access to the CHiC cluster and for supervising this research project.

Timo Schneider wants to thank Torsten Höfler for the countless valuable discussions and for curating his interest in computer science.

Bibliography

[AAA+02] N.R. Adiga, G. Almasi, GS Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich *et al.*: *An overview of the BlueGene/L supercomputer*, in *ACM/IEEE 2002 Conference Supercomputing*, S. 60–60, 2002, ISSN 1063-9535.

[ABB+09] S. Alam, R. Barrett, M. Bast, MR Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, JS Vetter *et al.*: *Early evaluation of IBM BlueGene/P*, in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, S. 1–12, IEEE, 2009.

[AHA+05] G. Almási, P. Heidelberger, C.J. Archer, X. Martorell, C.C. Erway, J.E. Moreira, B. Steinmacher-Burow und Y. Zheng: *Optimization of MPI collective communication on BlueGene/L systems*, in *Proceedings of the 19th annual international conference on Supercomputing*, S. 253–262, ACM, 2005, ISBN 1595931678.

[BHK+02] J. Bruck, C.T. Ho, S. Kipnis, E. Upfal und D. Weathersby: *Efficient algorithms for all-to-all communications in multiport message-passing systems*, *Parallel and Distributed Systems, IEEE Transactions on*, Bd. 8(11):S. 1143–1156, 2002.

[Cia03] G. Ciaccio: *Messaging on Gigabit Ethernet: Some experiments with GAMMA and other systems*, *Cluster Computing*, Bd. 6(2):S. 143–151, 2003, ISSN 1386-7857.

[Dyk99] P. Dykstra: *Gigabit ethernet jumbo frames*, *White Paper, WareOnEarth Communications*, 1999.

[Geo04] P. Geoffray: *Myrinet express (MX): Is your interconnect smart?*, in *High Performance Computing and Grid in Asia Pacific Region, 2004. Proceedings. Seventh International Conference on*, S. 452, IEEE, 2004, ISBN 076952138X.

[Gog08] B. Goglin: *Design and implementation of Open-MX: High-performance message passing over generic Ethernet hardware*, in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, S. 1–7, IEEE, 2008, ISSN 1530-2075.

[Gor04] S. Gorlatch: *Send-receive considered harmful: Myths and realities of message passing*, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Bd. 26(1):S. 47–56, 2004, ISSN 0164-0925.

[Hem94] R. Hempel: *The MPI standard for message passing*, in *High-Performance Computing and Networking*, S. 247–252, Springer, 1994.

[HGLR07] T. Hoefler, P. Gottschling, A. Lumsdaine und W. Rehm: *Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations*, *Elsevier Journal of Parallel Computing (PARCO)*, Bd. 33(9):S. 624–633, Sep. 2007, ISSN 0167-8191.

[HL06] T. Hoefler und A. Lumsdaine: *Design, Implementation, and Usage of LibNBC*, Techn. Ber., Open Systems Lab, Indiana University, Aug. 2006.

[HL08] T. Hoefler und A. Lumsdaine: *Message Progression in Parallel Computing - To Thread or not to Thread?*, in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, IEEE Computer Society, Oct. 2008, ISBN 978-1-4244-2640, ISSN 1552-5244.

[HLR07] T. Hoefler, A. Lumsdaine und W. Rehm: *Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI*, in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*, IEEE Computer Society/ACM, Nov. 2007.

[HRM+06] T. Hoefler, M. Reinhardt, F. Mietke, T. Mehlan und W. Rehm: *Low Overhead Ethernet Communication for Open MPI on Linux Clusters*, Bd. CSR-06(06), Jul. 2006, ISSN 0947-5125.

[HSL08] T. Hoefler, T. Schneider und A. Lumsdaine: *Accurately Measuring Collective Operations at Massive Scale*, in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, PMEO'08 Workshop*, Apr. 2008, ISBN 978-1-4244-1694-3, ISSN 1530-2075.

[HSL09a] T. Hoefler, T. Schneider und A. Lumsdaine: *LogGP in Theory and Practice - An In-depth Analysis of Modern Interconnection Networks and Benchmark-*

ing Methods for Collective Operations., *Elsevier Journal of Simulation Modelling Practice and Theory (SIMPAT)*, Bd. 17(9):S. 1511–1521, Oct. 2009, ISSN 1569-190X.

[HSL09b] T. Hoefler, C. Siebert und A. Lumsdaine: *Group Operation Assembly Language - A Flexible Way to Express Collective Communication*, in *ICPP-2009 - The 38th International Conference on Parallel Processing*, IEEE, Sep. 2009, ISBN 978-0-7695-3802-0.

[HSL10] T. Hoefler, T. Schneider und A. Lumsdaine: *Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale*, *International Journal of Parallel, Emergent and Distributed Systems*, Bd. 25(4):S. 241–258, Jul. 2010, ISSN 1744-5779.

[HZ07] T. Hoefler und G. Zerah: *Transforming the high-performance 3d-FFT in ABINIT to enable the use of non-blocking collective operations*, Techn. Ber., Commissariat a l'Energie Atomique - Direction des applications militaires (CEA-DAM), Feb. 2007.

[Int07] Intel: *Interrupt Moderation Using Intel GbE Controllers*, `http://download.intel.com/design/network/applnots/ap450.pdf`, Apr. 2007.

[MMHR07] F. Mietke, T. Mehlan, T. Hoefler und W. Rehm: *Design and Evaluation of a 2048 Core Cluster System*, *Kommunikation in Clusterrechnern und Clusterverbund-systemen*, S. 40, 2007.

[NI10] A. Nomura und Y. Ishikawa: *Design of Kernel-Level Asynchronous Collective Communication*, *Recent Advances in the Message Passing Interface*, S. 92–101, 2010.

[PHP03] N. Park, B. Hong und V.K. Prasanna: *Tiling, block data layout, and memory hierarchy performance*, *IEEE Transactions on Parallel and Distributed Systems*, S. 640–654, 2003, ISSN 1045-9219.

[PJD04] R. Prasad, M. Jain und C. Dovrolis: *Effects of interrupt coalescence on network measurements*, *Passive and active network measurement*, S. 247–256, 2004.

[PSZ92] R. Palumbo, H. Saiedian und M. Zand: *The operational semantics of an active message system*, in *Proceedings of the 1992 ACM annual conference*

on Communications, S. 367–375, ACM, 1992.

[Sch77] H.P. Schwefel: *Numerische optimierung von computer-modellen mittels der evolutionsstrategie*, *Interdisciplinary systems research*, Bd. 26, 1977.

[SMG96] Q.O. Snell, A. Mikler und J.L. Gustafson: *Netpipe: A network protocol independent performance evaluator*, in *IASTED International Conference on Intelligent Information Management and Systems*, Bd. 6, Citeseer, 1996.

[Spe08] P.L. Specifications: *IEEE Std 802.3-2008: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 2008.

[Str69] V. Strassen: *Gaussian elimination is not optimal*, *Numerische Mathematik*, Bd. 13(4):S. 354–356, 1969, ISSN 0029-599X.

[SWP01] P. Shivam, P. Wyckoff und D. Panda: *EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing*, in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, S. 57, ACM, 2001, ISBN 158113293X.

[Tre07] Matthias Treydte: *Application Specific Optimization of Ethernet Cluster Communication*, Diplomarbeit, TU Chemnitz, 2007.

[TRG05] R. Thakur, R. Rabenseifner und W. Gropp: *Optimization of collective communication operations in MPICH*, *International Journal of High Performance Computing Applications*, Bd. 19(1):S. 49, 2005, ISSN 1094-3420.

[WR73] GN Wilkinson und CE Rogers: *Symbolic description of factorial models for analysis of variance*, *Applied Statistics*, Bd. 22(3):S. 392–399, 1973.