enough to not decrease the performance and evaluate if the additional overhead for the computation of the waiting times is worth the savings.
As work like [Ehm12] shows, the saving potential is great, but also the potential to waste energy is great.

4.5. Busy waiting in the MPI implementation

In this section I will discuss the busy waiting implemented in the MPI implementation. I will also discuss a way to not wait as busy as it is implemented now.

As noticed in Section 4.4 the MPI implementation implements a busy waiting in order to ensure maximum performance. This busy waiting leads to a significant waste of energy. More than four percentage of the energy is spend to perform this busy waiting. Therefore in this section I will take a closer look on this busy waiting.

I will analyze it in the MPICH implementation, as this implementation is more efficient (Section 3.2).

```
1  if (!MPID_Request_is_complete(request_ptr))
2      {
3      MPID_Progress_state progress_state;
4
5      MPID_Progress_start(&progress_state);
6          while (!MPID_Request_is_complete(request_ptr))
7      {
8          mpi_errno = MPID_Progress_wait(&progress_state);
9          if (mpi_errno) { MPIR_ERR_POP(mpi_errno); }
10     }
11     MPID_Progress_end(&progress_state);
12     }
```

Listing 4.1: MPI implementation to wait for the request to complete (from MPICH implementation).

Listing 4.1 shows the implementation of the MPI-wait function. This function is called whenever a blocking MPI-function is called. One can see hat the `MPID_Progress_wait()` function is called repeatedly until the request is completed. The `MPID_Progress_wait()` function "Wait for some communication since '`MPID_Progress_start`'"[3]. This waiting is implemented with a busy waiting loop polling again and again if there is communication (see Listing 4.4) .

```
1  int main(int argc, char** argv) {
2      MPI_Init(&argc, &argv);
3      int rank=0;
```

[3]MPICH Source code file mpiimpl.h line 3427 version 3.2

```
4       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5
6       MPI_Barrier(MPI_COMM_WORLD);
7       if (rank==0)
8           printf("ALL: busy 10\n");
9       busy(10);
10      MPI_Barrier(MPI_COMM_WORLD);
11
12      MPI_Barrier(MPI_COMM_WORLD);
13      if (rank==0)    {
14          printf("MASTER: busy 10\n");
15          busy(10);}
16      MPI_Barrier(MPI_COMM_WORLD);
17
18      // Continiue:
19      // All Busy
20      // Master only
21      // All Busy
22
23      MPI_Finalize();
24      return 0;
25  }
```

Listing 4.2: Test program to show the busy waiting.

In order to analyze this behaviour, I have written a small test application shown in Listing 4.2. After the initial `MPI_Barrier` all processes call the busy function which will simulate that the process is doing some work (Listing 4.3). After the processes synchronized again only the master process will be busy in the next step. All other processes will be waiting in the next `MPI_Barrier`. This will repeat once, than in the last section all processes are busy again. As the program has 5 sections with 10 seconds each we would expect that one run needs minimal 50 seconds to finish.

```
1   // keep the calling process busy for a given amount of time
2   void busy(double s)
3   {
4   struct timeval start_time;
5   struct timeval cur_time;
6   gettimeofday(&start_time, NULL);
7   int i=0;
8   double time= 0.0f;
9
10      while (time < s)
11      {
12          i++;// do something
13          gettimeofday(&cur_time, NULL);
14          time = (cur_time.tv_sec - start_time.tv_sec)
15          + (cur_time.tv_usec - start_time.tv_usec) * 1e-6;
16      }
17  }
```

Listing 4.3: The busy function used in Listing 4.2.

I have executed this test program with the normal MPICH implementation as well as with the small adjustment shown in Listing 4.4. This adjustment alters the behaviour of the MPI implementation, so that the processor sleep between the polling.

```
1 int stime=0;
2     while (MPID_nem_queue_empty (MPID_nem_mem_region.my_recvQ) ||
          ↪ !MPID_nem_recv_seqno_matches (MPID_nem_mem_region.my_recvQ))
3     {
4 usleep(stime);
5 stime++;
6 // do the actual polling...
```

Listing 4.4: Changes to the `MPID_nem_mpich_blocking_recv function`.

I first will present why I have choosen this adjustment, I will present the influence it has on the needed energy as well as discuss the additional Runtime needed.

[BFT16] discuss several ways of reducing the energy consumption while waiting on a lock guarding some critical section. Other techniques than sleep between polling rely on the process currently holding the lock, which should than wake up the waiting process. In context of locks to secure a critical code section, this is not an optimal solution as it requires additional time when releasing the lock to figure out, who is waiting and which of the waiting processes is eligible to hold the lock next fo fairness reasons. This additional overhead might overcome the benifit of setting the processor to sleep.
When considering this for MPI-Communication this is especially true. In particular for immediate send operations, where the sending process marks the sendbuffer, so that the sending process can start at any time, and then immedeatly continiue without taking care if the rechieving process is ready to rechive the data. Then it is not feasible to spend any effort to try to wake up the rechieving process, as it may not be ready to rechieve the data yet. So techniques that rely on the sender to wake up the process, which is waiting for the tansmission, is incopatible with the concept of the immediate send/recieve operations.

Therfore I have choosen to sleep between polling.
The time that the processor sleep between two polls increases with every time the polling loop is executed. The intuition behind the adjustment is that at the begining one expect that the application is efficient and do not waste much runtime waiting on other processes. Therefore it is best to sleep only for short intervalls to guarantee that the processing can continiue as soon as the communication finished. But when the communication does not finish early the importance of saving energy grows while the importance of an immediate continuation of the processing decline, as the time, which before was wasted waiting anyway, is much larger than the additional time spend in the non busy waiting. In my particular test the 0.001 second the non busy waiting takes longer is very small in relation to the 10 seconds the process was waiting anyhow.

Figure 4.9 shows the energy usage of the test program (Listing 4.2) with the standard implementation (busy waiting) in green and the energy usage with sleep between polling in blue.

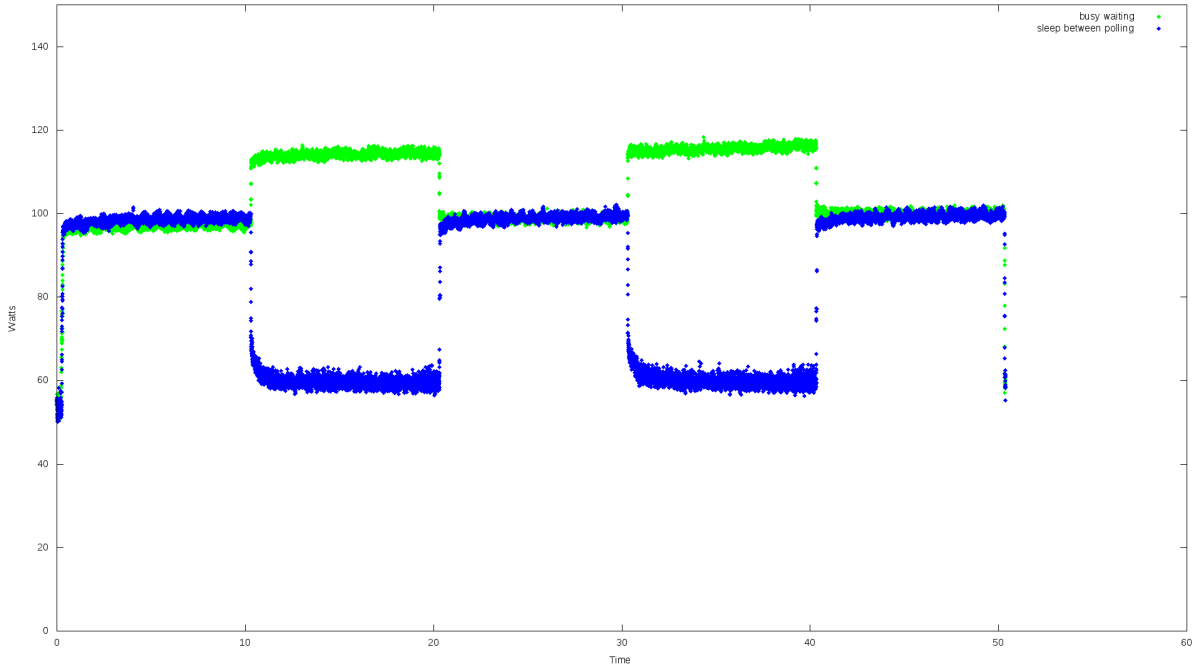

Figure 4.9.: Energy usage of the test program (Listing 4.2) with busy waiting (green) and sleep between polling (blue).

One can see that the energy needed in the sections where only one process is active is halved when the other processors sleep between polling. This is a significant saving.

But this huge saving is not without an impact on the runtime. In this particular test the runtime increases approximate 0.002 seconds. In this case the increase is so small that it cannot be noticed in Figure 4.9.

In the worst case, the communication is finished right after the processor go to sleep between two pollings. Let Y denote the time (in microseconds), the processor waited until that point. It can be written as $Y = \sum_{i=1}^{x} i$ + the additional time needed for the actual polling which I will leave out here. So before the next polling the processor will first sleep for $x + 1$ microseconds. To get the maximum propotion of the additional waited time I define $f(Y) = \frac{x+1}{Y}$ which describes the propotion of the additional time the processor have to wait in the worst case depending on the time that was spent waiting before. The sum $Y = \sum_{i=1}^{x} i$, where x describe the number of pollings done by the processor so far, is the triangular sum and can therefore also be written as $Y = \sum_{i=1}^{x} i = \frac{x(x+1)}{2} \Rightarrow x + 1 = \sqrt{2Y + 1}$. Therefore is $f(Y) = \frac{x+1}{Y} = \frac{\sqrt{2Y+1}}{Y}$.
For this particular test system, when the propotion of additional time is below 0.5, no additional energy is wasted, as the non busy waiting consumes only half as much energy as the busy waiting does. In the plot of the function in Figure 4.10, one can see that the

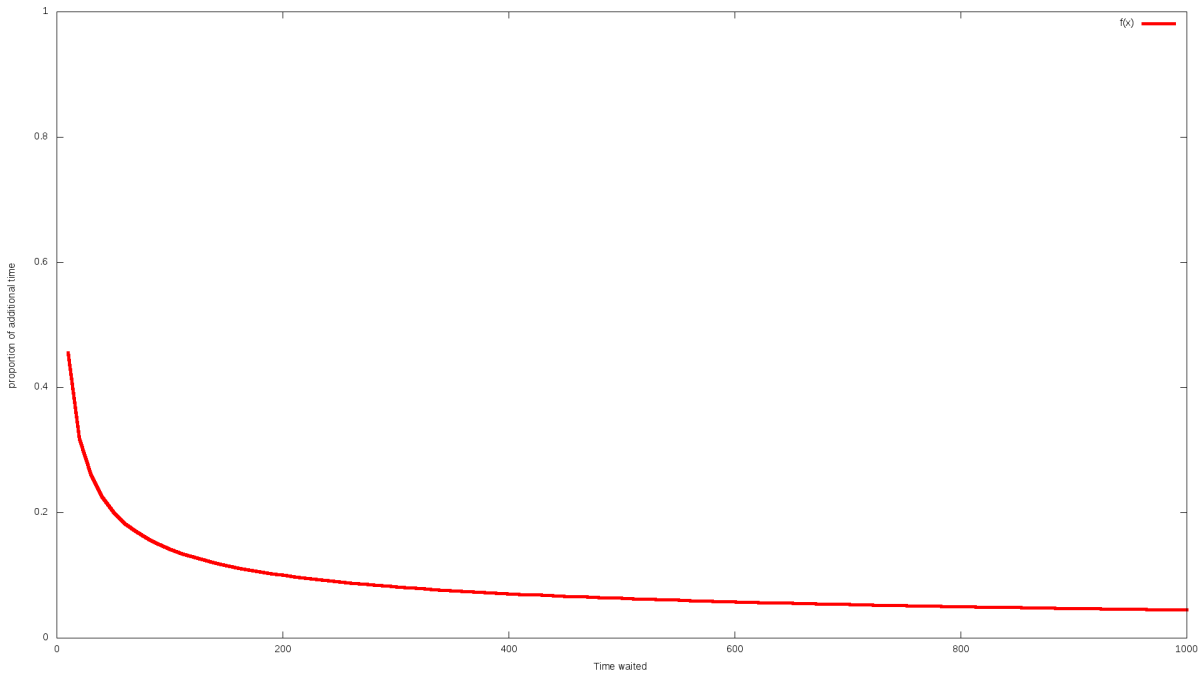

Figure 4.10.: Plot of $f(x) = \frac{\sqrt{2x+1}}{x}$ which describes the propotion of the additional time the processor have to wait in the worst case depending on the time that was waited before.

propotion of additional waited time declines with more time spended in waiting. But one can also see, that the proportion of 0.5 is not reached for the first few microseconds of waiting. The function stays below 0.5 for all values of x with $2(2+\sqrt{5}) < 9 \leq x$.

To get a view on the absolute values of the additional waited time, Figure 4.11 shows the waited time. The time for busy wait in bluish green and the time when the implementation waits between polling in purple. One can see that the time waited is a sort of step function. Every step is linked to the time which was waited between two polls (the function which describes the waiting time can be denoted as $f(x) = 0.5 * \left\lceil \sqrt{2x+2} \right\rceil * (\left\lceil \sqrt{2x+2} \right\rceil + 1)$ with x as the time when the message arrives). When looking at it on a larger scale like in Figure 4.12 it shows that the loss of runtime when waiting between polling, is negligible in comparison to the time that was spend waiting anyhow, as the two lines are so close they can not be distinguished. It also shows, that the sleeping time between two polls does not increase very fast. In fact, its increase will even slow down as the time until the next poll and therefore the next increase of the waiting time becomes longer and longer. In this Setup this effect prevents the waiting time to become too large so that the latentency grow too much. Nevertheless it is a good idea to set a maximum waiting time, as the result have shown that it will still be enough, to allow the processor to power down. Looking at Figure 4.9 one can see, that the energy used by the system dos not decline with an increase in the waited time after the first seconds of waiting.

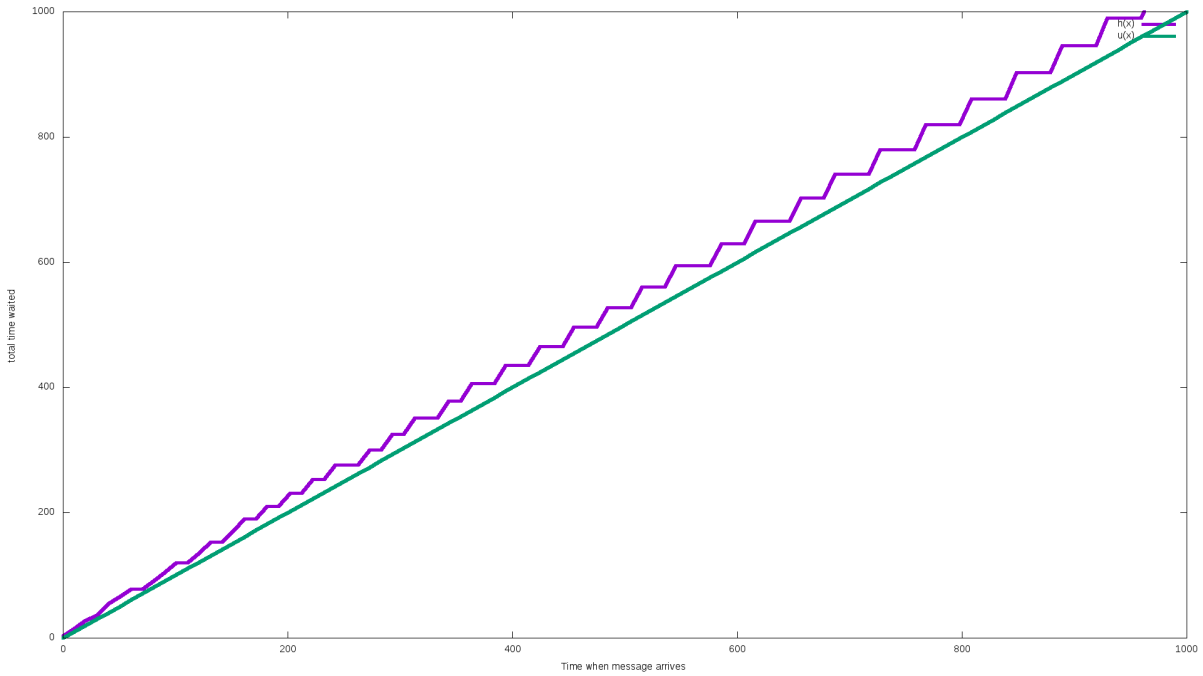

Figure 4.11.: Plot of the waited time. Busy wait in bluish green, wait between polling in purple

Therefore I suggest that in future MPI implementations the programmer can choose whether he expects a MPI call to be asynchronous and therefore might have to wait longer, or he expects it to be called by all involved processes at rughly the same time. When the call is expected to be asynchronous, the MPI implementation should not wait as busy as it does now. Otherwise the busy waiting ensures the maximal performance and is still the better option for the calls where all involved processes are expect to be synchronous. This counts in particular any use for `MPI_Barrier`, as the concept of `MPI_Barrier` is to synchronize asynchronous processes.

Considering that in an efficient application the waiting time is already minimized, one should ask the question if it is really worth the effort. For the ECOHAM application, more than 14 kJ are wasted performing busy waiting for the I/O to finish. This accounts for 4 percentage of the energy needed by the ECOHAM model.
As Minartz determine in his PhD Thesis, also savings in the lower pertentage range are in general worth the effort [Min13].

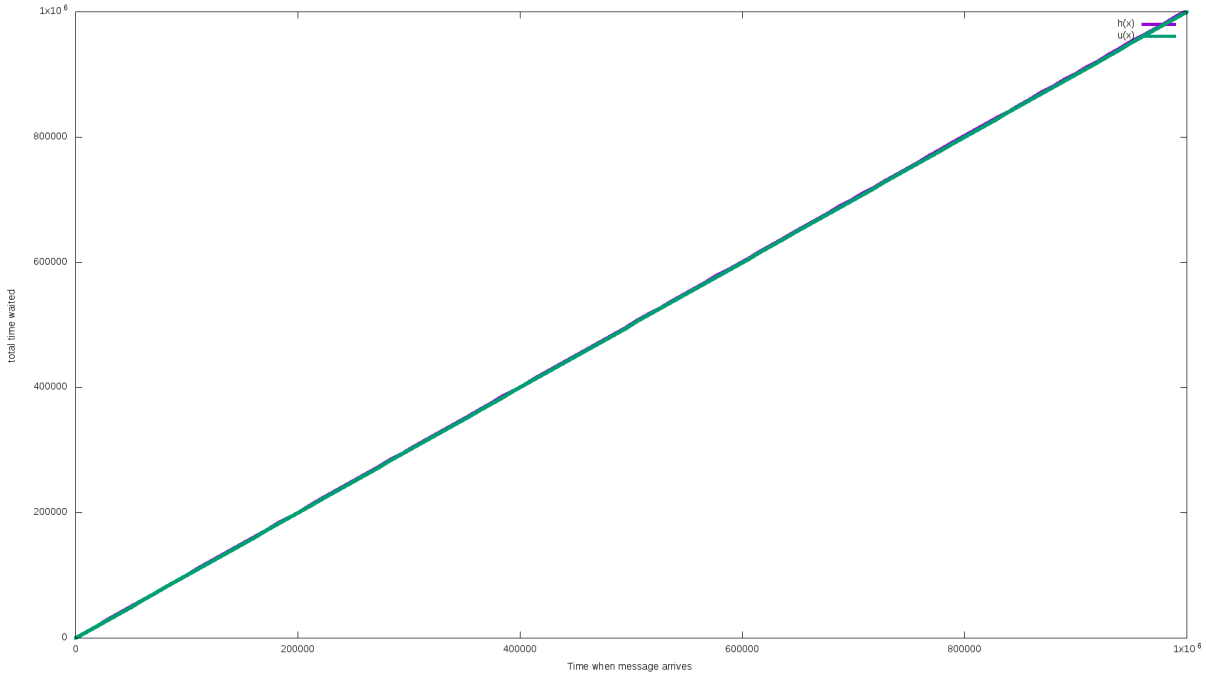

Figure 4.12.: Plot from Figure 4.11 on a larger scale.