Optimizing Recursive Functions for MPI

Andreas Wagner
*

Abstract

The message passing interface (MPI) can be regarded as one of the standard programming techniques for distributed memory architectures. But solving a problem efficiently with MPI forces the programmer to apply several transformations to the program to deal with task decomposition, communication and synchronization. On the other hand, many common problems in science can easily be formulated by means of recursion. Unfortunately, a recursive programming style is not efficent in MPI environments, as it introduces a huge communication and synchronization overhead. In this paper we present an algorithm, which automatically optimizes recursively defined functions for the use in MPI by exploiting data locality and reusing intermediate computations. We show that our algorithm leads to noteworthy speedups and a significant reduction in the overall data transfer. The results imply, that recursion can be used in MPI environments, whithout making too many concessions in terms of performance.

1. Introduction

Since its initial release in 1994 [5], the message passing interface (MPI) can be regarded as the de facto standard in high performance computing and cluster applications. MPI favors the single program multiple data (SPMD) or multiple program multiple data (MPMD) approach, where programs are started with a fixed number of processes. Due to this explicit programming model, programmers have to deal with data decomposition, task mapping and synchronization, which requires sophisticated tailoring of the problem and can make problem modeling quite difficult [1].

In contrast, many commonly used algorithms can be formulated naturally by means of recursion [8]. For example, Asanovic et al. present a number of applications which are easiest implemented in a recursive fashion. Also, some functional programming languages force programmers to use recursion due to a lack of loop-expressions.

Using recursion in a MPI program is therefore desireable, as it would simplify the implementation of a wide range of algorithms or the porting of functional programs to distributed memory architectures. However, two major drawbacks conflict with an efficient use of recursion in MPI: the (usually) fixed number of available processes during runtime and a large communication overhead. While the first problem has been more or less solved since the release of the MPI-2 standard (which allows dynamic process spawning [4]), there exists no standard approach to reduce the communication overhead.

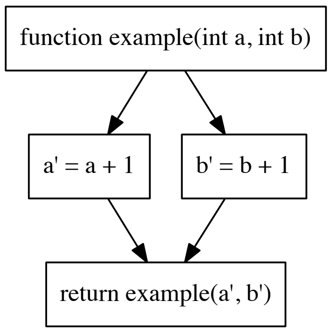
We motivate this issue with a simple example. Consider a recursive function which is already decomposited into tasks (figure 1(a)). Typically, each of the tasks will be pinned to a fixed processing node. Figure 1(b) illustrates the communication during execution. For the subtasks to be able to complete their work, all neccessary data (the variables a or b respectively) has to be transferred from the root-task to the corresponding subtasks. They will process the data and send the result to a joining node, which initiates another recursive step if neccessary. Therefore it has to send the input data (which was received immediately beforehand) to subtasks. Again they will process the data and send the result to a joining node. This process is repeated until the termination condition holds. Finally, each of the joining nodes returns its results to its predecessor (not illustrated in figure 1(b)).

What can be observed is, that the root or joining task transfers data to its subtasks in every recursive step. Furthermore, the subtasks forward their results to a common descendant, which spawns another function invocation. If the subtasks are located on the same processing node during program execution, which is typically the case in MPI environments, the joining node which invokes another recursive step sends data to the subtasks which has obviously been produced by themselves immediately beforehand. So in principle, this data transfer is redundant, as the data is already located on the processing node. Especially if the messages are considerably large (e.g. if they contain arrays or matrices), such communication is costly and can negatively influence performance on distributed-memory architectures.

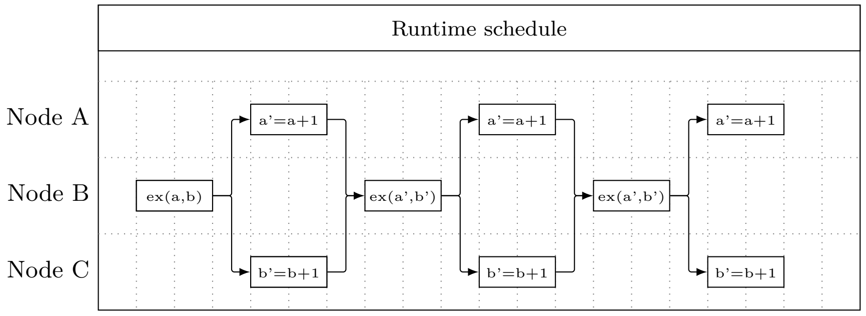
For iterative applications on shared-memory systems, similar considerations emerged years ago. Researchers have developed many methods to exploit spatial and temporal data locality to maximize cache hit rates and improve overall program performance[1].

In this paper we present a method to detect and exploit data locality in distributed-memory systems, in order to optimize recursive functions for MPI. We show that the total amount of data which has to be transferred during program execution can be reduced significantly by avoiding unneccessary data transfers resulting from recursive function

[1] For example, Wolf and Lam came up with a loop-transformation algorithm that uses interchange, reversal, skewing and tiling [13]



(a) A minimal task graph of a recursive function. For simplicity reasons, the termination condition has been omitted.



(b) The runtime schedule of the recursive function as it would probably look like during execution in a MPI environment. The tasks are strictly located at the processing nodes.

Figure 1.

invocations. In consequence, we are able to reduce the total execution time of recursive functions in MPI.

2. Methodology

The approach presented in the next sections follows two key ideas:

- Identification of variables which can be kept task-local without the need to send them in every recursive step
- Reuse of tasks which work with these variables and maintainance of their local context (so sending parameters to them in every step can be omitted)

Both is pursued under the constraint of being communication optimal. Our algorithm therefore consists of two major steps. First we get a better understanding of the recursive function by "expanding" it. The recursive call in the task graph is replaced by the task graph itself, modified and extended by additional expressions, to obtain a valid task graph.

In a second step this task graph is scheduled with an arbitrary, communication-aware scheduling algorithm. The resulting schedule is used to identify tasks which reside on the same processing node during function execution. The output of those tasks might be cached locally, so the overall communication overhead can be reduced.

In the following, we will explain the algorithm in detail. Step 1 to 4 focus on the creation of a expanded task graph. Step 5 finally describes how to obtain additional information from the task graph to find function parameters and variables which can be cached during the function application. This information can be used in MPI programs to speedup the execution time of recursive function applications.

The algorithm operates on a preconditioned task graph as described in [7]. Because the target platform is a message passing environment and each variable dependency may impose a data transfer in the final program, it is important that all transitive edges are present in the task graph. Figure 2 shows such an initial task graph for a simple recursive function. Based on this graph, the further steps of the algorithm will be illustrated.

A node in the task graph will be called task. A task consists of one or more statements. Edges in the task graph are directed and might be annotated with zero or several variable names, indicating that the variables act as an input

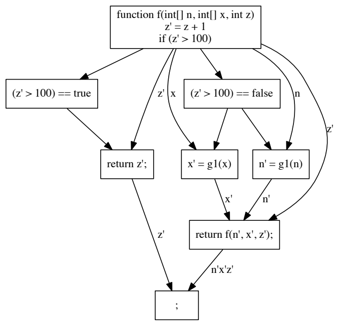


Figure 2. Task graph of a simple recursive function.

dependency to the corresponding task or are produced by a task. When talking about the *declaration node/task* or the *invocation node/task*, we refer to the nodes that contain the method declaration and the node where the recursive invocation occurs.

2.1 Step 1: Duplicating the task graph

In the first step of the algorithm, the input task graph will simply be duplicated to obtain an exact copy (Figure 3(a)). From now on we refer to the original task graph as *TG1* and the duplicated task graph as *TG2*.

Depending on the size of the invocation node in *TG1*, the algorithm requires an intermediate step at this point. If the invocation node contains more statements than just the invocation, it has to be split up to move the invocation expression to its own task. This is neccessary, as we intend to completely integrate *TG2* into *TG1*. Two cases have to be distinguished:

1. The expression which contains the invocation is the first or the last one in the task. Then it must be moved in a new task and new edges have to be added to *TG1* maintaining the dependencies.
2. The expression which contains the invocation is surrounded by statements. Then two new tasks have to be inserted into *TG1*, one before and one after the invocation. New edges have to be inserted into *TG1* to maintain dependencies, just like in the first case.

After duplicating and modifying *TG1*, we can begin to integrate *TG2* into *TG1*.

2.2 Step 2: Parameter expression analyisis and creation of bridge tasks

As a preparation step for merging *TG1* and *TG2*, it is neccessary to determine the expressions the function is recursively invoked with. These expressions have to be inserted as new tasks in the resulting task graph. The following procedure operates on *TG1*.

For each parameter expression it is checked which preceding tasks pose a dependency (which tasks produce a variable that is occuring in the current expression). We call these set of tasks T_{pred}. To find this set, the incoming edges of the invocation node are determined. It is checked if the variable provided by each single edge is equal to a variable present in the current expression (because the existence of transitive variable dependencies is a requirement for the input task graph, it is guaranteed that no dependency gets lost at this point). If so, a new node is created which consists of a single assignment expression, where the left hand side is the parameter in the method declaratation whose position matches the one of the current parameter expression in the invocation. The right hand side is the parameter expression itself. The newly created node is inserted into *TG1* by making it a direct successor of each task in T_{pred}. The variable attributes of each edge are simply copied from the corresponding edges which connect the tasks of T_{pred} and the invocation task. The set of all new tasks created this way is called T_{bridge}.

Next, each task in T_{bridge} has to be connected with an appropriate successor. For each direct successor T_{succ} of the method declaration node in *TG2*, it is checked which parameter of the method acts as an input dependency. This set of parameters is called V_{in}. Each of the parameters matches a corresponding task in T_{bridge} (with the parameter occuring on the left hand side of the expression). Each task in T_{bridge}, whose left hand side is equal to a variable in V_{in}, is connected to T_{succ}. The variable property of the inserted edge is set to the left hand side of the edge's source.

The resulting task graph after inserting the bridge tasks is shown in figure 3(b).

2.3 Step 3: Integration of return statements and joining nodes

Resulting from steps one and two, all the return statements of the input task graph appear twice in the current intermediate representation. To obtain a valid task graph, these duplicates must be eliminated. All tasks of *TG2* which provide a return value must be removed. Corresponding input edges have to be connected to returning tasks of *TG1*. For easier reference, a returning task in *TG1* is called Ret_1 and a returning task in *TG2* is called Ret_2.

To be able to remove Ret_2 from the current task graph, all predecessors of Ret_2 must be connected to the corresponding Ret_1 instead. After reconnecting each predecessor of Ret_2, it can safely be removed.

Apart from tasks representing the function's return value, both *TG1* and *TG2* contain a dummy task, indicating the end of the method-block. Like return-statements, these joining nodes must also be fused to obtain a valid task graph. This is achieved analogously to removing return nodes by connecting all predicessors of the joining node in *TG2* to the corresponding joining node in *TG1*.

Figure 3(c) shows the resulting task graphs after the return statement and joining node integration.

2.4 Step 4: Removing unneccessary nodes and task graph cleaning

After connecting *TG1* and *TG2*, some parts of the initial task graphs are dispensable and can be removed in this step. This includes the method declaration in *TG2*, the method invocation in *TG1* as well as dangling nodes which were unneccessarily added in step 2, but do not produce any dependencies for successing tasks (e.g. parameters which are not used in the method body).

Dangling nodes (except joining nodes described in step 3) as well as their incoming edges can safely be removed from the task graph. For method declaration and invocation nodes, two cases have to be distinguished: If the task which contains the declaration or invocation will be empty after removing the corresponding expression, then the whole task must be removed from the task graph. Otherwise, only the expression itself may be removed. Removing expressions might invalidate input and/or output dependencies of the task. Therefore it is neccessary to update all of the task's incoming and outgoing edges. If an edge's variable attribute refers to a variable which became obsolete due to the expression removal (this is, when no other expression in the task consumes or produces this variable), the corresponding edge must be deleted from the task graph.

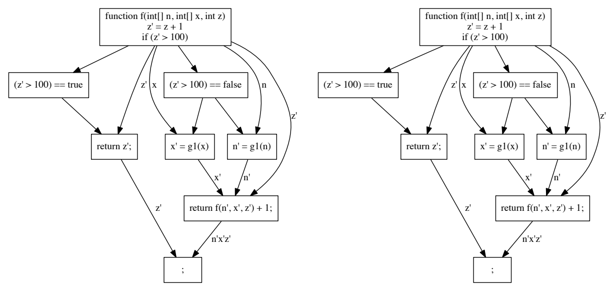
The result of step 4 is shown in figure 3(d). We now have a valid task graph, which represents the program flow of the recursion during execution. This extended task graph can be used to gain additional information about data transfer, parallel execution and possible caching opportunities. Therefore, two additional steps are needed which are described in the following: Creating a schedule and detection of cachable variables.

2.5 Step 5: Creating a static schedule and determine cachability of variables

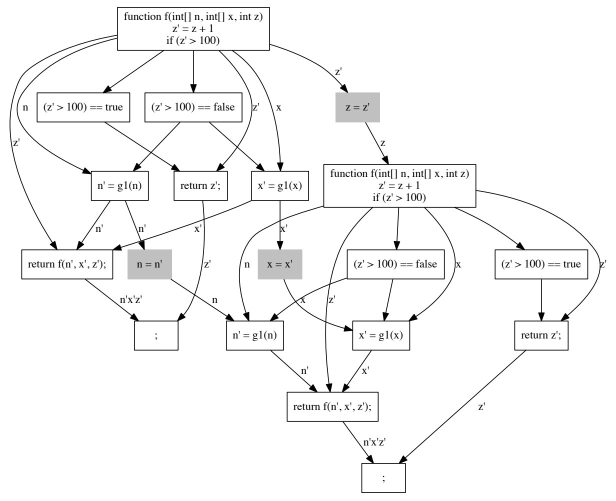
To be able to decide for a variable if it can be cached locally or has to be transferred to other processing nodes, detailed knowledge of the communication structure is mandatory. The expanded task graph created in steps 1 to 4 contains all explicit and implicit communication that occures during function execution.

Basically, a variable can be reused during the recursion, when all calculations concerning this variable reside on the same processing node. Then, only the final value (that is, when the recursion terminates) must be transferred back to the parent processor which initiated the recursion. From this point of view, it is desirable to place all tasks on the same processor, which affect this variable. On the other hand, this might lead to longer schedules because the load is not equally distributed among the available processing nodes.

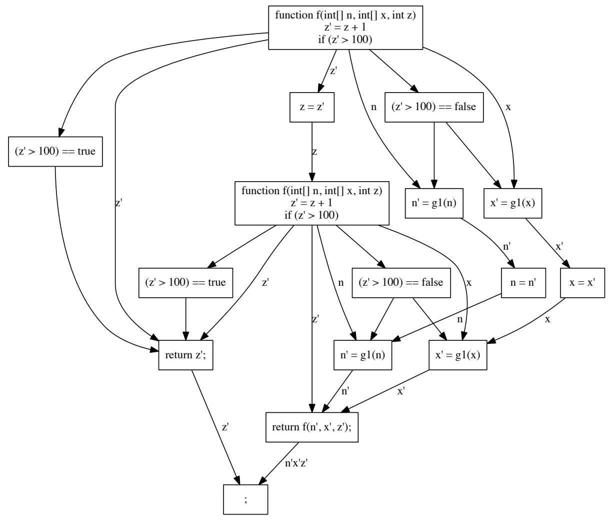
To overcome this issue, we use a standard communication-aware scheduling heuristic: List scheduling with start time minimization for clustering, as presented by Sinnen (p. 135f). A key characteristic of this heuristic is the so called "edge zeroing": When scheduling two interdependent tasks on the same processing node, the communication costs of the dependency can be set to zero, as the data does not have to be transferred anymore between two nodes.



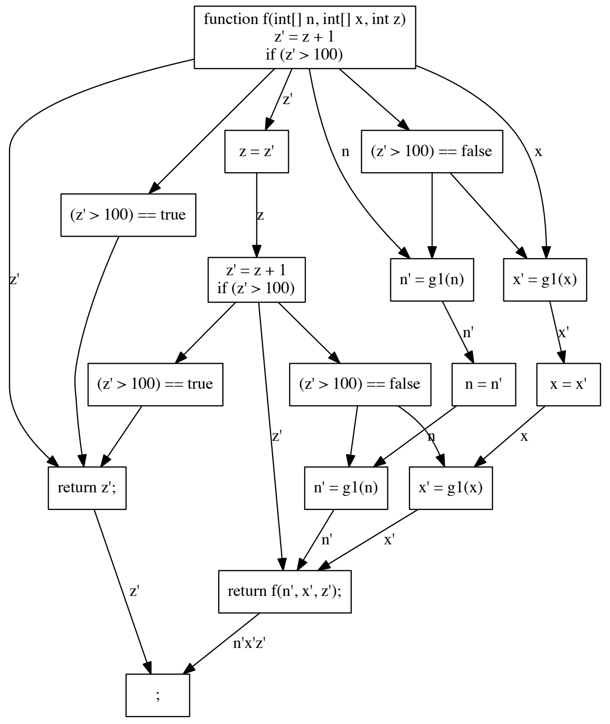
(a) The result of step 1 is an exact copy of the initial task graph. The left task graph is called *TG1*, the right task graph is called *TG2*



(b) After step 2, *TG1* and *TG2* are connected by bridge tasks. These tasks represent the expressions provided by the programmer to the method invocation in *TG1*.



(c) After step 3 *TG2* is almost completely integrated into *TG1*. The returning nodes as well as joining nodes of *TG2* have been unified with those of *TG1*.



(d) The final task graph. The function invocation in *TG1* has been completely replaced by *TG2*. Every declaration has been eliminated in *TG2*.

Figure 3.

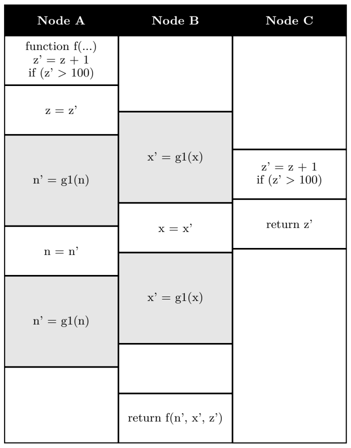


Figure 4. Resulting schedule after applying a suitable scheduling algorithm to the task graph from figure 3(d) (branch tasks have been omitted). Gray tasks are identical in both *TG1* and *TG2* and have been scheduled on the same processing node. Their outcomes can therefore be cached and reused in subsequent function calls.

We can exploit this characteristic by looking at the resulting schedule (figure 2.5)[2]. Several (not all) tasks have a duplicate in the expanded task graph. If the scheduling heuristic decided to place both the original task (initially from *TG1*) and the cloned task (initially from *TG2*) on the same processing node, we can conclude that overall runtime would benefit from reusing the corresponding task during function execution. Along with this, we can keep the task's outcome local during execution and only have to transfer the result when the recursion terminates.

This is most efficient for tasks whose predecessor is the declaration task and whose successor is the invocation task. Tasks which are preceded or succeeded by other tasks than those mentioned can not be (completely) reused, as they might rely on intermediate results of other tasks or produce intermediate results for other tasks. In this case, data transfer can not be omitted.

Altogether, determination of reusable tasks/variables is reduced to two key steps:

1. For each task in *TG1*, find its counterpart in *TG2*.
2. If both tasks have been scheduled on the same processing node and the task in *TG1* is proceeded by the declaration task and succeded by the invocation task, the task can be reused and its outcome must not be sent during function execution.

3. Experiments

To test how overall performance of recursive functions can benefit from our approach, we implemented it in the funky-Imp [3] compiler and applied it to two use-cases: An endrecursive variant of a 2D jacobi iteration and a non-endrecursive

[2] Theory teaches us that the scheduling problem itself is generally NP-hard [2]. The results of our approach heavily depend on the used heuristic which is used to approximate the schedule. Unfortunately, this schedule might be not optimal [11], so our algorithm at this point might fail in detecting all possible reusability options.

[3] http://www2.in.tum.de/funky

Listing 1. A recursive version of the Jacobi-iteration.

```
int[] step(int t[], int mt[], int m[], int
    mb[], int b[], double err)
{
  int t_[] = jt(t, mt, m);
  int b_[] = jb(b, mb, m);
  int m_[] = jm(mt, m, mb);

  double dt = delta(t, t_);
  double db = delta(b, b_);
  double dm = delta(m, m_);

  double d = max(dt, db, dm);

  if (d < err)
  {
    return merge(t_, b_, m_);
  }
  else
  {
    return step(t_, mt, m_, mb, b_, err);
  }
}
```

Listing 2. A non-endrecursive counting function with array manipulation.

```
int f(int n[], int x[], int z)
{
  int z_ = z + 1;
  if (z_ > 10000)
  {
    return z_;
  }
  else
  {
    int n_[] = g(n);
    int x_[] = g(x);
    return f(n_, x_, z_) + 1;
  }
}
```

counting-function which modifies two integer-arrays. Both programmes were automatically translated to C++ with and without applying our optimization. The resulting code was then profiled on a MPI linux cluster with 64 nodes connected via Infiniband using the mpip-library [12].

3.1 Endrecursive 2D-jacobi iteration

When recursion is used in functional programming languages like e.g. Scheme, endrecursive functions (also known as tailrecursive functions) are preferred, as they usually allow the runtime environment to execute the recursion in constant space [3]. To test the performance of our approach for endrecursive functions, we used a recursive version of the well-known 2D-jacobi iteration. As input for the function a square matrix with 10^2, 10^3, 10^4 and 10^5 elements is used. To allow parallel computations, the matrix is divided into five sections, namely t, m_t, m, m_b and b (see figure 5). Listing 1 provides the sourcecode for the function.

The step-function is recursively called until the difference of the matrix values compared to the iteration before falls below an error-value. During an iteration, new values for each of the sub-matrices are calculated in parallel. The sub-functions $jt()$ and $jb()$ calculate new values for the sub-matrices t and b. The arrays m_t and m_b are updated by

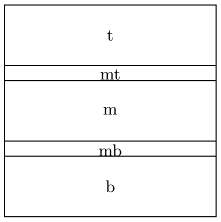


Figure 5. Structure of the input matrix for the Jacobi-iteration. The sub-matrices t, m and b are processed in parallel. m_t and m_b must be transferred in each recursive step.

reference in the corresponding sub-functions $jt()$ and $jb()$ by means of m.

The funkyImp compiler automatically divides the function into tasks and determines parallel execution paths. Our algorithm is applied to the function's task graph and determines, which parts of the matrix can be computed node-locally without the need of transferring intermediate computations to the node that initiates another recursive invocation. Depending on the matrix size, this might lead to a significant decrease in the overall transferred data, hence leading to an increase in runtime performance.

3.2 Non-endrecursive functions

We also tested our approach for non-endrecursive functions, as they are usually not optimizable using existing methods like tail-call optimization. The benchmark function for this test is shown in listing 2. The critical part regarding the MPI implementation is the modification of the arrays n and x. The function g simply iterates over the provided array and increments the value of each element. While this can be done simultaneously for both arrays, they have to be transferred to their corresponding processing node in each recursive step the arrays. After the calculation, the modified arrays have to be transferred to the node which invokes another function call, which would cause a huge amount of communication overhead.

Like for the endrecursive scenario, the funkyImp compiler automatically partitions the function into tasks, which are analyzed by our algorithm afterwards. In theory, the algorithm should decide that the arrays n and x can reside on their corresponding nodes during the recursion, which saves us from transferring them in every recursive step. We test the performance of the function with arrays of size 10^2, 10^4, 10^5 and 10^6 elements.

Due to the non-endrecursiveness of the function, the call stack increases during the function execution. In our case, this means that in every recursive step, another processing node is requested and blocked until the function returns. This may lead to deadlocks, when the amount of function invocations outnumbers the available processing nodes. To avoid this, we limit the recursion depth and stop allocating additional nodes when no free nodes are available. The recursion then continues locally on the last active node.

4. Results and discussion

For both the endrecursive and non-endrecursive test case we expected a decrease in overall program runtime and also a decrease in the overall transferred data. Our findings basically confirm this. However, some configurations perform even better than expected, where others fall short of our expectations.

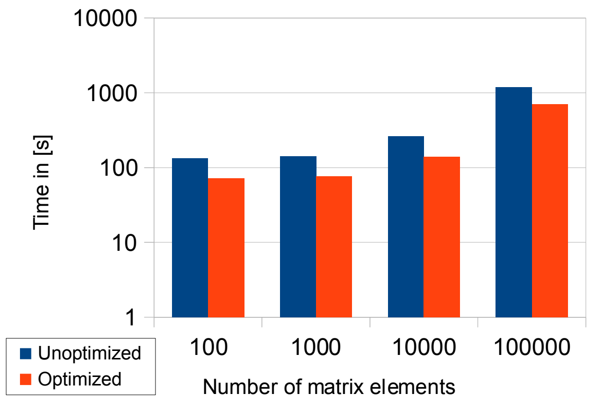
The runtime performances of both non-optimized and optimized versions of the Jacobi-program are shown in figure 6(a) and figure 6(b). As expected, the optimized version is generally faster than the unoptimized version and reduces the overall amount of data which is transferred during program execution. Precisely, we could observe a speedup between 1.69 and 1.85 and a reduction in the transferred data volume from 21.2 % up to 69.6 %[4]. Furthermore, the data transfer savings positively correlate with the matrix size. Unfortunately, this correlation doesn't hold for the speedup. We observed a constant speedup of 1.85 to 1.83 until a matrix size of 10^4. For 10^5 matrix entries the speedup decreased to 1.69. A possible explanation for this behaviour will be provided later. For the non-endrecursive example, the results behave similar. Figure 6(c) and figure 6(d) show the runtime differences and the differences in the transferred data volume. Again we could observe a decrease in overall runtime and data volume. Compared to the Jacobi-program, the effect of our optimization is significantly bigger. The speedup ranges from 1.48 up to 16.08 and the transmission costs could be reduced by 60 % in the worst and 98.4 % in the best case.

The fact that the non-endrecursive program profits more from the optimizations than the endrecursive Jacobi-program may seem surprising, but can be explained due to its simpler communication structure. After applying our algorithm to the task graph of the Jacobi-program, it suggested the variables t, b as well as mt and mb for reuse at their corresponding nodes. At first glance, this might result in the overall communication decreasing significantly, but this is not true. When considering listing 1 again, we notice several interdependencies between the tasks. For example, the statements in line 3 and 4 depend on the variable m. The function $jm()$ on the other hand has dependencies to the variables mb and mt to be able to update m. So in each recursive step, m must be sent three times (to $jt()$, $jb()$ and also to the return statement, as m was not marked for reuse by the algorithm) and mb as well as mt must be sent once. While mb and mt are small (see figure 5), m is considerably large. We noted before that the overall speedup for 10^5 matrix elements decreased compared to smaller matrix sizes. Obviously, the communication overhead created by m, mt and mb has a noteworthy negative effect when the data exceeds a certain size. However, the precise figures for these limits have to be further investigated.

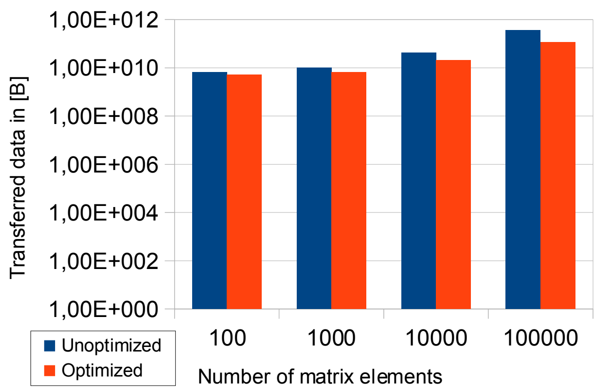
For the non-endrecursive program, the behaviour is different. Our algorithm was able to detect n and x for reuse at their corresponding nodes. When we consider listing 2, no further interdependencies exist between the statements. Consequently, the transmission of n_ respectively x_ can be completely omitted during the recursion. Instead, only small synchronization messages are sent which indicate another recursive step (see section 5 for details). As these notification messages are constant in size, a superlinear speedup could be achieved and the overall transferred data reduced tremendously.

From the results we can conclude that the communication structure and the task interdependencies of the input program have great influence on the effectiveness of our optimization technique. Therefore it is still up to the programmer to find a good partitioning of the data, so that the

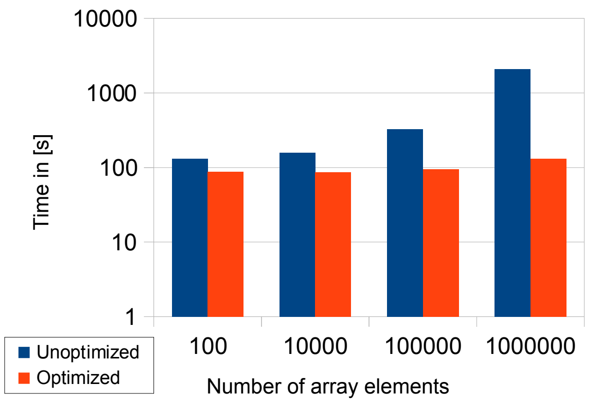
[4] The figures relate to 10,000 iterations of the program.



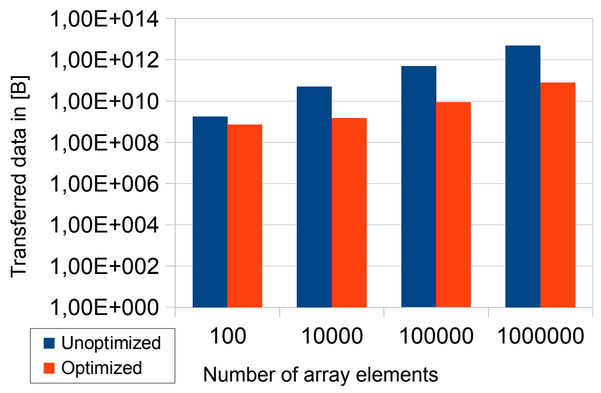
(a) Overall program runtime of the unoptimized and optimized Jacobi-program (after 10,000 iterations).



(b) Overall transferred data in the unoptimized and optimized version of the Jacobi-program (after 10,000 iterations).



(c) Overall program runtime of the unoptimized and optimized non-endrecursive program (after 10,000 iterations).



(d) Overall transferred data in the unoptimized and optimized version of the non-endrecursive program (after 10,000 iterations).

Figure 6.

algorithm is able to exploit data locality efficiently. Further we are able to optimize even non-recursive functions with our approach, which is a novelty.

5. Implementation details

Our implementations are generally based on a traditional master/worker pattern as described in [6], extended by a dynamic runtime scheduler which is aware of all available processing nodes. Consequently, we follow the MPMD (multiple program multiple data) paradigm, where this scheduler is realized as a separate program. By convention, the scheduler has rank 0 in the MPI environment. The main program consists of all available tasks (realized as classes) and a central loop which waits for incoming messages. From this loop, the individual tasks are spawned. The process with rank 1 starts the main program, so it can be regarded as the initial master. As a result, the tasks with rank 2 to N are available as workers.

When a master requires a worker, it has to send a request message to the scheduler, which replies with the rank of a free worker process. The master then sends an initial message to the worker, containing the actual method-context (e.g. parameters) and an indicator for the desired task (via an MPI message tag). The whole communication between the master and the worker(s) runs completely autonomous from the scheduler, which keeps the organizational overhead low. After the calculation finished and the worker sent its results back to the master, the worker sends a release message to the scheduler, indicating that it is available for other tasks to execute.

In the optimal case, a task is initiated at the beginning of the recursion and runs independently until the recursion terminates. But because tasks may be arbitrarily fine grain, they might not have any information about the state of the recursion (for example, whether the termination condition evaluates to true or false). Therefore, it is neccessary to exchange synchronization messages which indicate whether the recursion has terminated or not. These messages are sent by the task which is in charge of the recursive invocation and is therefore aware of the termination condition. As this synchronization messages only contain an indicator whether to do another calculation or not, these are much smaller than actual data messages. Listing 3 shows how the code skeleton for a optimized task looks like.

Listing 3. Code skeleton for an optimized task in the resulting MPI program. Lines 10 to 17 show how the task/-variable reuse is realized.

```
   void taskname::execute(char* buffer, int
       source)
 2 {
       //establishing method context from buffer,
           e.g. receive parameters
 4     ...

 6     REUSE_LOOP_LABEL:
       //do calculations
 8     ....
       //wait if next iteration neccessary
10     bool iterate;
       MPI::COMM_WORLD.Recv(&iterate, 1,
           MPI::BOOL, MPI_ANY_SOURCE, NOTIFY_REC);
12     if(iterate == true)
       {
14         //code for updating local variables
           ...
16         goto REUSE_LOOP_LABEL;
       }
18
       //release this task
20     int mpi__release__var;
       MPI::COMM_WORLD.Ssend(&mpi__release__var, 1,
           MPI::INT, SCHEDULER, RELEASE_TAG);
22 }
```

6. Related work

To the best of our knowledge, universal approaches to bring recursion (and its optimization) to message passing environments have not been described so far in literature. However, some researchers have addressed special issues in the SPMD programming model, which are kind of related to our work.

For example Nishimura and Ohori [10] presented a calculus for exploiting data parallelism on recursively defined data already 20 years ago. In their paper, they describe a "new form of recursion" for recursively defined data structurs (e.g. lists or trees), called "parallel recursion", which allows parallel processing of this kind of data. They also provide a first implementation of their approach in a SPMD-like style. However, their method is focused on recursively defined data structures, so e.g. matrix operations would be impossible with it.

Another approach comes from Kamil and Yelick [9], who present an implementation for divide and conquer algorithms in SPMD environments, which they call RSPMD (recursive single program multiple data). They follow a hierarchical team mechanism, in which all processes initially belong to a global pool (or team). This pool can be recursively subdivided into smaller teams, which solve simple problems by themselves and reintegrate during a merge phase. While this approach fits well for algorithms which apply the divide and conquer pattern, it is again not suitable for general recursion. Additionally, it does not allow for automatic optimizations, but must be applied manually by the programmer.

Finally, there is the work from Huo, Krishnamoorthy and Agrawal [8], who describe an idea which has certain similarities to our approach, but focus on modern GPU architectures (SIMD) instead of message passing environments. They improve performance of recursive applications on GPUs by better hardware-level thread scheduling. Basically, they improve the reconvergence method of threads to achieve a noteworthy speedup, but they don't take data localities into account.

7. Conclusion

Even though common problems in science can be formulated nicely using recursion, this programming style is hardly used in practical MPI programs due to loss of parallelization and bad performance. In this paper, we presented an approach to automatically optimize recursive functions for the use in MPI environments by taking data locality into account and reuse intermediate calculations during the recursion. Starting from an ordinary task graph, we described how to obtain an expanded and modified task graph, which represents the communication structure of the recursion. After scheduling this task graph by means of a standard communication-aware scheduling algorithm, we were able to detect and reuse temporary calculations during recursion.

By implementing our algorithm in an existing compiler, we were able to show that for both endrecursive and non-endrecursive functions, our method leads to performance benefits and a significant reduction of the MPI communication overhead. Depending on the communication structure of the application, speedups of more than factor two could be achieved.

Altogether, our approach allows programmers to exploit the convenience of recursive problem modeling without making concessions in terms of performance.

References

[1] K. Asanovic, R. Bodik, and B. Catanzaro. The landscape of parallel computing research: A view from Berkeley. 2006.

[2] P. Brucker. *Scheduling algorithms*, volume 3. Springer Berlin Heidelberg, 4th edition, 2004.

[3] W. D. Clinger. Proper tail recursion and space efficiency. *ACM SIGPLAN Notices*, 1998.

[4] J. K. Drury and B. F. Bonney. *Using MPI-2: Advanced Features of the Message-passing Interface*. Falcon Series. Globe Pequot Press, 2005. ISBN 9780762728206.

[5] T. M. P. I. Forum. MPI: A Message Passing Interface Standard. Technical report, 1994.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Number Bd. 1 in Scientific and engineering computation. MIT Press, 1999.

[7] A. Herz and C. Pinkau. Real-World Clustering for Task Graphs on Shared Memory Systems. 2013.

[8] X. Huo, S. Krishnamoorthy, and G. Agrawal. Efficient scheduling of recursive control flow on GPUs. *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13*, page 409, 2013.

[9] A. Kamil and K. Yelick. Hierarchical Computation in the SPMD Programming Model. *26th International Workshop on Languages and Compilers for Parallel Computing*, 2013.

[10] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data. *Theory and Practice of Parallel Programming*, 1994(Lncs 907), 1995. URL `http://link.springer.com/chapter/10.1007/BFb0026582`.

[11] O. Sinnen. *Task scheduling for parallel systems*. Wiley Series on Parallel and Distributed Computing. Wiley, 2007.

[12] J. Vetter and C. Chambreau. mpip: Lightweight, scalable mpi profiling, 2005. URL `http://mpip.sourceforge.net/`.

[13] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30–44, June 1991.