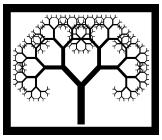
Paper 7



Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

Adjoining Hybrid Parallel Code

M. Schanen, M. Foerster, J. Lotz, K. Leppkes and U. Naumann
LuFG Informatik 12
Software and Tools for Computational Engineering
RWTH Aachen University, Germany

Abstract

OpenMP and MPI are typically used in combination to implement distributed shared memory parallelization schemes for numerical simulations on hybrid systems. The performance of said numerical simulations is frequently linked to the efficient and accurate computation of derivatives. One method of acquiring this derivative information for a given code is algorithmic differentiation (AD). AD implementations are subdivided into either source transformation or overloading tools. We investigate a generic approach of applying AD tools on hybrid parallel codes. A dense matrix-matrix multiplication serves as a case study.

Keywords: algorithmic differentiation, adjoint MPI, adjoint OpenMP.

1 Motivation

Numerical simulation software is generally run on multi-core parallel architectures. On these clusters, the number of nodes as well as the number of cores per node is steadily increasing. This trend implies hybrid parallelization schemes consisting of both distributed and shared-memory programming models. The de facto standard for distributed memory is the message passing interface (MPI) [5]. MPI is used to decompose the workload into large chunks which are distributed onto computer nodes (e.g. grid partitioning scheme). Additionally, each node is composed of several cores that access the same memory locations over a common physical memory. Hence, we assume that the core numerical problem, called *kernel*, is distributed among the nodes through MPI. On each node the kernel is assumed to use OpenMP [8] for shared-memory parallelization.

Numerical simulation and optimization typically rely on robust and efficient derivative information, thus potentially favoring AD [4] over finite difference approximation

in order to avoid truncation and rounding errors. There exist two distinct derivative models explained in more detail in the following section. First, the tangent-linear model based on the straightforward application of the chain rule. And second, the adjoint model resulting from the associativity of the chain rule. AD applies the two models semi-automatically by transforming a given original code into its derivative equivalent where in addition to the values, derivatives are computed. Thus, a potentially tedious implementation of the derivative code by hand is avoided.

Unfortunately, no existing AD tool is able to generate the derivative code of a hybrid parallel implementation automatically. In the following, we use both categories of tools (source transformation and overloading) to implement the adjoint derivative model. At runtime, crucial information for adjoining OpenMP pragmas is missing. Therefore only a source transformation tool (e.g. compiler) parsing these pragmas, is able to adjoin OpenMP code. If MPI is used in a kernel of an application it can be adjoined using dcc and in fact this was subject to a publication at EuroMPI2010 [10]. However, parsing the entire code using an AD tool is a difficult task that no tool has ever completely achieved or even strived for, since the additional effort far outweighs the benefits. As MPI resides mostly on a higher layer of an application, this is in particular true for adjoint MPI. Hence an overloading AD tool is used for adjoining MPI.

To motivate and illustrate our approach, we implemented a distributed dense matrix multiplication based on the Cannon algorithm [2]. Being a well documented problem in parallel programming, this example serves as an emulation of large-scale simulation codes, covering both the distribution of the input problem using MPI as well as a local computation of a kernel using OpenMP.

We consider a dense matrix multiplication $C = A \cdot B$, $A \in \mathbb{R}^{m \times p}, B \in \mathbb{R}^{p \times n}$ and $C \in \mathbb{R}^{m \times n}$. To avoid infeasible memory consumption for large m, p and n, the Cannon algorithm divides the matrices A, B and C into block matrices $A_{i,k}$, $B_{k,j}$ and $C_{i,j}$. Each block is computed according to $C_{i,j}$+=$A_{i,k} \cdot B_{k,j}$. The $C_{i,j}$ are incremented locally on each node, while the block matrices $A_{i,k}$ and $B_{k,j}$ are switched among nodes as described by Cannon's algorithm using two MPI grid topologies for A and B. The block matrices are communicated using blocking MPI_sendrecv_replace calls. Each node uses OpenMP shared-memory multithreading for computing the local product of one block for a given i, j.

The structure of the paper is as follows. Section 2 gives a short introduction to AD. In Section 3 we present the integration of the adjoint MPI library and discuss the adjoining of the OpenMP region. We then describe the coupling of two parallelization schemes along the adjoint version of the code. Section 4 presents the results of our case study in order to validate our approach.

2 Algorithmic Differentiation

We assume that numerical code implements multivariate vector functions $\mathbf{y} = F(\mathbf{x})$, F: $\mathbb{R}^n \to \mathbb{R}^m$. The *tangent-linear* model of F computes the directional derivative $\dot{\mathbf{y}} = \nabla F(\mathbf{x}) \cdot \dot{\mathbf{x}} = \dot{F}(\mathbf{x}, \dot{\mathbf{x}})$ of the outputs $\mathbf{y}$ with respect to the inputs $\mathbf{x}$ at the current point and for a given direction $\dot{\mathbf{x}} \in \mathbb{R}^n$. Exploitation of the associativity of the chain rule yields the *adjoint* model of F computing adjoints of the inputs $\bar{\mathbf{x}} = \nabla F(\mathbf{x})^T \cdot \bar{\mathbf{y}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) \in \mathbb{R}^n$ for given adjoints $\bar{\mathbf{y}} \in \mathbb{R}^m$ of the outputs.

The difference in complexity between these two modes is the runtime for accumulating the entire Jacobian ∇F. While the tangent-linear model needs n evaluations of $\dot{\mathbf{y}}$ for $\dot{\mathbf{x}}$ set equal to each Cartesian basis vector in $\mathbb{R}^n$, the adjoint model needs m evaluations of $\bar{\mathbf{x}}$ for $\bar{\mathbf{y}}$ set equal to each Cartesian basis vector in $\mathbb{R}^m$. For numerical codes where the number of inputs $\mathbf{x}$ far exceeds the number of outputs $\mathbf{y}$, the difference in runtime complexity $Cost(\dot{F}) = \mathcal{O}(n) \cdot Cost(F)$ versus $Cost(\bar{F}) = \mathcal{O}(m) \cdot Cost(F)$ may be crucial. $Cost(F)$ denotes the computational cost of a single function evaluation of F.

Technically, the adjoint code is split up into a *forward* and a *reverse section*. The forward section computes the function values following the same data flow as the original code. The reverse section computes the adjoints along the reverse of the original data flow. The control flow of the forward section needs to be traced. Moreover, values that are overwritten in the forward section may later be needed in the reverse section in order to compute the adjoints and, therefore, must be recorded. This is not the case in tangent-linear code being a rather straight-forward assignment-level augmentation of the original code. Hence, the development of adjoint differentiation tools is a more complex task than the implementation of the tangent-linear model. This paper will focus on the application of the adjoint model.

The matrix multiplication $C = A \cdot B$, with $A \in \mathbb{R}^{m \times p}$, $B \in \mathbb{R}^{p \times n}$ and $C \in \mathbb{R}^{m \times n}$ involves $p(n+m)$ inputs as opposed to $n \cdot m$ outputs. The ratio between the number of inputs and the number of outputs grows with p. We assume $p \gg m, n$.

Example: We consider the computation of a Jacbian matrix line $\frac{\partial c_{0,0}}{\partial [a_{i,k}, b_{k,j}]}$ of the dense matrix multiplication

$$A \cdot B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 15 & 22 \end{bmatrix} = C$$

in adjoint mode.

The computation of the matrix C represents the forward section in the adjoint mode. In the reverse section the adjoints $\bar{A}$ and $\bar{B}$ are *accumulated*. To accumulate $\frac{\partial c_{0,0}}{\partial a_{i,k}}$ and $\frac{\partial c_{0,0}}{\partial b_{k,j}}$ we set $\bar{c}_{0,0}$ equal to 1. By adjoining the computation of $c_{i,j}$ according to the adjoint model we obtain

$$\begin{aligned}
\bar{a}_{0,0} &= b_{0,0} \cdot \bar{c}_{0,0} + b_{0,1} \cdot \bar{c}_{0,1} = 1 \cdot 1 + 2 \cdot 0 = 1 \\
\bar{a}_{0,1} &= b_{1,0} \cdot \bar{c}_{0,0} + b_{1,1} \cdot \bar{c}_{0,1} = 3 \cdot 1 + 4 \cdot 0 = 3 \\
\bar{a}_{1,0} &= b_{0,0} \cdot \bar{c}_{1,0} + b_{0,1} \cdot \bar{c}_{1,1} = 1 \cdot 0 + 2 \cdot 0 = 0 \\
\bar{a}_{1,1} &= b_{1,0} \cdot \bar{c}_{1,0} + b_{1,1} \cdot \bar{c}_{1,1} = 3 \cdot 0 + 4 \cdot 0 = 0
\end{aligned}$$

Listing 1: Matrix multiplication with OpenMP directives serving as input to dcc.

```
void MxM(int n, int m, int p, double *A,
         double *B, double *C)
#pragma ad indep A B
#pragma ad dep C
{
   int i=0;
   int k=0;
   int Aidx=0;
   int Bidx=0;
   int C_size=m*n;
#pragma omp parallel for private(i,k,Aidx,Bidx)
    for(i=0;i<C_size;i++) {
#pragma ad simple loop
        for(k=0;k<p;k++) {
            Aidx=(i/n)*p+k;
            Bidx=(i%n)+k*n;
            C[i]=C[i]+A[Aidx]*B[Bidx];
        }
    }
}
```

and similarly $\bar{B}$.

Note that the adjoints $\bar{A}$ and $\bar{B}$ are evaluated in one computational run of the adjoint model. Thus the accumulation of the entire Jacobian ∇F would amount to 4 runs by setting one single entry $\bar{c}_{i,j}$ of $\bar{C}$ equal to 1 at each run. The tangent-linear model however would need 8 evaluations since each direction $\dot{a}_{i,k}$ and $\dot{b}_{k,j}$ would have to be set equal to 1 individually. $\bar{A}$ and $\bar{B}$ may be computed analytically with $\bar{A} = \bar{C} \cdot B^T$ and $\bar{B} = A^T \cdot \bar{C}$. This is considered to be an implementation by hand of the adjoint code as opposed to AD.

2.1 AD by Source Transformation using dcc

Source transformation tools take an implementation of a function F as an input and generate the implementation of $\dot{F}$ or $\bar{F}$ in tangent-linear or adjoint mode, respectively. Therefore, source transformation tools need to be able to parse the entire computer language in order to generate adjoint code. Our source transformation tool is the derivative code compiler dcc [9]. Its focus is on the efficient derivative code generation while supporting a well defined subset of C/C++ as an input code. The advantage of such a compiler is the access to extra information about the input code retrieved through program analysis techniques [1] at compile time. Moreover, the user may provide additional information through pragmas in the source code. These pragmas declare certain code properties. In Listing 1 we have two kinds of pragmas. On the one hand, the OpenMP pragmas starting with '**#pragma omp**', and on the other hand the AD related pragmas starting with '**#pragma ad**'. The pragmas in line 3 and line 4 declare that the derivative code should compute the derivative of output C (dependent

variable) with respect to inputs A and B (independent variables). In line 13, the user declares the subsequent loop as simple. This tells the compiler that the order of the loop counter is defined only by the loop itself and the counter is not modified during the evaluation of the loop body. This is important for the reverse section because the loop is evaluated in the opposite order of the forward section. Therefore, the loop header of the forward section **for**(k=0;k<p;k++) becomes **for**(k=p−1;k>=0;k−−) in the reverse section. The OpenMP pragma in line 11 defines the following loop to be evaluated in parallel by a group of threads. Each thread has a private copy of the variables i,k,Aidx,Bidx. With Listing 1 as input, dcc transforms the signature into

```
void t1_MxM(int n, int m , int p, double* A,
    double* t1_A, double* B, double* t1_B,
    double* C, double* t1_C)
```

in tangent-linear mode and into

```
void a1_MxM(int mode, int n, int m, int p,
    double* A, double* a1_A, double* B,
    double* a1_B, double* C, double* a1_C)
```

in adjoint mode.

Each floating-point variable v is augmented by t1_v (directional derivative) or a1_v (adjoint). The only difference in the signature between the two modes is the parameter mode. This is used for the interprocedural *joint reversal* scheme [7] by dcc for storing the function's arguments. Mode 2 is identical to the call of the original function except that the arguments are store before the function call. Mode 1 consists of restoring the arguments and evaluating first the augmented forward section and then the reverse section. This way the reverse section is always called immediately after the forward section. Another reversal scheme mostly used by overloading tools is *split reversal*, where the execution of the reverse section may not immediately follow the forward section. Joint reversal has higher memory efficiency than split reversal, whereas split reversal has a lower computational cost than joint reversal. Optimal reversal is known to be NP-complete [6] and is not subject of this paper.

Based on the code presented in Listing 1 we used dcc to generate t1_MxM in tangent-linear mode and a1_MxM in adjoint mode. For $m = 100$ and $n = 100$ we measured serial run times of the Jacobian accumulation for various p in Figure 1. With increasing number of inputs $p \cdot (m + n)$ the computational cost of $F, \dot{F}$ and $\bar{F}$ is increasing linearly. However, for the Jacobian accumulation the computational cost of the tangent-linear code is increasing quadratically whereas the adjoint code is still increasing linearly. The logarithmic scale of Figure 1 illustrates this by a twice as steep slope for the tangent-linear mode as for the adjoint mode. The Jacobian runtime is extrapolated from the runtime of $\dot{F}$ or $\bar{F}$ times the number of inputs or outputs, respectively.

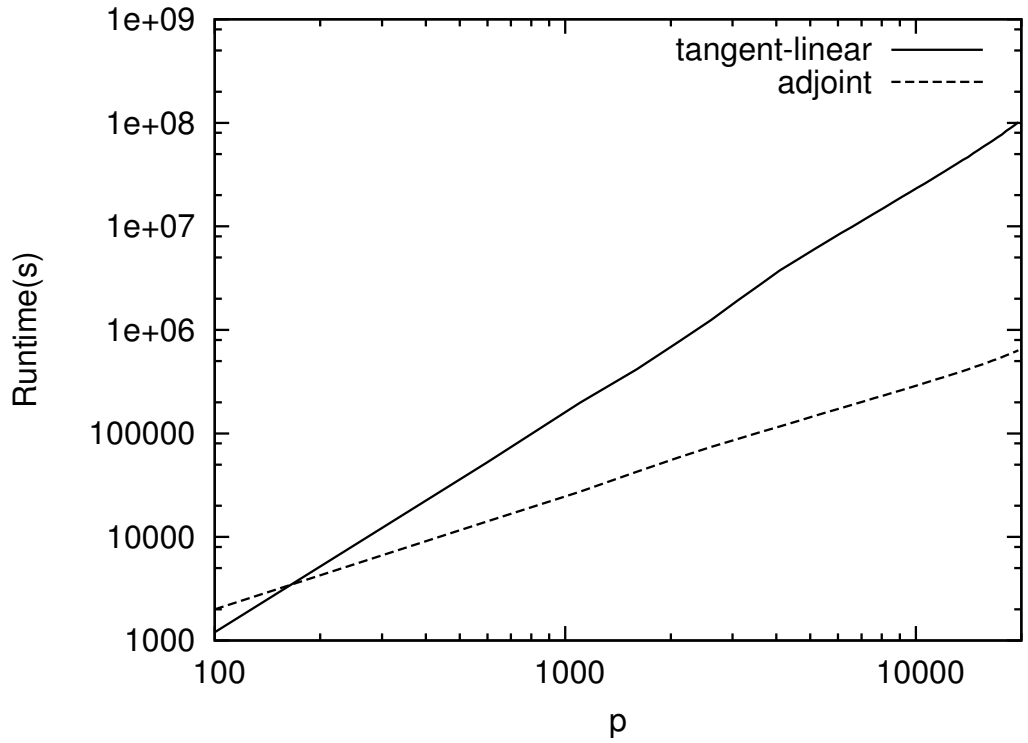


Figure 1: Jacobian accumulation runtime of a sequential matrix multiplication $A(m,p) \cdot B(p,n) = C(m,n)$ for various dimensions p and with $n = 100$, $m = 100$ using `dcc`.

2.2 AD by Overloading using dco/c++

AD by overloading is achieved at runtime by defining a new data type which overloads all operations used for variables of type double. The dco/c++ tape interpretation tool implements the overloading data type dco::a1s::type . It uses a data structure (tape) holding all necessary information for the adjoint projection. Additionally an adjoint value for each program variable is stored in the tape. The tape is being included and declared with

```
#include <dco.hpp>
dco::a1s::tape *global_tape;
global_tape=dco::a1s::tape::create();
```

To read the requested adjoints out of the tape at the end of the tape interpretation, we need to register the inputs. In our case study, the inputs are the distributed matrices A and B just before calling Cannon().

```
for(int i=0;i<m*p;i++)
    tape->register_variable(A[i]);
for(int i=0;i<p*n;i++)
    tape->register_variable(B[i]);
Cannon(m,p,n,A,B,C);
```

When the program is started, the values are computed and the tape is being recorded by executing the overloaded original code. This corresponds to the forward section. After the forward section has finished, the stored tape is interpreted in order to compute the adjoint projection of the recorded function. The adjoints seeded in the output variables in the distributed matrix C are propagated to the input variables A and B. This corresponds to the reverse section.

```
Cannon(m,p,n,A,B,C);
set_(C[0], 1.0,-1);
tape->interpret_adjoint();
```

As dco/c++ also implements the tangent-linear mode as well as higher-order models, the third parameter of set() is necessary for telling dco/c++ to set the adjoint component of C[0] accordingly. The tape interpreter propagates the adjoints through the entire tape, by assigning the corresponding adjoint to each variable for each operation using the information stored in the tape. After the tape interpreter returns, the adjoints of the registered variables are retrieved out of the tape with get(A[i],temp,−1), temp being a buffer for storing the adjoint.

3 Parallel Adjoints

Our proposed method of adjoining hybrid parallel code yields the generic call graph shown in Figure 2. Note that each MPI process will spawn its own call graph, differing only in its arguments and interprocess communication. First, the program starts in the top routine called *driver*. In the driver the dco/c++ tape as well as the AMPI

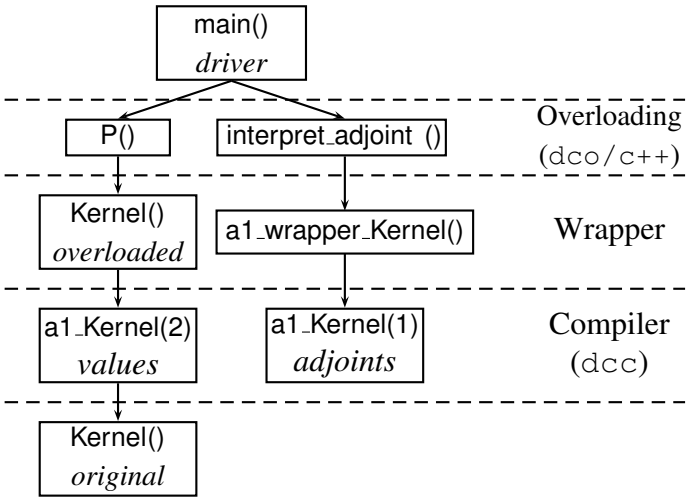


Figure 2: Call graph of an adjoined MPI program P (e.g. Cannon) relying on a numerical kernel (e.g. matrix multiplication) with OpenMP pragmas.

library is initialized followed by calling the overloaded program P, in our case Cannon , using the overloading type dco::a1s::type to record the process specific tape. In the overloaded program P a OpenMP parallelized *kernel* is called computing in our code a local matrix multiplication. This is implemented by calling the external function interface of dco/c++ with mode set equal to 2. This results in the execution of the original matrix multiplication computing only the values of matrix C. As described by the Cannon algorithm a1_MxM(2) may be called several times by the Cannon function depending on the number processes.

After the overloaded program P has finished the execution returns to the driver which starts the adjoint computation by calling the interpret_adjoint function of dco/c++. P (Cannon) is run in reverse off the recorded tape. Each time the adjoint matrix multiplication should be executed, the external wrapper function call is fired. This results in the a1_MxM(1) function called with mode set to 1. Eventually, the forward section and reverse section of the matrix multiplication is executed. We will now go through each layer of our implementation; MPI with dco/c++, OpenMP with dcc, and finally the coupling between these two layers.

3.1 Adjoint MPI by Overloading

Adjoint MPI (AMPI) is a self-contained library intended at adjoining MPI routines [11]. It is implemented in C and has been coupled with various AD tools. It is applicable to either source transformation or overloading tools and only incorporates the logic of reversing MPI communication. No numerical data is stored inside the adjoint MPI library. We will present the AMPI interface without going through the internals of the AMPI library.

As has been mentioned in the introduction, MPI is commonly used to distribute

subproblems on different nodes. Therefore MPI resides on a higher level in the implemented code. Hence, we assume that MPI is mostly differentiated by a tape interpretation tool as has been illustrated by our case study. If the adjoint MPI library is integrated into a tape interpretation tool, it has six external access functions needed to access tape information. These had to be implemented manually in `dco/c++`. In our Cannon implementation, the block matrices are exchanged using

```
MPI_Sendrecv_replace(local_A, n, MPI_DOUBLE, dest, 0, source, 0, comm, &status)
```

with local_A being the local block matrix, n its size, dest and source its source and destination respectively on the grid topology. To activate AMPI within a tape interpretation tool, we need to add the header ampi_tape.h and replace the MPI calls with its adjoinable counterparts

```
AMPI_Sendrecv_replace(local_A, n, MPI_DOUBLE, dest, 0, source, 0, comm, &status).
```

The AMPI routines are directly linked to the AMPI library, while executing the overloaded code. Since local_A is of arbitrary type, it may only be accessed via the external access functions from inside AMPI via a pointer of type **void**$*$. Values are read and written using

void ampi_get_val(**void** $*$local_A, **int** $*$i, **double** $*$x) and
void ampi_set_val(**void** $*$local_A, **int** $*$i, **double** $*$v).

AMPI needs to store the location *idx of the adjoints allocated in `dco/c++` for each buffer element buf[i] using

void ampi_get_idx(**void** $*$buf, **int** $*$i, INT64$*$idx).

The adjoints will be written to this location during the tape interpretation. In our case idx is of type **double**$*$, but may be set to any type through defines of INT64. Finally, AMPI writes an external AMPI function tape entry as well as tape entries for each buffer element buf[i] directly into the `dco/c++` tape.

void ampi_create_tape_entry(**int** $*$i) and
void ampi_create_dummies(**void** $*$buf, **int** $*$size).

During the tape interpretation, each time `dco/c++` reads an external AMPI function off the tape, it calls the AMPI interpretation routine

void ampi_interpret_tape().

This routine calls the adjoint computation of the original MPI call. In our case the adjoint of MPI_Sendrecv_replace will be

```
MPI_Sendrecv_replace(local_A, n, MPI_DOUBLE, source, 0, dest, 0, comm, &status).
```

The source and destination need to be switched while local_A now refers to the adjoints of local_A. These are again read and written using manually implemented external function calls in dco/c++.

void ampi_get_adj(INT64 *idx, **double** *x) and **void** ampi_set_adj(INT64 *idx, **double** *x).

The previously stored adjoint location idx is used to access the corresponding memory location in the dco/c++ tape. By implementing these eight external functions in dco/c++, we have enabled dco/c++ to interpret arbitrary code using MPI.

3.2 Compiler-Based Adjoint OpenMP

Based on [3], dcc reads in the parallel loop from Listing 1 line 4, exploiting the provided extra information about parallelism. Without going into the technical details we explain the output of dcc. In Figure 1, a1_MxM is first called by dco/c++ as an external function with mode set equal to 2 resulting in the evaluation of the original OpenMP enabled code as shown in Listing 1. When the dco/c++ tape interpreter interpret_adjoint calls a1_MxM with mode set equal to 1, the differentiated code in Listing 2 is executed. The forward and reverse section are merged under one single OpenMP parallel section. A parallel region for each section would imply the need to save the loop decomposition and its control flow for each thread. Using checkpoints on all the inputs to the parallel loops enables dcc to generate joint parallel regions, where in each iteration of OMP the reverse section immediately follows the forward section.

Two local stacks are defined in line 3 for storing values that are overwritten and needed in the reverse section. One stack is for integer values, the other one for floating-point values. For readability we used macros for the definition and for the stack operations. The stacks are defined locally to exploit memory locality. Memory efficiency-wise the stored values are consumed right afterwards in the reverse section.

The inner loop from line 6 to line 14 is similar to the original differing only in additional stack operations. Then follows the reverse section of the iteration, shown from line 16 to 25. The adjoint loop of a simple loop is again a simple loop, indicated by the pragma in line 16. Furthermore, the pragmas in line 19 and 21 ensure thread synchronization. This is necessary because the reversal of the data flow implies that all threads write to the adjoint arrays a1_A and a1_B in parallel. This race condition is solved by the **atomic** directive of OpenMP.

3.3 Coupling of Overloading Tool with Source Transformation Compiler

The coupling of a source transformation compilier with an overloading tool like dcc and dco/c++ requires a well defined interface for both tools. Our main goal is to minimize manual code manipulation while keeping both tools as generic as possible.

Listing 2: Adjoint parallel region generated by `dcc` corresponding to Listing 1 lines 11 to 19.

```
#pragma omp parallel for private(i,k,Aidx,Bidx)
for(i=0;i<C_size;i++) {
    OMP_LOCAL_INT_STACK;
    OMP_LOCAL_DOUBLE_STACK;
    // forward section
#pragma ad simple loop
    for(k=0;k<p;k++) {
        OMP_IDS_PUSH(Aidx);
        Aidx=(i/n)*p+k;
        OMP_IDS_PUSH(Bidx);
        Bidx=(i%n)+k*n;
        OMP_FDS_PUSH(C[i]);
        C[i]=C[i]+A[Aidx]*B[Bidx];
    }
    // reverse section
#pragma ad simple loop
    for(k=p-1;k>=0;k--) {
        OMP_FDS_POP(C[i]);
#pragma omp atomic
        a1_A[Aidx]+=a1_C[i]*B[Bidx];
#pragma omp atomic
        a1_B[Bidx]+=a1_C[i]*A[Aidx];
        OMP_IDS_POP(Bidx);
        OMP_IDS_POP(Aidx);
    }
}
```

In numerical codes, routines often implement common mathematical functions (e.g. matrix multiplication, dot product,...). These functions may be differentiated by hand or by a source transformation like `dcc`. In both cases we end up with one function computing the values and another one computing the derivatives. In `dcc` this distinction is made through the mode variable. From the `dco/c++` tape's perspective this is considered as an *external function*. In the forward section (tape recording) the external function is called and during its execution, the tape recording is suspended. Immediately after the external function returns, the tape recording is activated again. An external function data object has to establish the link between in- and outputs of the external function in the tape. Additionally all input data needed for the adjoint function has to be saved (*checkpointing*) as well as a pointer to the corresponding external adjoint function itself. The interface for calling an external function and the corresponding adjoint is done generically in `dco/c++`. The user has to provide correct wrapper routines.

`dcc` differentiated functions are treated by the `dco/c++` tape as external functions. The matrix multiplication wrapper function (MxM (...) – Listing 3) is called during the forward section. The wrapper has the same name as the original function, but is overloaded with the data type **atype** via the **typedef**. First an external function data object is created (line 4), which writes the checkpoint (line 5 - 7) and the connection to the inputs from the tape (line 12 - 17). This is followed by calling the `dcc` generated routine with mode=2 (augmented forward run – Section 2.1). In line 10 we allocate variables of type **double** in order to call the dcc generated routine. After the `dcc` routine has returned in line 19, we activate the output variables to create the link to the tape again. In line 24 we finally register the external function data object as well as a function pointer to the adjoint function wrapper in the tape.

During the tape interpretation the adjoint function wrapper (a1_wrapper_MxM (...) – Listing 4) is called with a fixed signature. It has two input arguments, the calling tape and the external function data object created during the forward section. As user-defined data objects are allowed, the incoming data type (ext_data_interface) has to be casted (line 3). The checkpoint is restored (line 6 - 8) and all needed variables are allocated (line 10 - 15). In line 19 the output adjoints are read from the tape and the `dcc` generated routine is called in mode=1 to compute the adjoint projection (line 22). The input adjoints are written back to the tape (line 24 - 31) and allocated memory is freed (line 33).

4 Results

All the benchmarks in this paper were conducted on a Sun SPARC Enterprise T5120 Server cluster consisting of Niagara T2 CPUs with each 8 cores and 32Gb of memory. The aim of the benchmarks is not to show the efficiency of the Cannon matrix multiplication. Our goal is to achieve similar scalability for the AD enabled code as for the original code. Two randomly generated matrices are read in from a file, multiplied and written out again. The size of the input matrices A and B is set to 64×200000

Listing 3: Matrix multiplication wrapper calling the dcc generated routine during the forward section of `dco/c++` (tape recording).

```
typedef dco::a1s::type atype;
typedef dco::a1s::tape::external_function_data_helper ext_data;
void MxM(int n, int m, int p, atype *A, atype *B, atype *C) {
  ext_data * user_data = new ext_data();
  user_data->write_to_checkpoint(n);
  user_data->write_to_checkpoint(m);
  user_data->write_to_checkpoint(p);

  double    pA[m*p],    pB[p*n],    pC[m*n],
         a1_pA[m*p], a1_pB[p*n], a1_pC[m*n];

  for(int i=0;i<m*p;i++)
    pA[i] = ext->register_input_from_tape(A[i]);
  for(int i=0;i<p*n;i++)
    pB[i] = ext->register_input_from_tape(B[i]);
  for(int i=0;i<m*n;i++)
    pC[i] = ext->register_input_from_tape(C[i]);

  a1_MxM(2, pA, a1_pA, pB, a1_pB, pC, a1_pC, n);

  for(int i=0;i<m*n;i++)
    C[i] = ext->register_output_in_tape(pC[i]);

  global_tape->register_external_function(&a1_MxM_wrapper, ext);
}
```

Listing 4: Adjoint matrix multiplication wrapper calling the generated adjoint dcc routine during the reverse section of `dco/c++` (tape interpretation).

```
void a1_wrapper_MxM(dco::a1s::tape *tape, int mode,
    ext_data_interface *data) {

  ext_data * user_data = static_cast<ext_data*>(data);

  int n, m, p;
  user_data->read_from_checkpoint(n);
  user_data->read_from_checkpoint(m);
  user_data->read_from_checkpoint(p);

  double *pA=new double[m*p];
  double *pB=new double[p*n];
  double *pC=new double[m*n];
  double *a1_pA=new double[m*p];
  double *a1_pB=new double[p*n];
  double *a1_pC=new double[m*n];

  for(int i = 0; i < m*p; i++) a1_pA[i] = 0;
  for(int i = 0; i < p*n; i++) a1_pB[i] = 0;
  for(int i = 0; i < m*n; i++)
    a1_pC[i] = user_data->read_next_adjoint_from_tape(tape);

  a1_MxM(1, n, m, p, pA, a1_pA, pB, a1_pB, pC, a1_pC);

  for (int i = 0; i < m*p; i++)
    user_data->write_next_adjoint_to_tape(a1_pA[i], tape);

  for (int i = 0; i < n*p; i++)
    user_data->write_next_adjoint_to_tape(a1_pB[i], tape);

  for (int i = 0; i < n*m; i++)
    user_data->write_next_adjoint_to_tape(a1_pC[i], tape);

  delete [] pA; delete [] a1_pA;
  delete [] pB; delete [] a1_pB;
  delete [] pC; delete [] a1_pC;

}
```

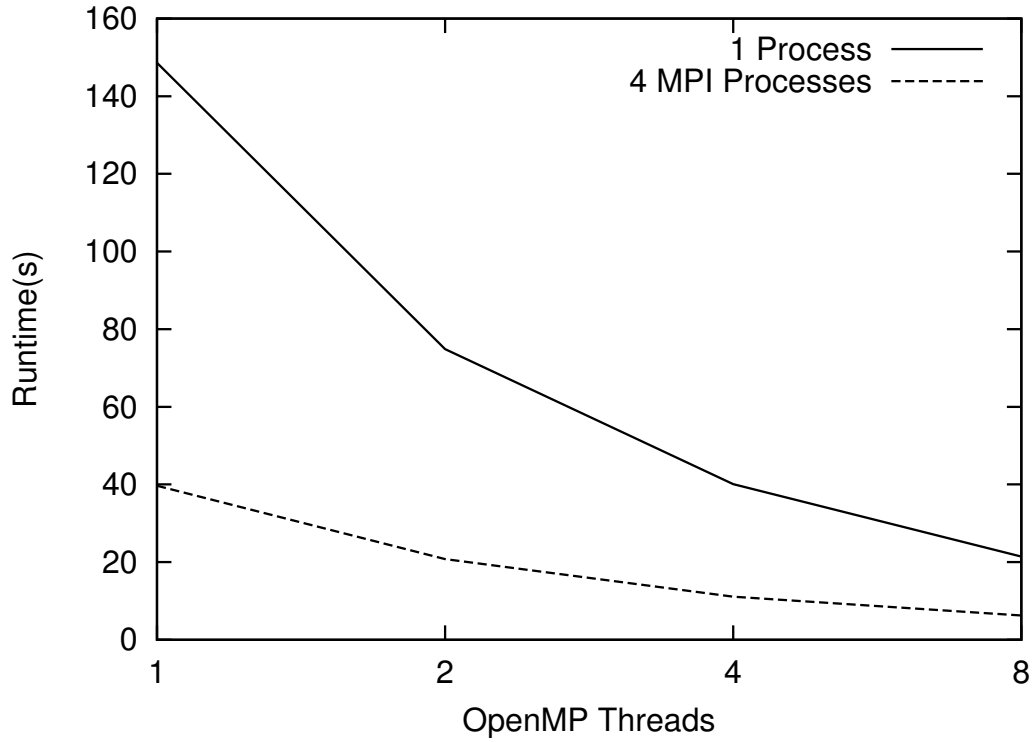


Figure 3: Runtimes for the hybrid Cannon matrix multipication with the size of the matrices set equal to A(64,200000),B(200000,64) and C(64,64).

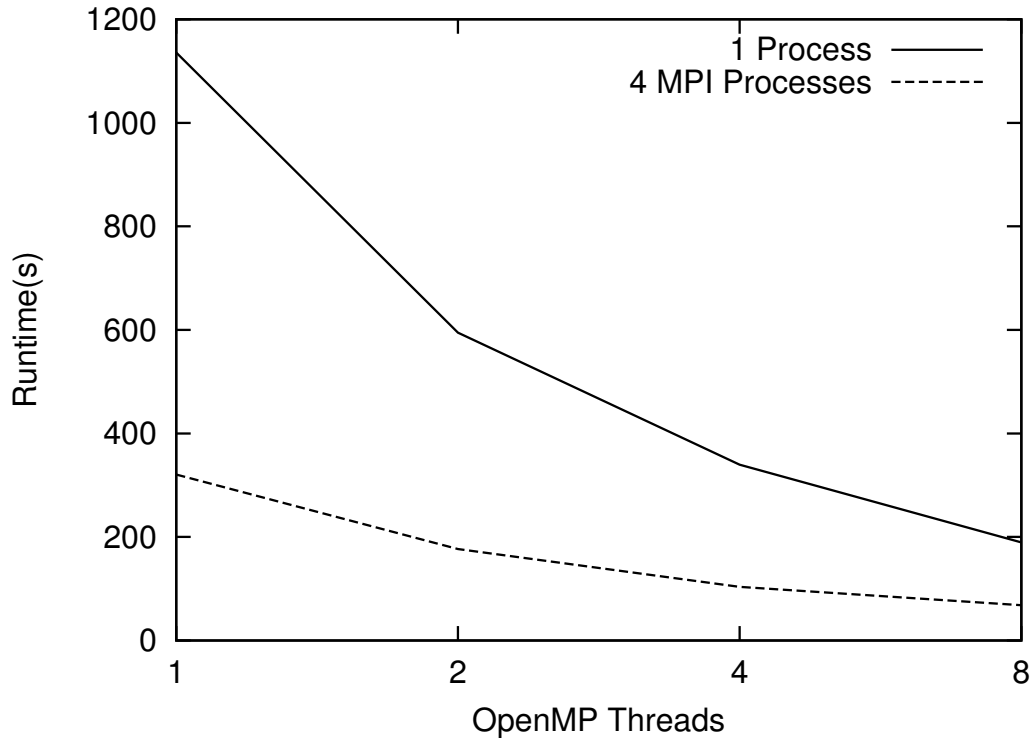


Figure 4: Runtimes for the adjoined hybrid Cannon matrix multipication using `dco/c++` and `dcc` with the size of the matrices set equal to A(64,200000),B(200000,64) and C(64,64).

and 200000×64, resulting in an output matrix C of size 64×64. This is a numerical problem where the inputs far exceed the number of inputs, thus suited for computing adjoints.

Figure 3 presents the run times obtained with the original hybrid code of the Cannon algorithm. Figure 4 are the run times of the AD differentiated code using `dco/c++` and `dcc`. While comparing the scalability, we observe that the curves' tendency exactly match. The differentiated code is a constant factor slower as the original code. Both with respect to MPI as well as with respect to OpenMP.

With 4 MPI processes and 8 OpenMP threads the runtime of the Jacobian accumulation of this problem would be equal to the number of outputs $m \cdot n$ times $68s$. This amounts to $278s$. Note that the accumulation of the Jacobian using e.g. finite difference would be equal to the runtime of one passive evaluation times the number of inputs. This is $6.2s$ for 4 MPI processes and 8 OpenMP threads times $p(m+n) = 25.6$ million, rendering the accumulation of the Jacobian using finite difference infeasible with current computer hardware.

5 Conclusion

This paper describes how a given hybrid parallel code may be adjoined using a source transformation and an overloading AD tool (`dcc` and `dco/c++`). MPI, being a library, is adjoined by overloading while OpenMP with its compiler directives is adjoined using source transformation. We conclude that this is the preferred approach for any hybrid parallel code. The OMP pragmas must be read by a source transformation tool. And in order to cover the entire C++ language the overloading tool is used to differentiate the Cannon implementation and its MPI calls. We explained how these two tools were linked together based the external function interface in `dco/c++`.

We validated our approach by providing benchmarks, comparing the scalability of the original code with the scalability of the differentiated code.

Acknowledgment

Michel Schanen is partially supported by the Fond National de la Recherche of Luxembourg under grant PHD-09-145.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison-Wesley, Reading, MA, 2007.

[2] L. E. Cannon. A Cellular Computer to implement the Kalman Filter Algorithm. 1969.

[3] Michael Förster, Uwe Naumann, and Jean Utke. Toward Adjoint OpenMP. Technical Report AIB-2011-13, RWTH Aachen, July 2011.

[4] A. Griewank and A. Walter. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (2nd Edition)*. SIAM, Philadelphia, 2008.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[6] Uwe Naumann. DAG reversal is NP-complete. *Journal of Discrete Algorithms*, 7(4):402 – 410, 2009.

[7] Uwe Naumann and Jean Utke. Source Templates for the Automatic Generation of Adjoint Code Through Static Call Graph Reversal. In Vaidy Sunderam, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science ICCS 2005*, volume 3514 of *Lecture Notes in Computer Science*, pages 337–383. Springer Berlin / Heidelberg, 2005.

[8] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2008.

[9] M. Schanen, M. Förster, B. Gendler, and U. Naumann. Compiler-based Differentiation of Numerical Simulation Codes. In *ICCGI 2011, The Sixth International Multi-Conference on Computing in the Global Information Technology*, pages 105–110. IARIA, 2011.

[10] Michel Schanen, Michael Förster, and Uwe Naumann. Second-Order Algorithmic Differentiation by Source Transformation of MPI Code. In *EuroMPI'10*, pages 257–264, 2010.

[11] Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. Interpretative Adjoints for Numerical Simulation Codes using MPI. *Procedia Computer Science*, 1(1):1819 – 1827, 2010. ICCS 2010.