Parallel Processing Letters
© World Scientific Publishing Company

A HYBRID SHARED MEMORY EXECUTION MODEL FOR A DATA PARALLEL LANGUAGE WITH I/O

CLEMENS GRELCK and STEFFEN KUTHE
Institute of Software Technology and Programming Languages, University of Lübeck
Ratzeburger Allee 160, 23538 Lübeck, Germany

and

SVEN-BODO SCHOLZ
Department of Computer Science, University of Hertfordshire
College Lane, Hatfield, AL10 9AB, United Kingdom

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT
We propose a novel execution model for the implicitly parallel execution of data parallel programs in the presence of general I/O operations. This model is called *hybrid* because it combines the advantages of the standard execution models fork/join and SPMD. Based on program analysis the hybrid model adapts itself to one or the other on the granularity of individual instructions. We outline compilation techniques that systematically derive the organization of parallel code from data flow characteristics aiming at the reduction of execution mode switches in general and synchronization/communication requirements in particular. Experiments based on a prototype implementation show the effectiveness of the hybrid execution model for reducing parallel overhead.

1. Introduction

Data parallel languages such as Hpf, Zpl [3], VectorPascal [5], Sisal [2], or SaC [21] represent one major approach to high-level parallel programming. Rather than providing programmers with explicit control over parallel execution, they introduce constructs for processing arrays that are particularly amenable to compiler-directed parallelization. Whether or not this opportunity is actually exploited in a specific context is up to the implementation. Details about concrete parallel or sequential realizations of these constructs are hidden. Fig. 1 illustrates this programming model by a simple pseudo code example. Each of the three `parfor` loops by itself may or may not be executed in parallel. However, program execution on the outer level is strictly sequential: The `parfor` loops and the scalar code in between them are expected to run in exactly the specified order.

A straightforward parallel implementation employs a dedicated *master thread* to execute the whole program essentially in a single-threaded manner. This measure automatically preserves the given execution order outside of data parallel constructs. Only when the master thread encounters a data parallel construct, it activates a number of *worker threads* to cooperatively compute only this individual instance of a data parallel construct. Having passed a subsequent synchronization barrier worker threads terminate or sleep while the master thread resumes sequential execution. This *fork/join* execution model nicely matches the semantics of the data parallel programming model and the intuition of the programmer. In other words, programming model (language semantics exposed to the programmer) and execution model (effective organization of program execution by compiler and runtime system) essentially coincide.

```
1    parfor i=0 to 100 { A[i] = ... ; }
2    print( "hello ");
3    A[x] = 42;
4    parfor i=0 to 100 { B[i] = ... A ...; }
5    print( "world");
6    parfor i=0 to 100 { C[i] = ... ; }
```

Figure 1: Pseudo code example mixing data parallel and I/O operations.

The simplicity of fork/join as a model of parallel programming is appealing. Likewise, its simplicity as an execution model facilitates parallel implementation. Unfortunately, the resulting code is suboptimal with respect to parallel performance: Any scalar code in between data parallel constructs enforces costly synchronization of all threads, the termination of the parallel execution environment, and its re-establishment shortly thereafter. We may re-arrange code such that data independent parallel constructs form larger regions of parallel execution, but data dependences like the one in line 3 limit the effectiveness of these measures.

Manually parallelized applications often implement an execution model called *SPMD* ("single program, multiple data"). In this model scalar code in between data parallel constructs is replicated: Each thread executes the same scalar code concurrently. While this does not improve performance per se, it reduces synchronization and communication overhead since the result of a replicated computation is immediately available to each thread for further processing. In the fork/join model a single thread had computed the same result in approximately the same time, but it would still need to broadcast the data to the other threads, which in turn must wait for the data to proceed with their work.

In principle, it is feasible to compile fork/join-style programs into SPMD-style executable code. However, the discrepancy between programming model and execution model incurs a subtle problem: correct handling of input/output operations and, more generally, operations that manipulate some externally visible state, e.g. the `print` statements in lines 2 and 5 of Fig. 1. In order to correctly implement the language semantics a compiler must preserve both the specified sequence of events as well as the single-threaded execution of each individual event. Regardless of the number of threads used for executing the code in Fig. 1 the words "hello" and "world" must appear exactly once and in exactly the given order.

If the execution model coincides with the programming model, i.e., both are fork/join, this property comes for free: All `print` statements outside of data parallel constructs are executed sequentially by the master thread, anyway. However, in the SPMD model, execution of scalar code is replicated. Consequently, the number of observable prints varies with the number of threads, and prints from different threads may interleave. In order to preserve language semantics the compiler must identify statements that manipulate an external state and protect their execution by a condition that ensures exclusive execution by a single thread. Furthermore, the compiler must introduce appropriate synchronization and communication operations, e.g., for the dedicated thread to broadcast data read from a terminal and for the other threads to wait for this data to arrive. *

In the presence of I/O the fork/join and the SPMD execution models become surprisingly similar: Both organize program execution as a sequence of alternating data parallel and single-threaded phases. The bulk of parallel overhead stems from switching from one execution mode to the other. The two execution models mainly differ in the way they handle scalar code that may (or may not) be replicated. Whereas the SPMD model replicates all of this code, the fork/join model replicates none. Starting from a fork/join execution model one aims at agglomerating parallel regions of code. Conversely, starting from an SPMD execution model one aims at agglomerating regions of single-threaded execution. In a sense the fork/join model and the SPMD model mark the two extremes of the same design space.

As is often the case with extremes, neither the fork/join model nor the SPMD model is optimal with respect to runtime performance. Therefore, we propose a hybrid execution model that aims at properly striking the balance between the two. We systematically distinguish between sequential, parallel, and replicatable code on the level of instructions. Rather than originating from one of the extremes of the design space, we start from an unbiased position and decide on the actual replication of replicatable code based on its context. As a consequence, the proximity to either the fork/join model or the SPMD model depends on an individual program's mix of operations and may differ from program part to program part.

The remainder of the paper is organized as follows: Section 2 introduces two abstract data parallel skeletons. We elaborate on our hybrid execution model in Section 3. Section 4 is concerned with code rearrangement techniques. We discuss experimental data and related work in Sections 5 and 6; Section 7 concludes.

2. Array Skeletons and their Implementation

For the context of this paper we consider two skeletons that capture the essential aspects of data parallel constructs in the field of array programming:

$$\texttt{GenArray(}\ shp,\ idx\ \rightarrow\ op_{idx}(\ idx,\ arg_1,\ \ldots,\ arg_n)\ \texttt{)}\quad ,$$
$$\texttt{FoldArray(}\ shp,\ idx\ \rightarrow\ op_{idx}(\ idx,\ arg_1,\ \ldots,\ arg_n),\ fold\_op,\ neutral\ \texttt{)}\quad .$$

The `GenArray` skeleton creates an array of shape *shp*, where *shp* must evaluate to a vector of non-negative integers. Elements of the new array are defined by a mapping

*Note that I/O statements that appear *inside* a data parallel construct are less of a concern. In this case, parallel execution does not affect the number of I/O events. It does, indeed, affect the order in which they occur, which may be different from program run to program run. However, this behaviour complies with the language semantics as the execution order in a data parallel construct is intentionally left unspecified.

of each legal index represented by *idx* to some operation op_{idx}. The operation op_{idx} may in fact specify different computations for each element of the index space; it may also accept an arbitrary number of arguments. Likewise, the `FoldArray` skeleton maps each element of an index space denoted by *shp* to the computation of a value. Rather than using these values for initialization of an array, they are reduced to a single value by applying a suitable folding operation. Both skeletons should be considered operational templates rather than concrete higher-order functions.

Assuming a fork/join-based parallel implementation of `GenArray` and `FoldArray` we have a dedicated master thread to execute our program in general. Whenever this master thread encounters a `GenArray` skeleton, it allocates storage for the result array (still in single-threaded mode). Afterwards, it writes the base address and the shape of the result array as well as the arguments of op_{idx} into a global broadcast buffer and activates the desired number of worker threads. All worker threads uniformly execute the same code, but can identify themselves through a unique ID. As a first step, a worker thread copies the data from the global broadcast buffer onto its private runtime stack in order to set up an appropriate execution environment. Then, each worker thread identifies a subspace of the entire index space defined by *shp*, based on its ID. For each element of its individual index subspace a worker thread computes the corresponding value and initializes the result array accordingly. After having completed their individual computations, the worker threads terminate. The master thread awaits the termination of all worker threads and, thereupon, proceeds with sequential execution.

The implementation of `FoldArray` is very similar to that of the `GenArray` skeleton. However, instead of having the master thread first allocate storage, each worker thread initializes a private accumulation variable with the neutral element of the folding operation and computes a partial fold result. Having completed its share of work a worker thread communicates this partial result to the master thread prior to termination. The master thread in turn waits for all partial fold results, computes the final result in a single-threaded manner and resumes sequential execution. A detailed presentation of the compilation schemes can be found in [14].

3. Towards a Hybrid Execution Model

There are basically two approaches to reduce runtime overhead without abandoning the fork/join execution model. Firstly, we may combine multiple data independent instances of skeletons into a single instance of some more versatile skeleton [13]. Secondly, we may improve the implementation of thread creation and termination, e.g. by caching worker threads during periods of sequential execution [14]. While the first approach reduces the number of costly synchronization and communication events, the second approach reduces the costs associated with each individual such event. If this optimization potential is thoroughly exploited, substantial speedups can be achieved [12,16]. Nevertheless, there are many non-artificial situations in which the conceptual constraints of the fork/join model effectively hinder successful parallelization. For example, in the code fragment

```
A = GenArray( [1000], idx -> op1(idx,U,V,W,X,Y,Z));
B = A[0];
C = GenArray( [1000], idx -> op2(idx,B,U,V,W,X,Y,Z));
```

the two skeletons computing `A` and `C` are separated by a trivial computation. Due to data dependencies we can neither move the scalar code in front of the first skeleton nor behind the second one. Hence, the parallel execution environment must be terminated after the computation of `A` and restarted for the computation of `C`. This is particularly dissatisfying as nearby skeletons typically share arguments.

Even without scalar code involved, the constraints imposed by the fork/join model may lead to dissatisfying results. For example, in the code fragment

```
A = GenArray( [1000], idx -> op1(idx,U,V,W,X,Y,Z));
B = GenArray( [1000], idx -> op2(idx,U,V,W,X,Y,Z));
C = GenArray( [1000], idx -> op3(idx,A,U,V,W,X,Y,Z));
```

the only data dependency is between the first and the third `GenArray` skeleton. This data dependency rules out the combination of all three skeletons into a single one, leaving us with the equally undesirable choices of synchronizing after computing `A` or after computing `B`. Instead of a traditional synchronization barrier we would prefer a scheme which separates notification of completion of some computation from waiting for completion of some computation. In the given example each thread could immediately signal completion of `A`. Rather than waiting for other threads to complete `A`, too, all threads could proceed with computing `B`. Only after having computed `B` as well, threads must wait for other threads to have completed computation of `A`. Separating barrier synchronization into a signal and a wait phase increases synchronization distances and reduces idle times.

Both examples have in common that they cannot be realized within the conceptual bounds of the fork/join model. Hence, we need a more complex execution model that allows us

- to execute non-skeletal code in a replicated manner by worker threads in order to reduce the number of expensive execution mode changes and

- to combine multiple skeletons with data dependencies in larger regions of SPMD-style parallel execution in order to exploit opportunities for relaxed split-phase synchronization.

The idea of replicating code in certain contexts raises the question of why not replicate the entire program execution and only synchronize and communicate when executing a skeleton, effectively switching to an SPMD execution model. The problem here is that replication is not generally feasible because we must preserve the program's extensional behaviour. For example, an I/O operation must be performed by a single thread only, in order to mimic sequential program behaviour. Shared memory systems typically also share file systems and I/O channels. Hence, concurrent access to such resources requires careful synchronization. Memory allocation in the course of `GenArray` skeletons must be single-threaded, too. Moreover, programs may contain parts, for which thread-safety cannot be guaranteed, e.g. calls to libraries that were designed without multithreaded execution in mind. In many cases, it is not even sufficient to restrict execution to a single thread, but all other threads must remain dormant while the unsafe code is executed.

Our key idea is to classify each atomic piece of code in a program into one of these four categories:

- *MT (multi-threaded):* code that may be executed by multiple threads in a cooperative manner, i.e. our data parallel skeletons.
- *ST (single-threaded):* code that must be executed by a single thread only, e.g. certain memory allocations or I/O operations.
- *ET (exclusive-threaded):* code that must be executed by a single thread only without other threads being active at the same time, e.g. library calls.
- *AT (any-threaded):* code that may be executed either in a replicated manner by multiple threads or by a single thread either in ST or in ET mode, e.g. simple reentrant function calls like `a = sin(b)`.

While skeletons make opportunities for parallel execution explicit, it is generally non-trivial to identify code that must not be replicated. In practice, this requires us to make conservative assumptions and to rely on ET classification if in doubt.

```
ST a_mem = Allocate(...);             ST a_mem = Allocate(...);
MT     a = GenArray( a_mem, ...);     MT     a = GenArray( a_mem, ...);
MT     h = FoldArray( ...);           MT     h = FoldArray( ...);
AT     b = a[0];                 ==>  MT     b = a[0];
MT     c = FoldArray( ...b...);       MT     c = FoldArray( ...b...);
ET     d = StrangeFun1( c);           ET     d = StrangeFun1( c);
AT     e = sin(d);                    ET     e = sin(d);
ET     f = StrangeFun2( c);           ET     f = StrangeFun2( c);
```

Figure 2: Classification of code example.

As our aim is to reduce the number of execution mode changes, we extend MT, ST, and ET classifications as far as possible by absorbing adjacent AT-classified code. Following this idea, we build larger code regions featuring the same execution mode. Fig. 2 illustrates this procedure by a small example. We make the initial allocation instruction of a `GenArray` skeleton explicit here because it must be executed by a single thread only. In the example we have two instructions classified as AT. By expanding MT, ST, and ET classifications as far as possible we turn one into MT and the other into ET. If an entire function is classified as AT, we may build specializations for applications in MT, ST, and ET contexts. If a function contains both ET and MT classified code, it is marked ET.

```
                                       MT  a = GenArray( a_mem, ...);
MT  a = GenArray( a_mem, ...);         MT  a = signal( a);
MT  h = FoldArray( ...);          ==>  MT  h = FoldArray( ...);
MT  b = a[0];                          MT  a = wait( a);
MT  c = FoldArray( ... b ...);         MT  b = a[0];
                                       MT  c = FoldArray( ... b ...);
```

Figure 3: Introduction of explicit split-phase synchronization in MT-cells.

After successful expansion of classifications we build *cells* of code with the same execution mode. In order to realize split-phase synchronization within MT cells, we explicitly introduce `signal` and `wait` operations into the code. This is done in two steps: First, we insert a `signal` operation after each skeleton and a corresponding `wait` operation before the first reference to a data structure computed by a skeleton. In a subsequent optimization step, we remove superfluous `signal`/`wait` pairs, e.g., if some data is concurrently computed before other data and used later, then the inner `signal`/`wait` pair implies the outer. Fig. 3 illustrates the introduction of `signal` and `wait` operations for the example MT-cell of Fig. 2.

4. Rearrangement of Code

Unfortunately, the techniques described in the previous section are often not as effective as desired. The problem is that sequences of assignments constitute a total order while the data dependencies only enforce a partial order. In practice, concrete sequences of assignments in intermediate code often stem from pure coincidence. Maybe, an unaware programmer has written the code without the necessary knowledge about parallelization or simply without parallelization in mind. Maybe, preceding compilation steps have already reorganized the code in some way not taking parallelization sufficiently into account.

Fig. 4 illustrates how a small change in the sequence of assignments may hinder generation of efficient parallel code. In the first example (left) we have moved the `FoldArray` skeleton that computes `h` to the bottom. As a consequence, we end up with two MT-cells instead of one. In the second example (right) we have moved the same `FoldArray` skeleton only one line down. As a consequence, we end up with a single MT-cell, but split-phase synchronization no longer has the desired effect: the `wait` immediately follows the `signal` thus effectively forming a regular synchronization barrier.

```
ST a_mem = Allocate(...);          ST a_mem = Allocate(...);
MT     a = GenArray( a_mem, ...);  MT     a = GenArray( a_mem, ...);
MT     a = signal( a);             MT     a = signal( a);
MT     a = wait( a);               MT     a = wait( a);
MT     b = a[0];                   MT     b = a[0];
MT     c = FoldArray( ...b...);    MT     h = FoldArray( ...);
ET     d = StrangeFun1( c);        MT     c = FoldArray( ...b...);
ET     e = sin(d);                 ET     d = StrangeFun1( c);
ET     f = StrangeFun2( c);        ET     e = sin(d);
MT     h = FoldArray( ...);        ET     f = StrangeFun2( c);
```

Figure 4: Two variants of Fig. 2 that motivate the need for code rearrangement.

In order to successfully apply the proposed techniques, we need a preprocessing code rearrangement phase that decouples MT, ST, and ET classified code as far as data dependencies allow. Within individual MT-cells, code must be arranged in a way that renders split-phase synchronization effective. Our code rearrangement algorithm is divided into four consecutive steps that turn the total ordering of a given sequence of assignments into the partial ordering of a data flow graph and then re-introduce a refined total ordering step-by-step.

4.1. Step 1 — Creating a data flow graph

The aim of this step is to abstract from a concrete execution order of assignments. Creation of a data flow graph critically depends on properties of the programming language. In intermediate SAC code, an exact cycle-free data flow graph can easily be inferred. Control flow instructions like loops and if-statements are represented by equivalent tail-end recursive functions and conditional expressions in a variant of static single assignment form. Side-effects in function applications are made explicit through a variant of uniqueness types [15]. In other languages more conservative assumptions may be necessary. Fig. 5 shows an example data flow graph, which will be used for the illustration of our rearrangement algorithm throughout the remainder of this section.

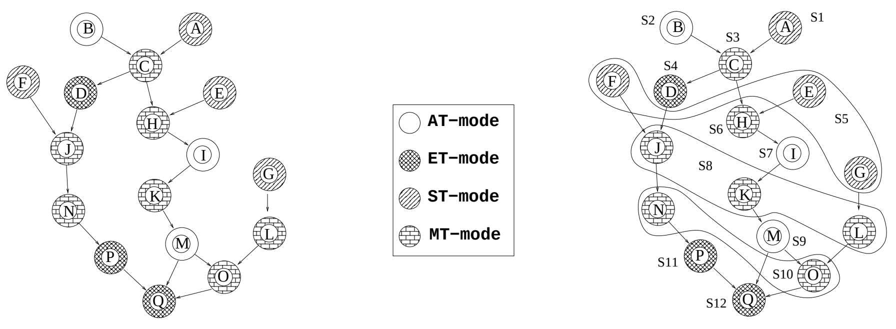

Figure 5: Example data flow graph to illustrate the rearrangement algorithm (left) and result of step 2 (right).

4.2. Step 2 — Reducing execution mode changes

To reduce the number of execution mode changes between ET-, ST-, and MT-mode, we identify clusters of nodes of the same mode and create a list of sets of such nodes. While this step determines the execution order of the sets, ordering within individual sets is postponed.

As mentioned in Section 3, each function containing at least one node in ET-mode is tagged ET. Hence, either the current task is trivial or the execution mode of the function's return statement is ET. Therefore, the algorithm starts putting all ET-coloured sinks of the function's data flow graph G into one set.[†] This set will be the last set of the list returned by step 2. Afterwards, the algorithm takes the subgraph G', created by G's nodes without the nodes already collected, and puts all ET-coloured sinks of G' into a new set. This set will be the new head of the resulting list. This procedure continues until G' includes no more ET-coloured sinks. The ST- or AT-property of a node gives us the opportunity to execute it in ET-mode as well. Therefore, the algorithm continues collecting AT- and ST-coloured nodes into the initial set. Each AT- and ST-collection phase is followed by another ET-collection phase. Collection of nodes into the initial set continues until no more ET-, AT- or ST-coloured sinks are available in G'.

Now, the algorithm switches into collecting MT-coloured nodes. The ST-coloured nodes are the last nodes collected before because nodes in ST-mode can be executed in parallel to nodes in MT-mode. In contrast to the collection of ET-coloured nodes,

[†]In the absence of dead code the function's return statement is the only sink of the graph.

which puts dependent nodes into consecutive sets, dependent MT-coloured nodes are put into sets with sets of AT-coloured and ST-coloured nodes in-between whenever possible. For short, the algorithm puts as few other sets as possible in between ET-sets, but as many other sets as possible in between MT-sets.

This procedure is motivated by reducing the idle time of threads waiting at the corresponding barriers. The reason to handle AT-coloured nodes earlier than ST-coloured nodes is to increase the distance between nodes in ST-mode and dependent nodes in MT-mode. Thus, the master thread is given as much time as possible to compute an ST-set of nodes before the first worker thread reaches a dependent MT-set. Due to the use of the SSA form, a constructed data flow graph does not contain cycles. Therefore, the algorithm always finds a sink in the data flow graph G or any of its subgraphs. Hence, the algorithm always terminates.

The effect of the second step of our rearrangement algorithm on the running example is shown on the right hand side of Fig. 5. We have identified a sequence of 12 sets. While most sets contain a single node only, the more interesting cases are $S_5 = \{E, F, G\}$, $S_8 = \{J, K, L\}$, and $S_{10} = \{N, O\}$. Each set contains data independent nodes with equal execution modes.

4.3. Step 3 — Preorder nodes of the same set

The third step of the rearrangement algorithm leaves the order of the sets S_i untouched, but introduces a partial ordering within the sets. Technically, each set S_i of nodes is transformed into a corresponding list L_i of sets of nodes. Each node n is tagged with an additional attribute, called *minimum backward dependence distance*. This attribute yields the minimum number of sets S_i in between n's set and each of the nodes on which n depends. In the running example of Fig. 5 the minimum backward dependence distance of node K of set S_8 is 1 ($K \in S_8$ depends on $I \in S_7$) while the minimum backward dependence distance of nodes J and L of the same set is 3 ($J \in S_8$ depends on $F \in S_5$, $L \in S_8$ depends on $G \in S_5$).

The nodes of a set S_i are grouped into subsets $l_{i,*} \in L_i$ in ascending order of their associated minimum backward dependence distances. In the running example we get $L_5 = [\{E, F, G\}]$, $L_8 = [\{J, L\}, \{K\}]$, and $L_{10} = [\{N, O\}]$. If two nodes have the same minimum backward dependence distance, they remain in the same subset, and the decision which one to execute first is postponed to the next step.

The rationale for establishing this kind of order is to avoid execution of data dependent nodes immediately one after the other. Instead, we aim at arranging other nodes in between data dependent nodes to increase the potential distance between notification of other threads about completion by one thread and the waiting for completion of a multithreaded computation by all threads. This way, we relax synchronization requirements and reduce thread idle time. Since these considerations do not apply to sets S_i tagged as ET-mode, we simply preserve the original sequence of nodes in these cases.

4.4. Step 4 — Committing the final order

The last step of the rearrangement algorithm determines the final execution order. Only those lists $l_{i,j}$ that contain non-singleton sets require our further attention. We traverse the list of lists of sets of nodes $l_{i,j}$ in reverse order starting with the last element of the last nested sublist. Nodes are collected in an initially empty list $\mathbb{NL}$. If $l_{i,j}$ contains only one element, this node is pushed on top of $\mathbb{NL}$. Otherwise, we choose that element of $l_{i,j}$ that has the maximum *forward dependence distance*, i.e., that element whose first dependency in $\mathbb{NL}$ is closest to the end of the list. The motivation of this procedure is very similar to that of Step 3: we aim at separating the computation of data as far as possible from where this data is needed in order to relax synchronization requirements. If two nodes have the same maximum forward dependence distance, we re-establish the original order.

```
Cell 1 ST: A
Cell 2 MT: B -> C
Cell 3 ET: D
Cell 4 ST: E -> F -> G
Cell 5 MT: H -> I -> J -> L -> K -> M -> N -> O
Cell 6 ET: P -> Q
```

Figure 6: Effect of code rearrangement on the running example.

Fig. 6 shows the effect of code reordering on the running example. We successfully avoid the inapt sequences $[\ldots, L, J, \ldots, N, O \ldots]$ and $[\ldots, J, L, \ldots, O, N \ldots]$, where data dependent skeletons become adjacent in final code. In contrast, the sequence $[\ldots, J, L, M, N, \ldots]$ increases the synchronization distance by inserting data parallel node L in between nodes J and M.

5. Experimental Evaluation

To quantify the impact of our compilation techniques we have built a prototype implementation, which we used for experiments on three different machine architectures: a SUN Enterprise 4000 with 6 UltraSparc II processors, a Linux server with two hyper-threaded Intel Xeon processors, and a SUN SF15k with 72 UltraSparc III processors. These experiments use the benchmark kernel fragment shown in Fig. 7. It consists of 6 `GenArray` skeletons, each computing a vector of `N` elements in a fairly simple manner. Applications of the function `fun` basically serve two purposes. Firstly, they lead to a more realistic argument set size without introducing additional data dependences. Secondly, they allow us to systematically create workload imbalances among threads without making the example overly complicated. Apart from reflecting the irregularity of more complex real world applications, this feature allows us to mimic highly loaded multiprocessor systems. Our benchmark serves as test bench to quantify the potential performance impact of our proposed measures in a rather simple sand box environment.

```
A = GenArray( [N], idx -> 1                 + fun(U,V,W,X,Y,Z));
B = GenArray( [N], idx -> 2 * B[idx]        + fun(U,V,W,X,Y,Z));
C = GenArray( [N], idx -> 2                 + fun(U,V,W,X,Y,Z));
D = GenArray( [N], idx -> A[idx] + C[idx]   + fun(U,V,W,X,Y,Z));
E = GenArray( [N], idx -> 3                 + fun(U,V,W,X,Y,Z));
F = GenArray( [N], idx -> E[idx] + B[idx]   + fun(U,V,W,X,Y,Z));
```

Figure 7: Code fragment used in experimental evaluation.

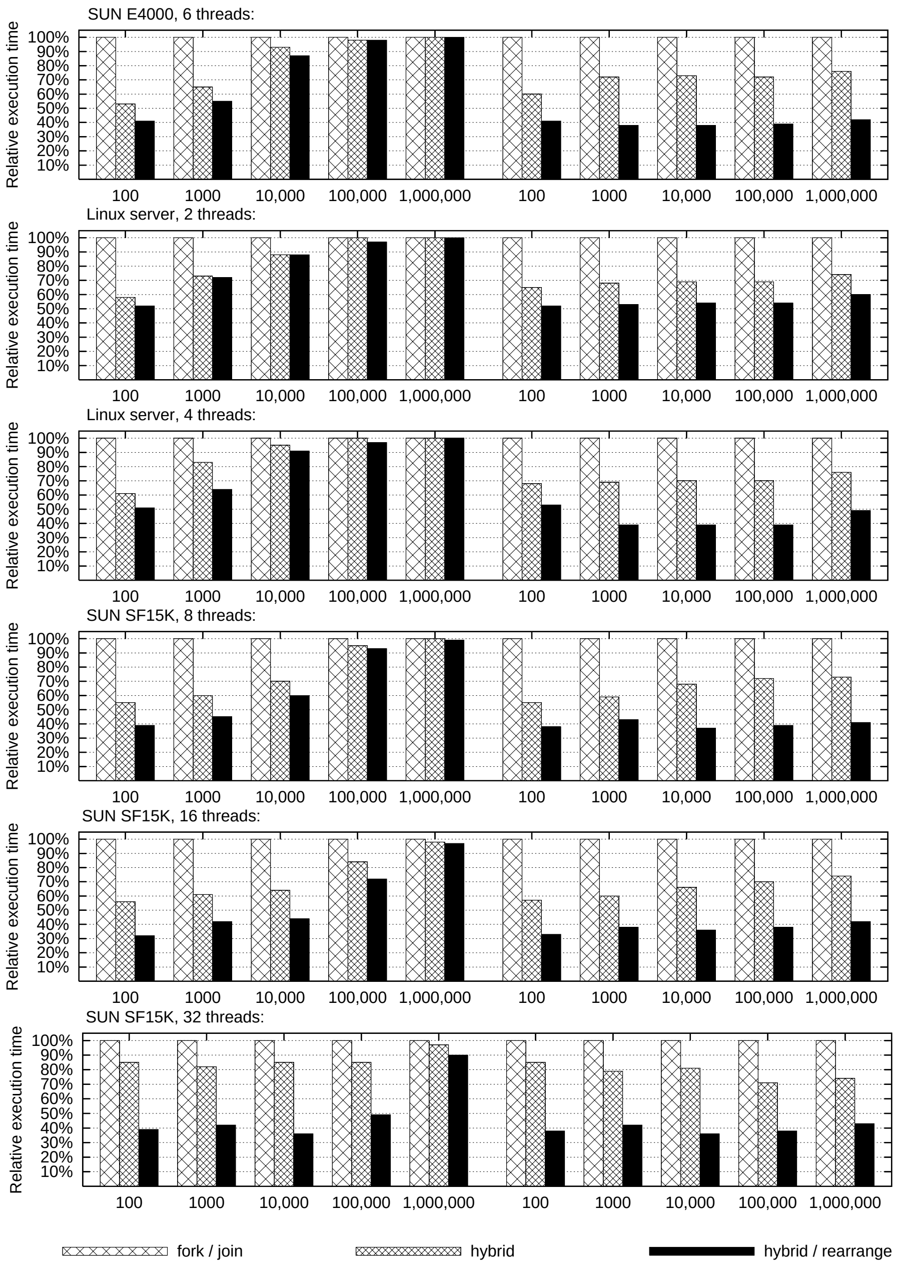

Figure 8: Results of experimental evaluation for various architectures (top to bottom) with (right) and without (left) workload imbalances. The x-axis denotes the length of the vectors involved, i.e., smaller vector size means more frequent synchronization and communication events.

We have investigated three parallelizations of this benchmark kernel: the first one follows a fork/join execution model, the second one our new hybrid model, and the third one additionally applies code rearrangement. The fork/join approach treats each skeleton in isolation, i.e., we end up with six communication operations to broadcast parameters to worker threads and with six synchronization barriers. The hybrid model described in Section 3 leads to a single large MT cell comprising all six individual skeletons. However, data dependences enforce immediate synchronization after computing `A`, `C`, and `E`, respectively. If we additionally apply code rearrangement, as introduced in Section 4, we achieve a maximum distance of 2 between `signal` and `wait` of our split-phase barriers, i.e., we compute two more vector operations before waiting for completion of the first one.

We have intentionally left out I/O operations or other scalar code in Fig. 7 to keep the example short. Taking the code literally, our hybrid approach would coincide with the SPMD model, code rearrangement apart. However, assuming I/O operations scattered between the skeletons, the SPMD model would coincide with the fork/join model. Therefore, we dispense with specific treatment of the SPMD execution model in the following.

Fig. 8 shows the results of our experiments. We normalize the execution time of the fork/join-based variant to 100%, and show the relative performance increase when switching to the hybrid model without and with code rearrangement. Using different vector sizes `N` allows us to systematically vary the frequency of synchronization/communication events. Furthermore, we have made each experiment without (left hand side) and with (right hand side) introducing workload imbalances.

The results essentially back the initial assumptions that motivated our work. With frequent synchronization substantial performance gains can be realized. They gradually vanish as we reduce the synchronization frequency by increasing the vector size. The gains are generally lower on the Linux server than on the SUN servers. Synchronization and communication overhead typically grows with system size. Likewise, larger thread counts benefit more from our optimizations than smaller ones. These observations can be made in particular on the large SF15k system.

The right hand side of Fig. 8 shows data gained by repeating the entire experiment with workload imbalance introduced by substantially delaying individual threads. The positive effect of our hybrid execution model on runtime performance is even greater than before. The decoupling of threads due to the removal of synchronization barriers gains in importance with load imbalance. In contrast to the first experiment this effect is rather independent of the vector size and, hence, of the frequency of (enforced) synchronization. This is owing to the amount of imbalance we introduced. Our aim was to demonstrate how the impact of the hybrid execution model may grow with increasing load imbalance, which is typical for both complex applications and for large, heavily loaded multi-processor environments.

6. Related Work

Reduction of synchronization and communication overhead has long been considered as critical for the success of parallel execution. Some approaches are specific to parallelization, while others are beneficial in sequential execution as well. Examples of the latter case are deforestation [22] and tupling [4]. They avoid multiple traversals of the same data structure by combining the necessary computations in a single sweep. Deforestation addresses computational pipelines that transform an aggregate data structure in a sequence of steps, whereas tupling aims at computing several results from the same aggregate data structure at the same time. Assuming data parallel execution deforestation and tupling lead to a reduction of synchronization barriers in compiled code. However, their applicability is restricted by properties of the operations involved that are independent of parallelization.

Algorithmic skeletons are a popular approach to high-level parallel programming [6,7,18,1]. In this context, frameworks of meaning-preserving transformation rules have been developed, that aim at replacing multiple related instances of skeletons by a presumably more efficient combination [9,10,19]. More specific to high-level array processing is the psi-calculus [20] that offers transformation rules on Apl-like array operations. Similar optimizations have been proposed on the level of collective operations in Mpi [11]. All these approaches have in common that code transformation always remains within a pre-defined set of operations. Our own work described in [13] also falls into this category. In contrast, the problems we address in our current work stem, among others, from optimizing skeleton-like code with arbitrary non-skeletal code.

The data-flow language Sisal [2] circumvents the entire problem by restricting input/output to reading input data at program startup and writing output data upon program termination. Hence, parallel execution of Sisal programs may follow an SPMD model without penetrating the sequential extensional behaviour. OpenMP [8] supports the *explicit* specification of hybrid program execution, but the programmer alone is responsible for correct interaction with the execution environment. The idea of split-phase synchronization was, to our best knowledge, first proposed under the term *fuzzy barriers* in [17].

7. Conclusion and Future Work

Compilation of a data parallel language with I/O for multithreaded execution inevitably leads to alternating single-threaded and multi-threaded supersteps, regardless of whether we adopt a fork/join or an SPMD execution model. In the fork/join model data parallel operations remain pockets of parallel activity in an otherwise sequentially executed program as a matter of principle. However, in the presence of I/O or other non-replicatable operations even the SPMD approach cannot avoid pockets of single-threaded execution in an otherwise parallel program. Effectively, the two approaches mainly differ in the default case for code sections that may or may not be replicated.

Neither replicating all replicatable code (SPMD) nor dispensing with replication at all (fork/join) is likely to yield best parallel performance. Therefore, we have proposed a hybrid execution model that decides about replication of instructions based on the individual context. We have outlined a compilation framework that aims at reducing communication and synchronization requirements. Large-scale code-restructuring techniques improve the effectiveness of our approach in practice.

Experiments on three different shared memory architectures using a prototype implementation demonstrate the effectiveness of our approach in principle. Therefore, we aim at incorporating these techniques into the production compiler of our functional array language SaC to further investigate the impact of our techniques on a larger range of applications in the future.

References

[1] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and Transformations in an Integrated Parallel Programming Environment. In *Parallel Computing Technologies (PaCT'99)*, LNCS 1662. Springer, 1999.

[2] D.C. Cann. Retire Fortran? A Debate Rekindled. *CACM*, 35(8), 1992.

[3] B.L. Chamberlain, S.-E. Choi et al. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Software Engineering*, 26(3), 2000.

[4] W.N. Chin. Towards an Automated Tupling Strategy. In *Proc. PEPM'93*. ACM Press, 1993.

[5] W.P. Cockshott. Vector Pascal reference manual. *SIGPLAN Notices*, 37(6), 2002.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, London, UK, 1989.

[7] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3), 2004.

[8] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[9] S. Gorlatch and C. Lengauer. (De)Composition Rules for Parallel Scan and Reduction. In *Proc. MPPM'97*. IEEE Computer Society Press, 1997.

[10] S. Gorlatch and S. Pelagatti. A Transformational Framework for Skeletal Programs: Overview and Case Study. In *Parallel and Distributed Processing*, LNCS 1586. Springer, 1999.

[11] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization Rules for Programming with Collective Operations. In *Proc. IPPS/SPDP'99*, 1999.

[12] C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proc. IPDPS'02*. IEEE Computer Society Press, 2002.

[13] C. Grelck. Optimizations on Array Skeletons in a Shared Memory Environment. In *Implementation of Functional Languages (IFL'01)*, LNCS 2312. Springer, 2002.

[14] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *J. Functional Programming*, 15(3), 2005.

[15] C. Grelck and S.-B. Scholz. Classes and Objects as Basis for I/O in SAC. In *Proc. IFL'95*.

[16] C. Grelck and S.-B. Scholz. Towards an Efficient Functional Implementation of the NAS Benchmark FT. In *Parallel Computing Technologies (PaCT'03)*, LNCS 2763. Springer, 2003.

[17] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. ASPLOS-III*. ACM Press, 1989.

[18] H. Kuchen. A Skeleton Library. In *Parallel Processing (Euro-Par'02)*, LNCS 2400. Springer, 2002.

[19] H. Kuchen. Optimizing Sequences of Skeleton Calls. In *Domain-Specific Program Generation*, LNCS 3016. Springer, 2004.

[20] L.M. Restifo Mullin and M. Jenkins. Effective Data Parallel Computation using the Psi Calculus. *Concurrency Journal*, 1996.

[21] S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *J. Functional Programming*, 13(6), 2003.

[22] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2), 1990.