JaMP: An Implementation of OpenMP for a Java DSM

Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen

Abstract

Although OpenMP is a widely agreed-upon standard for the C/C++ and Fortran programming languages for semi-automatic parallelization of programs for shared memory machines, not much has been done on the binding of OpenMP to Java that targets clusters with distributed memory. This paper presents three major contributions. (1) JaMP is an adaptation of the OpenMP standard to Java that implements a large subset of the OpenMP specification with an expressiveness comparable to that of OpenMP. (2) We suggest a set of extensions that allow a better integration of OpenMP into the Java language. (3) We present our prototype implementation of JaMP in the research compiler Jackal, a software-based distributed shared memory implementation for Java. We evaluate the performance of JaMP with a set of micro-benchmarks and with OpenMP versions of the parallel Java Grande Forum (JGF) benchmarks. The micro-benchmarks show that OpenMP for Java can be implemented without much overhead. The JGF benchmarks achieve a good speed-up of 5–8 on 8 nodes.

1. Introduction

The two common platforms used in today’s High Performance Computing (HPC) are large-scale Symmetric Multi-Processors (SMP) with shared memory and computational clusters. A cluster’s distributed and non-uniform memory hierarchy is still visible in runtime performance and forces the programmer to use message-oriented programming models such as MPI [17] or PVM [12] to explicitly transfer data between the different nodes of a cluster.

Software-based Distributed Shared Memory (S-DSM) systems strive to hide the complexity of message passing by adding a middleware layer that accesses remote objects and maintains memory consistency. Instead of worrying about placing send and receive operations at the right locations of the code, the S-DSM programmer can focus on developing an efficient solution for the scientific problem. Of course, only a truly efficient S-DSM implementation that adheres to known concepts will be accepted by the HPC community. OpenMP [20] is such a widely agreed-upon standard for writing HPC programs on SMPs. An OpenMP programmer annotates a
sequential program with special directives that are used by the OpenMP compiler to generate a parallel version of the program.

This paper demonstrates that the expressiveness of OpenMP can be brought to Java and can be efficiently implemented in a Java-based S-DSM system on clusters. The paper is organized as follows. Section 2 surveys related work in the area of both OpenMP and DSMs. Section 3 briefly introduces the Jackal S-DSM, i.e. the middleware layer on which JaMP is built. Section 4 shows how the OpenMP directives are adapted to and should be extended for the Java programming language. The implementation of JaMP in Jackal is presented in Section 5. Section 6 evaluates the performance of our JaMP implementation.

2. Related Work

With POSIX threads [6, 9] or with the Java threading API [13], two popular APIs for parallelizing programs on SMPs, the programmer writes an explicitly parallel program by creating, synchronizing, and joining worker threads. With OpenMP [20], the compiler semi-automatically parallelizes a sequential program enriched with OpenMP directives that inform the compiler on how to transform the program into a parallel version by means of threads.

Many commercial compilers for modern hardware architectures can compile OpenMP programs. There are also various open-source implementations of the OpenMP standard for SMPs. OdinMP/CCp [1], Omni OpenMP [16], and OpenUH [14] are source-to-source compilers that preprocess source code with OpenMP directives and emit a source program that uses a threading library (OdinMP uses pthreads; Omni OpenMP can use different thread packages; OpenUH can also compile to native Itanium code.) The upcoming GCC version 4.2 is expected to also compile OpenMP (C/C++ and Fortran) code to native.

We are not aware of any OpenMP specification for Java. The JOMP [3] source-to-source compiler transforms a subset of the OpenMP standard to regular Java and uses the Java Threading API for parallelism. In contrast to JOMP, our JaMP compiler benefits from translating rather than rewriting the OpenMP directives, because the Jackal compiler is aware of the parallelization applied. This enables various compiler optimizations, e.g., data race analysis, use of explicit send/receive operations instead of the DSM protocol, and the like. Such optimizations are difficult for source-to-source translators, since the parallelization is hidden from the final compiler stages by calls to the threading API.

There is little OpenMP-related work on clusters. Intel Cluster OMP [8] extends the OpenMP specification by a special clause to share data between different cluster nodes. It is based on an extended version of the TreadMarks DSM [10]. Omni/SCASH [19] transparently executes OpenMP-enriched programs in the SCASH-DSM [7]. However, Java is not an issue in any of these projects.

3. The Jackal DSM System

Jackal [23] is an object-based DSM system that automatically distributes a thread-parallel Java program onto a cluster. Object-based DSMs distribute data on the level of objects objects...
```java
int foo(SomeObject o) {
    return o.field;
}

int foo(SomeObject o) {
    if (!readable(o))
        fetch(o, readable);
    return o.field;
}
```

Figure 1. Example function `foo()` before and after the insertion of access checks by the Jackal compiler.

(rather than operating-system pages or cache lines). Since in Jackal multiple threads may share a process, Jackal not only efficiently exploits SMP nodes in a cluster, but in addition it can overlap communication of one thread with the computation of another.

Conceptually, Jackal’s DSM prefixes each object access with a so-called access check that tests whether or not the object is already cached locally. In case it is locally available no further action is required. Otherwise a request is sent to the machine holding the object’s master copy (the home-node) to send a copy of the object. Fig. 1 shows a simple read-only object access (left) and the compiler-inserted access check (right).

At runtime, the address space of each process of a Jackal application is partitioned into three storage areas. Regular Java objects live in a garbage-collected heap. An object cache temporarily stores copies of objects that have been transferred from their home-node by failed access checks. Finally, the runtime system stores information about the DSM state, locking, etc. in an administrative heap. To implement the Java Memory Model [15] a flush list per thread contains the set of objects that have to be flushed to their respective home-nodes whenever a synchronization point is reached. A read/write bitmap indicates the accessibility of objects that have been transferred to the local node. Finally, some hashtables map global object references to memory addresses on the local node.

Jackal uniquely identifies each object by a so-called Global Object Reference (GOR), i.e. a tuple `(allocation-node-rank, allocation-address)`. The `allocation-node-rank` is the rank of the object’s home node; the `allocation-address` is the object’s address on that node. The GOR is fixed during the object’s life-time and may only be reused if the object is reclaimed by the garbage collector. When a failed access check requests a non-local object, the object’s data is mapped into the object cache and can efficiently be accessed by a local pointer. The DSM runtime maintains a mapping between local pointers and their respective GORs (for status or data updates on the home node).

For performance reasons, Jackal does not use object granularity when transferring arrays. A Java array might consume a large amount of memory and instead of the whole array often only a fraction of it will be used. Hence, the DSM runtime partitions an array into segments of 64KB or 256KB (depending on the DSM configuration). An access check requests and transfers only one segment from the array’s home node.

Jackal applies standard compile-time optimizations and implements DSM functionality efficiently [24, 23]. First, it removes redundant access checks by aggressively applying static optimizations. For example, no access check is needed if the compiler can prove that an object...
is still locally available from an earlier access check. Second, objects are replicated if no thread writes to them. Hence, flushing and state invalidation can be avoided. Third, for multiple writers that concurrently modify a particular object, Jackal avoids false-sharing by constructing a differential image that is sent to the home-node in a flush operation. The three optimizations result in fewer messages being transferred.

4. JaMP Directives

Since JaMP directives closely follow the OpenMP standard, the JaMP programming model is as expressive as the OpenMP programming model. An OpenMP programmer can use JaMP without learning a new syntax for directives.

This section introduces the JaMP directives and discusses the few issues that made syntactical changes necessary to implement an OpenMP dialect for Java. For most directives, no modification of the semantics is necessary. We also present a set of new features that are not part of the OpenMP specification, but that extend the OpenMP approach to better fit to the Java philosophy.

4.1. Pragmas

Since they are missing in the Java specification, JaMP provides its own implementation of pragmas (inspired by the Fortran free-source form directives [20]). As the code examples (Figures 2 to 8) show, it is straightforward to turn OpenMP pragmas into corresponding JaMP pragmas.

Moreover, Java does not provide a preprocessor. In OpenMP there are special macros that allow the inclusion of code in case of an OpenMP-enabled compilation. Hence, JaMP also has to provide a means for conditional compilation. The JaMP compiler treats special line-comments as conditional compilation guards. A Java statement beginning with //# is only considered when it is compiled with a JaMP-enabled compiler. Other Java compilers safely ignore such statements.

4.2. Parallel regions and data sharing

The parallel directive marks a section of a program as parallel. When a thread reaches a parallel region, it conceptually creates a team of threads that execute the region’s code in parallel. At the end of each parallel region, there is an implicit barrier. Only after all threads that execute the region reach the barrier, the master thread continues.

JaMP supports all types of data-access restrictions defined by OpenMP. See Fig. 2 for an example. For variables marked as shared, the same memory location in the DSM is used by all threads that are put to work on the parallel region. For private variables, every thread receives a private copy of the variable that is initialized with a default value. To initialize a private variable with the value it had before the parallel region, the firstprivate clause can
int a = 1; int b = 2; int c = 3;
//#jamp parallel private(a) firstprivate(b) shared(c)
System.out.println("a= " + a + ", b= " + b + ", c= " + c);

Figure 2. Example of a parallel region with a private, a firstprivate, and a shared variable.

package jamp:
interface CustomCloner<T> {
    T clone(T other);
}

package jamp:
interface Reducer<T> {
    T reduce(T a, T b);
    T init();
}

(a) (b)

Figure 3. Interfaces for custom cloning and reductions.

public class IntegerCloner extends jamp.CustomCloner<Integer> {
    public Integer clone(Integer other) {
        return new Integer(other.intValue());
    }
}

IntegerCloner cloner = new IntegerCloner();
Integer i = new Integer(1);
Integer j = new Integer(2);
//#jamp parallel firstprivate(cloner:i,j)
{ ...

Figure 4. Example of custom object cloning.

be used. The this reference that is implicitly defined in instance methods of Java classes is always passed to parallel regions as a shared variable.

In the C++ binding, the semantics of firstprivate variables depend on the data type. If a firstprivate variable x is a pointer to an object, C++ privatizes it. However, if x is of an object type, the copy constructor of the class is invoked. For Java, this approach is not feasible because of two reasons. First, Java can only pass object references, i.e. there is no way to pass (a copy of) an object. Hence, only the reference to an object can be privatized and not the object.
//##jump parallel
{
    //##jump for
    for (<init>; <cond>; <inc>) {
        // some code
    }
}

Figure 5. Syntax example of the for directive.

itself. Second, there are no copy constructors in Java. Although Java supports object cloning by means of the Cloneable interface, this interface is not implemented by all classes of the Java API and may not be implemented by all classes of the application. JaMP therefore needs a mechanism for object privatization: Objects that do not implement the interface Cloneable can be cloned by specifying a custom cloner in the firstprivate directive (see Fig. 3(a)). Fig. 4 shows how two Integer objects (i and j) may be privatized by means of a customized cloner. The clone() method provided by the programmer is invoked by the runtime system.

4.3. Work-sharing directives

The iteration space of a loop can be distributed among a set of worker threads by means of the for directive (see Fig. 5). In the subsequent for statement, init is the initialization expression of the loop, cond is a loop-invariant termination condition, and the inc increment specifies how to increment the loop variable by some loop-invariant value. According to the OpenMP standard, the loop variable is privatized.

The schedule clause (not shown in the example) that can contain the values static, dynamic, and guided controls the load distribution. A static distribution divides the iteration space of the loop into blocks of equal size that are assigned to the worker threads. A dynamic distribution partitions the iteration space into chunks of a certain size. The default size is 1. Each thread then continuously requests a new block as long as there are unprocessed chunks left. In guided loops a thread also requests an unprocessed block. But the block size is proportional to the number of unassigned iterations divided by the number of threads, decreasing to the specified (or default) chunk size [20].

4.4. Reductions

JaMP supports all types of arithmetic reduction operations defined by the OpenMP standard. Fig. 6 shows a parallelized summation of an array into the variable sum. OpenMP 2.5 defines reductions for primitive types (e.g. int, double) and for built-in operators such as addition and multiplication only. Since this restriction is too severe for a Java programmer who is used to polymorphism and overloading, JaMP offers reductions of arbitrary object types with arbitrary
int sumOfArray(int[] array) {
    int sum = 0;
    //#jamp parallel
    {
        //#jamp for reduction(+:sum)
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
    }
    return sum;
}

class BigIntegerAddReducer implements jamp.Reducer<BigInteger> {
    public BigInteger reduce(BigInteger a, BigInteger b) {
        return a.add(b);
    }
    public BigInteger init() {
        return BigInteger.ZERO;
    }
}

BigInteger value = ...; BigInteger[] array = ...;
BigIntegerAddReducer reducer = new BigIntegerAddReducer();
//#jamp parallel for reduction(reducer: value)
for (int i = 0; i < array.length; i++) {
    value = value.add(array[i]);
}

Figure 6. Example of a parallel reduction of all array elements.

class BigIntegerAddReducer implements jamp.Reducer<BigInteger> {
    public BigInteger reduce(BigInteger a, BigInteger b) {
        return a.add(b);
    }
    public BigInteger init() {
        return BigInteger.ZERO;
    }
}

BigInteger value = ...; BigInteger[] array = ...;
BigIntegerAddReducer reducer = new BigIntegerAddReducer();
//#jamp parallel for reduction(reducer: value)
for (int i = 0; i < array.length; i++) {
    value = value.add(array[i]);
}

Figure 7. Example of a custom object-oriented sum reduction.

operations. By implementing the Reducer interface (Fig. 3(b)) customized reduction operators can be implemented. The reduce() method is called whenever the reduction operator is applied to two objects. If the neutral element of the reduction operator is needed, the JaMP runtime system calls the init() method. Fig. 7 shows a custom reduction for objects of type BigInteger. Note that the parallel and the for directives are combined into a single line in Fig. 7, i.e. the usual syntactic sugar is available in JaMP as well.
4.5. Other OpenMP directives

JaMP also supports parallel sections. Whereas all threads execute the same part of a parallel region (parallel and for constructs), they can execute different code concurrently with the sections directive. With the single and master directives it is also possible to have code that is executed by only one thread. Only the single directive has additional clauses (such as data-access clauses) and an implicit barrier at the end of the construct.

User-defined barriers can be created by means of the barrier directive to create program locations at which all threads wait for each other. The critical directive can be used to mark critical sections that may be executed by only one thread at a time.

4.6. Iterators and exceptions

In addition to the extended data-sharing and reduction clauses, JaMP has two major enhancements to better fit into the Java philosophy: parallel iterators and exception handling.

JaMP provides a means to parallelize for-each loops. A for-each loop iterates either over an array or over an arbitrary data structure that implements the Iterable interface. As can be seen in Fig. 8, such loops can be parallelized by adding the iterator or parallel iterator directive. Conceptually, the master thread iterates over the collection and flattens it into a regular array that is then processed in parallel.

So far OpenMP only supports work-sharing of data that is stored in (multi-dimensional) arrays. Arbitrary data structures such as object graphs cannot be partitioned automatically. A future extension of JaMP might turn the iterator construct into a more general construct (such as the taskq directive of the Intel Compiler Suite).

Exception handling is not part of the current OpenMP specification. But as exceptions are an integral Java concept (they are potentially thrown by many byte-codes and are part of many APIs), JaMP can deal with exceptions. The current implementation of JaMP uses an approach that is also taken by JCilk [5] to abort parallel computations in case of an exception. If one of the worker threads throws an exception in a parallel region, the exception is caught and a cancellation flag is set at the thread team. Each of the other threads frequently check...
this flag at so-called cancellation points (e.g. barriers, critical sections). If an exception has been flagged, the threads exit the parallel region. Finally, the exception is re-thrown by the master thread. If more than one exception occurred, the runtime system chooses one of them arbitrarily.

4.7. Current limitations of JaMP

JaMP does not yet fully implement the whole OpenMP 2.5 specification. The limitations listed in this section, however, do not restrict its overall expressiveness since the missing directives can be expressed with little effort by other directives or with a bit of Java code.

JaMP currently does not provide any means to declare orphaned regions. An orphaned region is a region (e.g. a for region) that is not enclosed by a lexically surrounding parallel directive. In such cases an OpenMP-compliant implementation is forced to dynamically determine if the region’s code was invoked from inside a parallel region. If there is none, the orphaned region is executed sequentially. Nested parallelism, i.e. creating parallel JaMP regions inside other parallel regions, is also not yet supported.

We have limited the functionality of the JaMP compiler on some issues to keep the complexity of our prototype low. Currently, instance variables or class variables cannot be privatized. In particular, threadprivate and the copyin and copyout clauses are not supported. Also, the ordered directive and clause as well as the if clause on the parallel directive are still missing.

5. Implementation Details

The implementation of JaMP in the Jackal framework is divided into three parts (see Fig. 9): (1) an extension to the Jackal compiler, (2) a set of JaMP library classes written in Java, and (3) a dispatcher added to the runtime system (written in C for efficiency reasons). For this paper, we focus on the transformation that the JaMP compiler applies to the code. The implementation of the runtime system and the dispatcher will not be covered in detail.

The JaMP compiler front-end identifies the parallel region and compiles it to Jackal’s intermediate code, called LASM. We extended LASM with special instructions that match the JaMP directives. This way, the compiler back-end becomes aware of the parallel execution of the program and can analyze and optimize the code specifically for parallelism. A similar approach is also used by the Intel C/C++ compiler [22].

Fig. 10 shows the (unoptimized) high-level LASM code of the reduction in Fig. 6. For simplicity, we omitted some instructions, e.g. boundary checks and access checks. Lines 2–5 determine the number of threads that are needed to execute the parallel region. Line 7 represents the work-sharing over the for loop. Lines 11–21 contain the body of the for loop. The loop uses a private loop counter i, declared in line 8. The original for loop has been replaced by a do-while loop guarded by an if statement.

Several optimizations are applied to the code at this level. For example, the back-end performs a hierarchical data-flow analysis (DFA) that respects the hierarchical nature of the JaMP intermediate instructions. Other standard optimizations include common sub-expression
Figure 9. Architecture of the JaMP implementation.

```
1 local('i', X1+0+0, src=sum:int) = 0
2 jump-parallel-block <num_threads {block(%i24793) {
3  %i24793 = call jackal_jamp_JampRuntime.getTeamSize()
4 }},
5 parallel {
6  jump-for-block <reduction [local('i', src=sum:int), +] { 
7    declare local('i', src=i:int)
8    local('i', src=i:int) = $0L
9    %g24798 = param('g', src=array:int[])
10   if ((local('i', src=i:int) < (%g24798,i).length)) {
11      do {
12         %g24839 = param('g', src=array:int[])
13         %i24840 = local('i', src=i:int)
14         %g24850 = &(%g24839,i).field.data[%i24840]
15         local('i', src=sum:int) = 
16          (local('i', src=sum:int) + (%g24839).field.data[%i24840])
17         %i24921 = local('i', src=i:int)
18         local('i', src=i:int) = (%i24921 + $1L)
19      } while (local('i', src=i:int) < (%g24798,i).length));
20    }
21  }
22 %i24943 = local('i', src=sum:int)
23 return %i24943
```

Figure 10. High-level LASM representation of the example in Fig. 6.
elimination, bounds check removal, and access check removal. In addition, there are JaMP-specific optimizations. For example, a shared variable is only passed to the worker threads when it is live in the parallel region.

Before the code is transformed to a low-level intermediate code, the back-end moves the statements that are enclosed by the parallel region instruction into a newly created function. The technical details of how to extract arbitrary code sequences into spliced-out procedures are described in [11]. Parallelized for loops are then rewritten such that they obey the work-sharing rules of OpenMP. Fig. 11 shows (in Java-like pseudo-code) the result of applying the JaMP transformation to the high-level LASM code of Fig. 10.

The address of the newly created function (lines 12–33 in Fig. 11) is registered in a table to establish a mapping between the address and the function’s globally unique name. Additionally, the compiler inserts code that starts up and terminates the parallel execution of the region.

Line 3 of Fig. 11 constructs and initializes the thread team. Internally, the dispatcher locates the address of the spliced-out function by searching for the function’s name in a look-up table. The execution of the parallel region starts when the stub invokes the function pointer by a call to the dispatcher (line 7). The implicit barrier at the end of a parallel region is hidden in the startAndJoin call. Finally, line 8 performs the reduction of the partial sums.

All the data access is handled by special parameter objects. When a parallel region begins execution, a single Java object of the type JampParam is created to store shared data. Another private JampParam object is allocated per worker thread to keep the thread’s private data. For each primitive type (int, double, etc.), the JampParam class contains an array of the particular type. For every variable, the JaMP compiler generates an offset pointing into the respective array. As an optimization, the read-only implicit variable this is passed directly to the parallel region. For example, for the code shown in Fig. 2, the compiler would assign offsets 0 and 1 of a thread-private JampParam object to the private variables a and b. The shared variable c would be stored at offset 0 of the shared JampParam object. The transformation passes both JampParam objects to the spliced-out function (line 12 of Fig. 11), in which each variable access is replaced by an access to the respective JampParam object and offset.

The transformation of JaMP for regions must take the specified work-sharing models (static, dynamic, and guided) into account. This is handled by a special Java object (LoopInformation) that contains the original loop boundary, the termination condition, the increment, as well as the list of blocks (in case of dynamic or guided scheduling). Although the use of a LoopInformation object makes the implementation less efficient, it is kept flexible for future enhancements. As can be seen in Fig. 11, the loop is rewritten such that every thread handles only a part of the iteration space. For that purpose the code of the original for loop is rewritten into two loops that are executed by each thread. The outermost loop continuously requests a new block (line 21–23) and computes the block’s boundaries (lines 24 and 25). The while loop in lines 26–29 holds the body of the original loop. Only its termination condition is transformed to stop when the loop reaches the end of the current block.

The startAndJoin call in line 7 deals with exceptions as explained in section 4.6. The call to the dispatcher is enclosed by a catch block that handles all exceptions from within the parallel region.
6. Performance Evaluation

We have evaluated the performance of the JaMP implementation with a set of microbenchmarks and a subset of the Java Grande Forum benchmarks. The benchmarks were run on a commodity cluster of dual Intel Xeon machines (2.66 GHz) with 2 GB of main memory per machine. The nodes are connected by Gigabit Ethernet and run Debian Linux (kernel version 2.6.15.7-smp). The average ping-pong latency for a Jackal-internal RPC in this cluster is roughly 140 µsec. For the measurements, we only use one CPU of each machine. For each
benchmark, we compute the runtime as the average of five independent runs. The baseline for the speed-up measurements is the runtime of the benchmarks with only one CPU.

6.1. Micro-benchmarks

To determine the speed of the basic JaMP operations, we use the same set of micro-benchmarks that has been used to assess the JOMP implementation [4]. As suggested in [2], the micro-benchmarks compute the overhead of a particular directive by measuring the runtime of the execution of an empty loop and the runtime of the same loop with the directive added. Fig. 12 shows the execution times of the individual JaMP directives takes.

The rather high overhead of the barrier statement (see Fig. 12(a), diamond line) is due to Jackal’s quite naive linear barrier implementation, for which the master node maintains the barrier’s counter. Whenever a node reaches the barrier, it sends an RPC to the master node and waits until a reply is received. The master node sends reply messages for all outstanding RPCs only after all threads have reached the barrier. Since for large numbers of nodes this kind of barrier algorithm will become a bottleneck, a hierarchical implementation with better scalability will be added in the future.

The 5 msec needed for a barrier with 8 nodes consists of the time required to send 2 RPCs per node. The master node can only process the incoming RPCs sequentially, because it must lock the barrier’s data. Hence, roughly 4 of the 5 msec are contributed by message latencies and by RPC handlers. About 1.5 msec are spent in the DSM protocol, that processes the flush lists of the thread. The single directive takes roughly the same time, as it is currently implemented as a check of the thread ID plus a barrier at the end of the construct (which is required by the OpenMP specification).
Figure 13. Overhead of various scheduling schemes and different node counts. The first three bars per node show the time needed by a static scheduling of chunk sizes 32, 64, and 128. The next group of three bars depicts dynamic scheduling.

The critical directive (square line) is faster, because fewer RPCs are sent to the master node. Whenever a thread encounters a critical region, it sends an entry request to the master node. If the region is currently not owned by a thread, the master node immediately replies. Otherwise, the grant message is deferred until the current owner leaves the critical region.

The overhead to start and finish the execution of a parallel region is about 20 msec (see Fig. 12(b), triangle line, 8 nodes). The overhead consists of (1) creation and initialization of the shared and private JampParam objects and the thread team object, (2) a sequence of RPCs that create the worker threads on the remote machines, and (3) the final barrier (roughly 5 msec). Creating a remote thread in the Jackal environment is expensive because of a large amount of DSM protocol overhead. The overhead includes allocating and setting up read/write bitmaps of the object cache, allocation and initialization of the GOR hashables, and other DSM specific actions.

The overhead of a for directive (circle line), mainly consists of two barriers that explain 10 of 13 msecs. At first, a barrier is required by the JaMP implementation to wait for the initialization of the LoopInformation object. The second barrier at the end of the for region is required by the OpenMP standard. The remaining roughly 3 msec are needed to initialize and distribute the LoopInformation object by the DSM protocol. The overhead of the compound
Table I. Data sizes used for the JGF benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size A</th>
<th>Size B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>400,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Crypt</td>
<td>80,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>Sparse</td>
<td>400,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>SOR</td>
<td>4,000</td>
<td>6,000</td>
</tr>
<tr>
<td>LU Fact</td>
<td>2,000</td>
<td>4,000</td>
</tr>
<tr>
<td>Raytracer</td>
<td>150</td>
<td>500</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>10,000</td>
<td>60,000</td>
</tr>
</tbody>
</table>

parallel for (diamond line) region approximately consists of the time needed to execute the parallel and the for region.

For the reduction clause (Fig. 12(b), square line), the overhead consists of the time needed for the parallel region (20 msec for 8 threads) and the time needed to combine the partial results of the workers. For 8 threads, the time needed for a +-reduction of a variable of type long is roughly 3 msec.

Fig. 13 displays the overheads of scheduling sizes in for loops for different numbers of nodes. For a given number of nodes, there is almost no difference in the overheads for different chunk sizes in static scheduling (first group of bars). Each worker thread receives a read-only copy of the LoopInformation object and can then work on its blocks. In contrast, the overhead of a dynamic loop strongly depends on the chunk size (second group of bars). For such loops, a worker thread gets the next block from a list of unprocessed blocks. As this list has to be synchronized among all other workers, a significant amount of DSM protocol activity is involved to maintain consistency. This overhead of the central block list decreases with growing block sizes, as less blocks need to be requested.

To determine the overhead of accessing shared JaMP variables compared to regular instance variables, we touched both types of variables $10^6$ times in separate loops. We then subtracted the loop overhead from both results. An access to a shared JaMP variable is about 2.6 times slower than an access to an instance variable. The higher latency of the shared variable access is caused by the fact that it resides in an array in the JampParam object. For shared variables, the values may not be cached locally as many threads may access the variable concurrently. In contrast, firstprivate variables can be cached after their initialization; subsequent accesses are overhead-free.

6.2. Java Grande Forum benchmarks

To assess the scalability of JaMP programs, we evaluated a subset of the parallel Java Grande Forum (JGF) benchmarks [21], for which the JOMP project provides OpenMP versions [18]. Section 1 was omitted from the measurements, because our micro-benchmarks already provide a detailed evaluation of the directives. We ported section 2 to JaMP by translating the JOMP
directives to the JaMP syntax. From section 3, we evaluated Raytracer and Monte Carlo. Euler is skipped, because it uses OpenMP features that are currently not supported by JaMP. To get a reasonable data size for a DSM, we used larger data sizes than the JOMP benchmark did (see Table I for the sizes used).

**Series** (section 2) computes the first $N$ coefficients of the function $f(x) = (x+1)^x$. It mainly uses transcendental and trigonometric functions to compute the coefficients. The main loop is parallelized by means of the **parallel for** directive.

The computation of Series is inherently parallel. After the threads have received their parts of the computation, no remote memory accesses are necessary. Fig. 14(a) shows that the Series benchmark scales perfectly for data size $B$ with a speed-up of roughly 7.9 on 8 nodes. Since for the smaller data size the overhead of start-up and finish cannot be compensated, the overall speed-up is just 6 on 8 nodes.

**Sparse** (section 2) computes (200 times) the product of two sparse $N \times N$ matrices in compressed-row format. The memory-intensive benchmark shows a high degree of indirection and non-regular memory references. The main loop is parallelized using the **parallel** directive; work is assigned by means of calls to `JampRuntime.getThreadNum()`.

The super-linear speed-up (see Fig. 14(b)) is caused by cache effects and disappears for the larger data set. As the number of nodes increases, the relative data size per node decreases. For larger data sizes, this effect is less significant, which is reflected in a speed-up of 7.9 on 8 nodes for the larger data size.

**Crypt** (section 2) performs IDEA encryption and decryption of $N$ bytes stored in an array. Crypt strongly depends on bit and byte operations. The main encryption/decryption loop is parallelized using the **parallel for** directive.

The kernel decrypts a sequence of bytes from one (read-only) array and writes them to a second (write-only) array. As Jackal partitions arrays into chunks of 64 KB, each thread receives only the parts of the array it is working on. As the array is allocated on node 0 initially, every other node needs to request its chunks from this central node. Second, in contrast to the read-only array, the write-only array causes false-sharing for chunks that are processed by two neighboring threads. The number of chunks affected by false-sharing does not exceed the number of threads. When the parallel region is terminated, each thread flushes the decrypted data to node 0. The central data-allocation scheme and the false-sharing limit the speed-up to roughly 4.7 on 8 nodes (see Fig. 14(c)).

Since Crypt would benefit from the required data being allocated with the respective thread, Jackal should support distributed allocation of a single array in the future.

**SOR** (section 2) is a simple over-relaxation (100 iterations) in a red-black style on an $N \times N$ grid. The inner loop is parallelized using the **parallel** directive while the outer loop over the grid is parallelized with the **for** directive. We also parallelized the data allocation using **parallel for** so that the nodes that work on a partition of the grid also perform the allocation. This is a well-known optimization technique for OpenMP programs on NUMA architectures.

As Fig. 14(d) shows, SOR scales reasonably well, because the amount of shared data is rather small compared to the amount of computation. Since only two rows of the matrix are shared between neighboring threads, the overhead of remote data accesses is small compared to the computational effort. However, there is a barrier after each iteration that synchronizes
(a) Speed-up of the Series benchmark.

(b) Speed-up of the Sparse benchmark.

(c) Speed-up of the Crypt benchmark.

(d) Speed-up of the SOR benchmark.

(e) Speed-up of the LU Fact benchmark.

Figure 14. Speed-ups of the Java Grande Forum benchmarks (section 2).
the SOR iterations. This accounts to roughly 1.1 sec of the sequential overhead which limits scalability.

It is obvious that increasing the matrix size also improves SOR’s scalability, because the computational effort grows quadratically while the DSM protocol and JaMP overheads grow linearly. Again, a better barrier implementation would improve the overall scalability of the benchmark. Furthermore, the compiler should analyze the code and provide means to automatically perform the distributed allocation, which we have implemented manually.

LU Fact (section 2) solves an $N \times N$ linear equation system by means of LU factorization and triangular backwards substitution. The benchmark mainly uses floating point arithmetics and is memory intensive. Only the Gaussian elimination is parallelized by means of parallel for; the rest is sequential. The matrix is accessed in column order, that is, a column is stored along the second dimension of the Java array.

For LU Fact we do not achieve any speed-up at all. LU Fact contains a rather large sequential kernel that is not parallelized and limits the theoretically possible speed-up. However, the main cause for the poor behavior is false-sharing. The sequential part of the kernel is executed by each thread concurrently. Hence, all threads access the same data at the same time. This causes the DSM runtime system to copy objects back and forth instead of spending time on the computational kernel. As can be seen from Fig. 14(e), the larger data size improves the poor result somewhat, as the ratio of DSM protocol activity to computation is slightly better. On an SMP system the effect is less severe, because the latencies of memory accesses for individual CPUs are much smaller than in a DSM environment.

In total, the parallelization of the LU Fact benchmark does not fit the requirements of an S-DSM system. In the current form, the code will not benefit from any optimizations that the Jackal DSM provides. A DSM-optimized parallelization of the benchmark has to be provided, which will show a speed-up.
Raytracer (section 3) renders a scene of 64 spheres in a picture with a resolution of $N \times N$, which is flattened into a 1D Java array. The main loop of the benchmark is parallelized with the `parallel for` directive. Hence, each thread renders a partition of the picture.

Raytracer works on a read-only data set (the spheres) and writes the picture information to the 1D array. As raytracing is an inherently parallel algorithm, the speed-up that is achieved by JaMP is close to 7 on 8 nodes (see Fig. 15(a)). For the small data size, scalability is weaker because of the relative overhead of synchronization and computation.

Monte Carlo (section 3) is a financial simulation that uses a Monte Carlo technique to compute product prices from the prices of the underlying assets. The kernel generates $N$ sample time series of equal mean and fluctuation. The main loop was parallelized with `parallel for`. Partial results are combined in a critical section.

As Fig. 15(b) shows, Monte Carlo also shows a slow-down for both data sizes. The benchmark shows the same data access behavior as LU Fact. False-sharing and a high amount of object allocations cause a lot of DSM protocol activity. Hence, the benchmark does not perform as desired, because it is blocking in calls to the DSM runtime.

7. Conclusion

We have shown an implementation of a Java-OpenMP binding called JaMP as an extension to Jackal’s native Java compiler. With JaMP, a programmer can write a purely sequential Java program and enrich it with parallelization directives to turn it into a parallel JaMP program. The directives are expressed as pragmas that are implemented by a special type of Java comment. Our current implementation of JaMP supports a large subset of the OpenMP directives and provides a programming model that is of similar expressiveness. We have suggested extensions to the OpenMP specification to make it fit better to the object-oriented programming style used in Java programs. The implementation of JaMP in the Jackal research compiler is sketched. The runtime overheads of the individual JaMP directives are small. OpenMP versions of most of the Java Grande Forum benchmarks JaMP achieve a reasonable speed-up of 5–8 on an eight node S-DSM system.

Download: The JaMP compiler is available as part of the Jackal project under a GPL license, see http://www2.cs.fau.de/Research/Projects/Jackal. Please feel free to contact one of the project members to receive a copy of the current JaMP code base.

REFERENCES