Reparallelization and Migration of OpenMP Programs

Michael Klemm, Matthias Bezold, Stefan Gabriel, Ronald Veldema, and Michael Philippsen
Computer Science Department 2 • University of Erlangen-Nuremberg
Martensstr. 3 • 91058 Erlangen • Germany
{klemm, veldema, philippsen}@cs.fau.de, bezold@msbezold.de, stefan-gabriel@gmx.net

Abstract

Typical computational grid users target only a single cluster and have to estimate the runtime of their jobs. Job schedulers prefer short-running jobs to maintain a high system utilization. If the user underestimates the runtime, premature termination causes computation loss; overestimation is penalized by long queue times. As a solution, we present an automatic reparallelization and migration of OpenMP applications. A reparallelization is dynamically computed for an OpenMP work distribution when the number of CPUs changes. The application can be migrated between clusters when an allocated time slice is exceeded. Migration is based on a coordinated, heterogeneous checkpointing algorithm. Both reparallelization and migration enable the user to freely use computing time at more than a single point of the grid. Our demo applications successfully adapt to the changed CPU setting and smoothly migrate between, for example, clusters in Erlangen, Germany, and Amsterdam, the Netherlands, that use different processors. Benchmarks show that reparallelization and migration impose average overheads of about 4% and 2%.

1. Introduction

While offering novel computing opportunities, the boundaries between individual clusters of a computational grid are still visible to users. In addition to heterogeneity as a problem, the user is faced with a cluster's job scheduling mechanism that assigns computing resources to jobs. Usually, the scheduler prefers short-running over long-running jobs and it prefers jobs that only need a small number of CPUs over more demanding ones. Short jobs with a only a few CPUs increase the cluster's utilization, while long-running jobs or jobs that require many CPUs often cause unproductive reservation holes [26]. To be fair to waiting users the job manager terminates jobs that exceed their claimed resource limit. A terminated application loses all computed work.

However, it is difficult to provide an exact estimation of a job's runtime, as often runtime depends on the input and is influenced by unpredictable environmental issues (e. g. the load of the network, which in turn depends on a cluster's overall load). Generally, two "solutions" exist for a user. First, a user can request a too long time slice and accept the penalty of more waiting time until the job runs. As an educated guess, a user might double the estimated time to avoid losing results upon termination of the program. Second, the program is rewritten into a number of smaller phases, which can then run within more predictable time boundaries.

Reparallelization and migration are not only a third solution to this problem but in addition they blur the boundaries between clusters. The user can start the application with a certain number of CPUs and a small time estimate at any cluster of the grid. If the application is about to exceed the time slice or if a more powerful cluster becomes available, the application checkpoints and migrates to another cluster, transparently adapting to the potentially different architecture and to a changed degree of parallelism.

Our solution consists of two parts: (1) OpenMP [15] programs can transparently alter the number of threads during the execution of a parallel region, and (2) checkpoint-based migration between clusters is supported which enables an application to halt on one cluster and to resume on another one. Both the origin and the target can be of different architectures. Reparallelization is crucial for migration, since the number of available CPUs is likely to change. And even without migration, reparallelization allows the next local resource reservation to request fewer or more CPUs depending on the overall system load and queuing times. Our prototype is implemented on top of the Software Distributed Shared Memory (S-DSM) Jackal [23], a shared memory emulation for Java on clusters. Jackal's compiler supports JaMP [12], an OpenMP 2.5 port to Java.

The paper is organized as follows. Section 2 covers related work. Section 3 describes the OpenMP reparallelization. Section 4 discusses the migration and the distributed checkpointing algorithm. Section 5 presents the performance of both OpenMP reparallelization and migration.

2. Related Work

To our knowledge, reparallelization and migration techniques for OpenMP programs are not available. Related work can roughly be divided into three categories: (1) reparallelization of OpenMP programs, (2) migration of processes and MPI programs, and (3) checkpointing. (2) and (3) are related as migration uses checkpointing.

Although the OpenMP specification allows to alter the number of threads per parallel region [15], in existing implementations except ours the number is fixed for the duration of the region. Adaptation of the thread count by the runtime system is restricted to code areas outside of parallel regions. Only the extension of OpenMP for irregular data structures proposed in [21] offers deferred cancellation. While new threads may not be created, worker threads can be scheduled to exit at certain cancellation points.

MOSIX [4], Sprite [6], and others, can migrate a process from one node to another. In contrast to our solution, they neither offer capabilities to change the degree of parallelism nor can they migrate on heterogeneous clusters. Our approach does not leave a process stub back at the old node to access immobile resources such as open files and network connections. Finally, we avoid kernel modifications, which we find unacceptable for general-purpose clusters.

Cactus [3] and DGET [8] can adapt to changes in the computing environment and acquire or release nodes while an application is running. However, both enforce own programming models in which the application has to extend a framework with callbacks to its application-specific functionality. Furthermore, our approach allows generic checkpointing without manual registration of data.

There is work that focuses on the migration of MPI programs from one cluster to another. In GrADS [22] the application registers its to-be-checkpointed data at a user-level checkpointing library. Similar to our approach, the application is migrated by checkpointing and restoring; the number of processors can be changed upon a restart. In contrast to our work, GrADS is limited to iterative MPI programs that are explicitly designed by the programmer for reparallelization. P-GRADE [13] checkpoints and migrates MPI/PVM applications. However, the application's degree of parallelism is fixed after a migration. The mobile MPI programs of [9] and AMPI [10] do not perform true reparallelization. Instead, the application is over-decomposed for a high number of virtual processors that are in turn mapped to actual MPI processes by multiplexing them in MPI function calls. In contrast to our approach, a major limitation is the maximum degree of parallelism. Being unrelated to the application's general scalability, an application cannot use more nodes than the number of virtual processors it started with.

Checkpointing [11] forms the basis of most migration approaches. Checkpointing libraries such as *libchkpt* [16], or kernel modules such as BLCR [7] that dump the address space of a process to disk cannot be used in heterogeneous environments. Porch [17] follows a compiler-based approach to provide heterogeneous checkpointing, but is limited to single processes. Multi-process checkpoints may be created by [18, 1, 14], and many more. LAM/MPI [18] offers multi-process checkpointing with BLCR and a coordinated (stop-the-world) algorithm. In [1], the application's control flow is analyzed to find locations of checkpointing initiations that achieve a consistent global state. In [14], synchronized clocks are employed to maintain a consistent state for a checkpoint. Our checkpointing approach uses a coordinated algorithm for simplicity, as the overhead of writing the heap and the thread stacks to disk completely hides the coordination overhead.

```
//#omp parallel
{
    ...
    //#omp adjust
    ...
}
```

Figure 1. Example of the adjust directive.

3. Reparallelization

While prior work creates a large number of virtual processors and maps the virtual to the physical processors (called *over-decomposition*), our approach modifies the actual parallelization and data partitioning at the application level. The advantage is that this does not limit the maximum degree of parallelism as done by over-decomposition.

In our approach, the number of worker threads can be changed at certain points in the program, called *adjustment points*. They can be inserted either manually by means of the *adjust* directive (see Fig. 1) or automatically by the compiler (future work). At adjustment points, new threads can enter a parallel region or existing ones can be terminated.

Our reparallelization covers all OpenMP constructs. Below we first discuss reparallelization of work-sharing constructs. Then we examine the differences between the loop schedule types. Finally, we study adjustment issues of reductions and parallel regions.

3.1. Repartitioning of work-sharing constructs

OpenMP programs often process a data structure in parallel by means of the *for* work-sharing construct that assigns different parts of an iteration space to the available worker threads. Hence, the reparallelization of the *for* directive is crucial for the reparallelization of OpenMP codes.

According to [15], size and shape of the iteration space of an OpenMP *for* construct are known before the loop

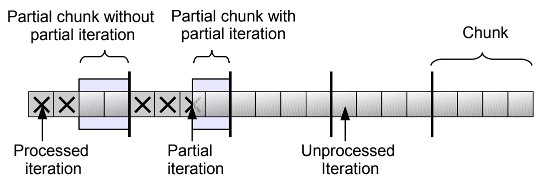
Partial chunk without partial iteration
Partial chunk with partial iteration
Chunk
Processed iteration
Partial iteration
Unprocessed Iteration

Figure 2. Iteration space layout with iterations, chunks, and partial chunks.

starts. The iteration space can be divided into chunks of a fixed size that are then processed by the worker threads.

A dynamic reparallelization must take into account what fraction of the iteration space is already computed, and what remains to be done by the changed number of threads, see Fig. 2. To be flexible, adjustment points may be placed at arbitrary code locations. Hence, at some point in time, some chunks are still unprocessed, others are partially processed (i. e. some of their iterations are done), and some chunks are completed. In the partially processed chunks some iterations are completed, while other iterations still need to be done. There might even be single unfinished iterations.

If a worker thread is removed at an adjustment point in the body of a work-sharing construct, another thread has to take over the remaining work. Hence, a *description of a partial chunk* has to be created and stored in a set of still uncompleted chunks. In addition to an *iteration space description* (loop counter value, the lower and upper bounds, and the step width or—in more complex situations—a bit vector) the partial chunk description must store the number of the adjustment point (i) that caused its creation, and the values of all live variables for subsequent use by another worker. Standard compiler analysis is used to identify this set of live variables. To copy the values, their type must be known to the compiler. Hence, our approach relies on type-safe environments.

When new threads are added to the work force, they can grab any unprocessed chunk and start executing the region's code for that chunk. (We skip the question where the unprocessed chunks come from for now.) If no unprocessed chunks are left, new threads have to take on partial chunks. But instead of starting to execute the region's code from its beginning, it is necessary to jump to the position of the adjustment in the code. Hence, we need a jump table to branch to the appropriate adjustment point (START_i).

With partial chunks and the jump table explained, we are ready to discuss the compiler's code template for the i-th adjustment point (see Fig. 3). When an adjustment is requested, a worker first checks if it is selected for termination. If so, it stores a partial chunk description and terminates. The master thread (ID 0) adjusts JaMP's internal data structures if the number of workers has changed. When a thread takes on a partial chunk for adjustment point i, it

```
    if (adjustmentRequested()) {
        if (removeSelf()) {
            storePartialChunk();
            doTermination();
        } else if (threadId() == 0
              && changeThreadCount()) {
            adjustDataStructures();
            createThreads();
        }
        finishAdjustment();
    }
    goto END_i
START_i:
    loadPartialChunk();
END_i:
```

Figure 3. Code template adjustment points.

```
barrierIncrement();
boolean finished = false;
while (! barrierGoalReached()
    || partialBlocksAvailable()) {
    partialBlock = popPartialBlock();
    if (partialBlock != null) {
        barrierDecrement();
        startLabel = partialBlock.getLabel();
        goto JUMP_TABLE;
    }
    barrierWait();
}
```

Figure 4. Reparallelization barrier at work-sharing constructs.

jumps to the START_i label, loads the iteration space information and the live variables from the partial chunk description, and starts execution.

At the end of a work-sharing construct there must always be a barrier synchronization. If the parallel region has an adjustment point, the barrier code is more complex, since partial chunks might remain to be processed by the existing threads. The skeleton code given in Fig. 4 shows that a work-sharing construct is completed when (1) all chunks have been processed, (2) no partial chunks are left, and (3) all worker threads have arrived at the loop barrier construct.

3.2. Loop schedule types

OpenMP defines different schedule types for the assignment of loop chunks to worker threads. While the compiler uses the common code transformation template discussed in Section 3.1 for all types, there are schedule specific issues.

If *no loop schedule* type is specified, the iteration space is divided such that every worker thread receives exactly one chunk. If an adjustment adds a new thread, at least one new chunk is needed as well. Hence, the whole iteration space has to be redistributed. This requires a bit vector in which a bit is set for every finished iteration. At the adjustment point, a new set of chunks can then be created by first computing the unprocessed iterations, dividing them by the new number of worker threads, and then assigning the same

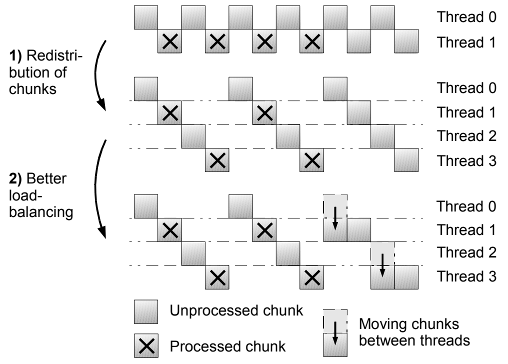

Figure 5. Static reassignment of chunks.

number of unprocessed iterations to each of the threads.

Since in a *dynamic loop schedule* threads request small chunks from a global work pile anyway, new worker threads can participate in the computation without further effort.

In a *static loop schedule*, all chunks are assigned to the worker threads at the beginning of the loop in a round-robin fashion. Therefore, this assignment has to be updated when the number of worker threads changes. Since a new assignment can only be computed if it is known which chunks have already been processed, every worker thread memorizes the list of completed chunks. Reassignment is done in two steps. First, all chunks are assigned as usual to the worker threads in a round-robin fashion. Since completed chunks are marked, they can be skipped later on. Step 1 creates an unbalanced load since some worker threads receive more unprocessed chunks than others. Step 2 relocates unprocessed chunks from over-loaded threads to under-supplied ones to achieve a better load-balancing. We have adapted an algorithm from [2] for this purpose.

An example of static reassignment is given in Fig. 5. The iteration space is distributed over two threads. Thread 0 has not yet completed any chunks, whereas thread 1 has computed four chunks (marked "X"). After doubling the number of threads, step 1 redistributes the chunks to four threads in a round-robin fashion. While threads 0 and 2 each receive three unprocessed chunks, the other two only receive one uncomputed chunk. To improve load-balance, step 2 moves two chunks to lighter-loaded threads.

3.3. Reductions

We now discuss the reparallelization of OpenMP reductions that combine the partial results accumulated by all threads into a single result at the end of a parallel region.

In work-sharing constructs, every single iteration contributes to the global result. In case of reparallelization, only termination of threads needs special treatment. There are two cases. First, a thread that is terminated at the beginning of a work-sharing construct still has to contribute its already accumulated partial result to the reduction. Second, a thread that is terminated inside the loop body stores its partial results in the partial chunk descriptor. When another thread takes on this partial chunk, it can no longer simply *copy* all the live variables from the partial chunk descriptor. Instead of overwriting it, the reduction value has to be *merged* with the thread's partial result.

```
//#omp parallel
{
    Collector c = ...;
    int z = ...;
    //#omp for
    for (int i = ...) {
        //#omp adjust
        compute(i, c, z);
    }
}
```

Figure 6. Variable context at an adjustment point.

3.4. Context for added worker threads

When a worker thread takes on a partial chunk, all live variables are contained therein. On the other hand, when new worker threads are added to a work-sharing construct, they start execution at the beginning of the construct. If there are live variables that have been defined outside of the work-sharing construct, they have to be initialized with valid values for the new worker threads. For that purpose, the set of live variables is stored by the master thread directly before the work-sharing construct and a copy of it is loaded by the added worker threads.

In Fig. 6, a variable *c* of a class called *Collector* is created inside of the parallel region by every worker thread and used in some computation. Moreover, there is a variable *z* of type *int*. Both *c* and *z* are live variables from outside of the work-sharing construct. When a worker thread is added at the adjustment point, it has to receive valid values for *c* and *z*. Therefore, the set of live variables is initialized from the live variables of the master thread.

The values of variables of primitive data types (such as *z*) are copied. The *clone()* method is used for objects of classes that implement the *Cloneable* interface. For other classes and if standard cloning does not produce a valid copy, special cloning facilities can be provided by the user.

3.5. Limitations

Our approach has the following three limitations:

a) Reparallelization can only be performed if the complete information about the parallelization and the partitioning of the work-sharing constructs is available. In Fig. 7 parallelization is done by OpenMP, but work distribution is programmed explicitly. The example code introduces an indirect data dependence from the thread ID to *array[i]*. Due

```
void dependent(double[] array) {
    //#omp parallel
    {
        int id = JampRuntime.getThreadNum();
        int cnt = JampRuntime.getNumThreads();
        int sz = array.length / cnt;
        int fr =  sz * id;
        for (int i = fr; i < fr + size; i++) {
            // computation using array[i]
        }
    }
}
```

Figure 7. Dependency to the thread ID.

```
void independent(double[] array) {
    //#omp parallel for
    for (int i = 0; i < array.length; i++) {
            // computation using array[i]
        }
    }
}
```

Figure 8. Corrected example of Fig. 7.

to that data dependence a reparallelization causes undefined behavior, as the thread with a given ID might have been removed. Hence, we disallow the use of *getThreadNum()* and *getNumThreads()* and the compiler issues warnings. Fig. 8 is the correctly parallelized version of Fig. 7.

b) If worker threads are added to or removed from a parallel region, the number of times every statement is executed can change. This might affect program semantics. Assume a thread is removed at the barrier in Fig. 9. Then fewer "B"s than "A"s are printed. If such a behavior is undesired, the programmer can disallow reparallelization by adding the *adjust(none)* clause to the *parallel* directive.

c) As mentioned above, the live variables of the master thread are copied to a newly created thread. However, this might not be the desired application's semantics. For example, when a variable is used to store the current thread ID, the new thread would receive the ID of the master thread. The programmer has to be aware of this issue and deal with it accordingly. However, since in most OpenMP programs the use of a thread ID indicates a weak application design, we consider this to be an acceptable restriction of the programming model.

4. Migration

Checkpointing the DSM space of a Jackal application forms the basis of our migration approach. Checkpointing allows to save the computational state of an application [11]. The state can then be transferred to another cluster, on which the application is resumed. We have presented a compiler-based approach for migrating threads in heterogeneous clusters in [24]. It provides a means to checkpoint the computational state of a single thread, to move the state to another machine of a potentially different archi-

```
//#omp parallel
{
    System.out.println("A");
    //#omp barrier
    System.out.println("B");
}
```

Figure 9. Limitations example.

tecture, and to continue the computation there. For migration of OpenMP applications, this papers adds a coordinated checkpointing algorithm for multi-process applications. We first sketch how a checkpoint is created for a single thread. We then shortly describe the extensions made to support checkpointing of multi-process applications.

4.1. Thread Checkpointing

A generic stackframe format to store the current stack of a thread is the basis of platform-independence in [24]. For each call-site of a function, the compiler creates so-called *checkpointers* and *uncheckpointers*. Checkpointers map the stackframe of the function at that call-site to a generic, machine-independent format. In turn, uncheckpointers load a function's stackframe from the generic format.

The computational state at a given call-site can be described by the live variables at that location. For each of the live variables, the compiler creates a unique *Usage Descriptor String* (UDS) that platform-independently describes the variable. The generic stackframe format consists of a set of tuples (UDS, value). As described in [24], the creation of the UDS is mainly based on the following assumption: the value of a variable A at a given point in a program is determined by the preceeding computation H that affected A. This computation has to be the same on all architectures. Otherwise, the program would compute different results on different architectures. Hence, the central idea is to encode H in the UDS. For brevity, only the rules for constructing a UDS for H are listed in Fig. 10. The details can be found in [24].

As an example of how to construct the UDS, let us consider a Java function (Fig. 11) with three basic blocks (B0 through B2). At the call-site of *createCheckpoint()* in B2, two variables a and b are live. According to the rules R1 and R7 of Fig. 10, the compiler creates the UDS "*C:1000@B:0*" for b, which reads as "variable b is initialized with value 1000 in basic block B0". A more complex UDS is created for a. The compiler searches backwards from the call-site in B2 to find all reaching definitions for a. For the assignment in B1 it creates the partial UDS "*+ a C:1@B1*" (R6 & R7). It then continues to search for the contained a. This delivers "*C:0@B0*" (R1 & R7). Hence, the final UDS for a at the call-site is "*+ C:0@B0 C:1@B1*", a platform-independent description of the computation of the value of a in line 8.

The algorithm above forms the basis for our checkpoint-

R1 *"A = constant"* $\rightarrow$ *string(A) = "C:<constant>".*

R2 *"A = B"* $\rightarrow$ *"string(A) = string(B)".*

R3 *"A = call"* $\rightarrow$ *string(A) = "call:<index of call in all calls inside the containing function>".*

R4 *"A = param(X)"* $\rightarrow$ *String(A) = "P:<index of param(X)>".*

R5 *"A = object_access(expression, field)"* $\rightarrow$ *string(A) = "access:field"* and recurse into *expression*.

R6 *"A = B op C"* $\rightarrow$ *string(A) = "< op >" + string(B) + string(C)*, where *op* is one of the binary operands such as $+, -, *, /$.

R7 When making a modification to an UDS by one of the above rules, add a basic block identifier to the string: *string(A) = string(A) + "@B:<basic-block-number>"*

Figure 10. UDS construction rules.

```
      int someFunction(int c) {
B0        int a = 0;
B0        int b = 1000;
B0        if (c != 0)
B1            a = a + 1;
B2        // live variables: {a,b}
B2        createCheckpoint();
B2        return a + b;
      }
```

Figure 11. Java checkpointing example.

ing approach. A thread is checkpointed by sequentially calling the checkpointers for each function on the call stack. In addition, the reachable objects in memory are traversed and written to disk. In contrast to standard Java serialization [20], this is done asynchronously. Chunks of data are handed off to a service thread for compression and for writing the compressed data to disk with bulk transfers. Hence, the checkpointer does not need to wait for the disk to catch up. As soon as the checkpointer has finished, the application can continue while the checkpoint data is still being written to disk in the background.

4.2. Multi-process checkpointing

A Jackal application consists of a set of processes that together form the application. With JaMP, each process receives one OpenMP thread. However, Jackal processes contain not only application-specific worker threads but also execute several service threads such as the garbage collector and the finalizer thread that both are needed by the Java runtime. To checkpoint a multi-process application, we have implemented a coordinated checkpointing algorithm. It (1) stops all local threads, (2) ensures that no messages are on the network, and (3) checkpoints all the threads of a process.

A single-process application is checkpointed by saving the state of all threads and the heap to disk. This is accomplished by collecting all threads in a special barrier, effectively stopping all threads. Whenever, a thread reaches a synchronization point (e. g. *Object.wait()*, *Thread.sleep()*) it checks whether a checkpoint is requested. If so, it triggers the thread checkpointing algorithm. Otherwise, it proceeds with the regular synchronization code.

Already blocked threads that wait for a notification or a timeout to occur need a special treatment. To inform them about a new checkpoint request, we wake them up by means of a *CheckpointException*. The threads then checkpoint and re-enter their waiting state. Hence, from an application point of view, they seem to never leave their blocking state.

Multi-process checkpointing is achieved by means of coordinated checkpointing [11]. To request a checkpoint, the master node sends a broadcast to all other nodes. Due to the FIFO property of Jackal's communication layer, this broadcast message pushes all data messages through the network to their respective receivers. As soon as another node receives a checkpoint request message, it informs its threads about the checkpoint request. All threads continue until they reach either a synchronization point or a call of *createCheckpoint()*. After all local threads are blocked, the nodes send a broadcast message to clear the network of application traffic. Then, all local threads start to write the checkpoint. This implements a race-free stop-the-world approach, since all threads block and the network is clear of messages. Hence, no rollback operations are necessary upon restarting from a checkpoint.

4.3. Interaction with reparallelization

When the application resumes from the checkpoint, the number of nodes that execute the program can change. If the new number is lower, the runtime system merges the images of the removed nodes into the resuming ones. If the number is increased, new nodes start with empty images.

The reparallelization runtime directly interacts with the checkpointing system by being notified of changes whenever an application is resumed from a checkpoint. The runtime then starts the reparallelization of the currently active parallel region and adapts the thread count such that each node receives one OpenMP thread. Data distribution is handled automatically by the Jackal DSM runtime system.

5. Performance

To show the feasibility of our reparallelization and migration approach, we have undertaken a set of experiments. The experiments were performed on a cluster of quad (2x dual core) AMD Opteron 2.0 GHz 64 bit nodes with 4 GB of main memory and Gigabit Ethernet. The cluster is located at the University of Erlangen, Germany. To show the effects of network traffic, we only used one CPU per node. The results presented are the average over 5 runs of each benchmark. Overheads are computed as the relative

Table 1. Runtimes, overheads, and checkpoint sizes (in MB) for the benchmark suite.

Thr.	LBM				SOR				Crypt				Raytracer			
	Time (sec)	Adjust-ment Overhead	Check-point Overhead	CP size	Time (sec)	Adjust-ment Overhead	Check-point Overhead	CP size	Time (sec)	Adjust-ment Overhead	Check-point Overhead	CP size	Time (sec)	Adjust-ment Overhead	Check-point Overhead	CP size
1	924	**-2.5%**	0.3%	117	430	0.9%	0.1%	16	91	9.7%	3.1%	80	473	3.6%	1.7%	1
2	467	**-2.1%**	0.2%	122	221	2.9%	0.3%	16	52	9.9%	2.6%	80	234	2.3%	1.7%	1
4	242	0.4%	0.3%	128	118	**12.9%**	-0.2%	18	29	5.1%	5.5%	80	117	3.1%	2.7%	1
8	127	2.6%	1.1%	136	65	**15.9%**	0.7%	19	19	3.5%	8.8%	80	60	3.0%	4.7%	1

increase of runtime of each benchmark without adjustment points and/or checkpointing.

5.1. Benchmarks

For our evaluation we implemented a compute intensive *Lattice-Boltzmann Method* (LBM) [25] benchmark. In addition, we use JOMP's Java-OpenMP port of the JGF benchmarks [19, 5]. We skip section 1 of the benchmark suite since it solely contains microbenchmarks for individual OpenMP directives, such as creation of parallel regions. From section 2 and 3, we study *SOR*, *Crypt*, and *Raytracer*. (*Euler* uses the unsupported OpenMP construct *ordered*. *Sparse* introduces data dependencies to the thread ID, which we disallow. *LU Fact* and *Monte Carlo* are not suited to be executed on a DSM system as the JGF versions contain large sequential fractions and/or suffer from false-sharing.)

LBM simulates fluids with cellular automata. Space and time are discretized and normalized. In our case, LBM operates on a 3D domain divided into 120$\times$120$\times$120 cells that hold a finite number of states. In one time step the whole set of states is updated synchronously by deterministic, uniform update rules. The kernel is parallelized in a straightforward way; the time-stepping loop is parallelized with *parallel* and the loop over the *x*-axis of the grid is parallelized with the *for* directive (default scheduling). We have also parallelized the data allocation using *parallel for* such that the nodes that work on a partition of the grid also perform the allocation. This is a well-known optimization for OpenMP programs on NUMA architectures. The benchmark computes 50 time steps over the 3D grid.

SOR solves a discrete Laplace equation with simple over-relaxation (200 iterations) in a red-black style on an 10,000$\times$10,000 grid. The outer loop is parallelized with the *parallel* directive while the inner loop over the grid is parallelized with the *for* directive and default scheduling. The data allocation was parallelized with *parallel for*. *Crypt* performs IDEA encryption and decryption of 140 MB of data and strongly depends on bit and byte operations. The main encryption/decryption loop is parallelized with a *parallel for* with default scheduling. The *Raytracer* renders a scene of 64 spheres in a picture with a resolution of 800$\times$800. The main loop of the benchmark is parallelized with the *parallel for* directive and *dynamic* scheduling with chunk size 10. Hence, each thread renders a partition of the picture. Raytracer works on a read-only data set (the spheres) and represents the picture as an 1D array.

5.2. Overheads

Table 1 shows the runtimes of the individual benchmarks. Overall, the average overhead for supporting dynamic adjustment of the thread count is approximately 4%, which can be considered acceptable. The overhead is determined by the amount of work of the parallel region, as the code transformation adds a constant overhead. Checkpointing imposes a runtime overhead of roughly 2% on average when creating one checkpoint during the execution of the benchmark. The overhead is influenced by the data set of the application and is almost unrelated to the disk transfer rate. For realistic applications and realistic data sizes, the overhead is negligible (below 1%).

Inserting adjustment points into the SOR kernel decreases performance by upto 15.9% (see Table 1). This large overhead is caused by adding a constant overhead per adjustment point to a very low runtime per iteration. Thus, the relative overhead per iteration becomes significant. For LBM a negative overhead of roughly 2% can be observed due to processor caching effects.

5.3. Speedup

For the speedup measurements, we have set up Jackal such that half of the processes received OpenMP threads while the other half is idling. At benchmark half-time, the thread count is doubled. The additional threads are spawned on the idle nodes. This scenario is reversed for the removal of workers.

Fig. 12 shows the runtime per LBM time step over time. At time step 25, the number of threads is doubled. As can be seen on the left, the runtime roughly decreases by a factor of 1.8. A slow-down of of about 2 occurs when the number of threads is halved (middle). This closely matches the speed-up behavior of LBM (on the right). The peak run-

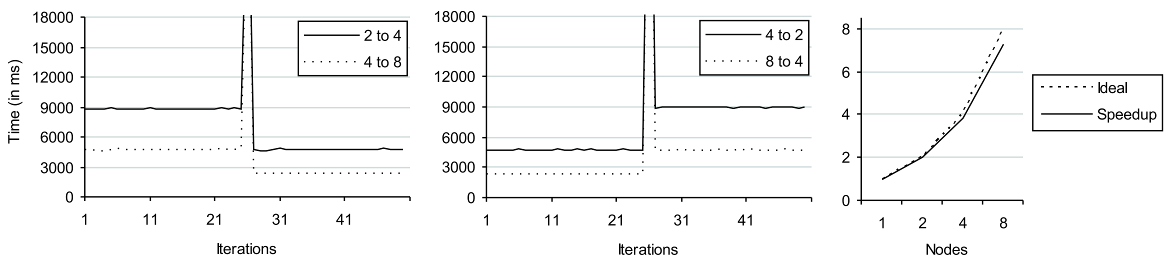

Figure 12. Runtime per LBM time step and speed-up graph of LBM.

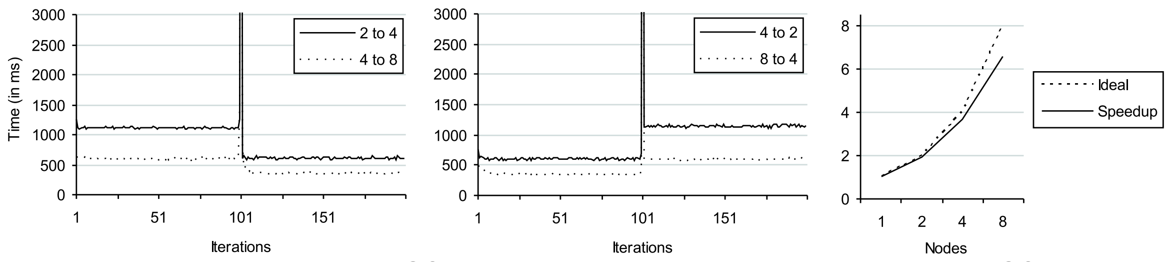

Figure 13. Runtime per SOR red-black iteration and speed-up graph of SOR.

time after the adjustment in iteration 25 is caused by the DSM protocol that needs to redistribute the data after the reparallelization, i. e. that moves the data accessed (roughly 3.6 LBM cells) by the threads to their respective execution nodes. Note that such delays are caused by any NUMA system and the height strongly depends on the latencies of the NUMA implementation. In addition, a NUMA implementation that allows migration of data is desirable to avoid performance penalties after reparallelization.

A similar result is achieved for SOR. As Fig. 13 shows, the runtime of a single SOR red-black iteration is decreased by a factor of 1.8 when the number of threads is doubled at iteration 100. The runtime peak at iteration 26 is again caused by the DSM runtime that needs to redistribute data (10,000 arrays or 10,000 messages over the network). Crypt and Raytracer also show the desired speed-up behavior (not depicted for brevity), when the number of threads changes. When doubling the thread count, both applications achieve a speedup of about 1.9, while they yield a slowdown of 1.9 when the thread count is halved. The runtime peak for Crypt and Raytracer is lower, as the data that needs to be redistributed is smaller.

5.4. Migration

To demonstrate our approach, we migrated LBM from the cluster at Erlangen, Germany, to a cluster at the Vrije Universiteit in Amsterdam, the Netherlands, and back. The cluster in Amsterdam uses dual Intel Pentium 3 CPUs with a 1 GHz processor clock, 1 GB of memory, Gigabit Ethernet, and Myrinet for each node. LBM is migrated two times: (1) at the 16th time step from Erlangen to Amsterdam, and (2) at the 33rd time step from Amsterdam back to Erlangen.

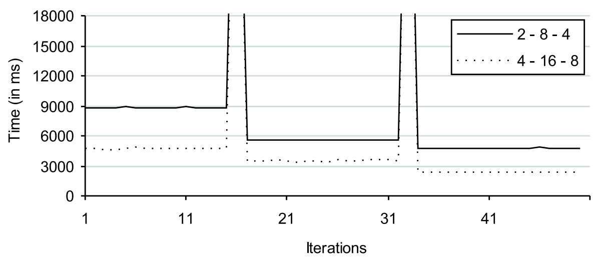

Figure 14. Migration of LBM.

Fig. 14 shows the times for one LBM time step. After the 16th iteration we have manually aborted and migrated the application from Erlangen to Amsterdam. The number of CPUs was hereby increased by a factor of four. The performance roughly doubles as the target CPUs are slower than the originating CPUs. In time step 33, LBM is moved back to Erlangen. This time, the number of CPUs was halved. Please note, that the time to transfer the checkpoint image and to wait for the cluster reservation are not included. The time to transfer the checkpoint of LBM between the clusters roughly was 50 sec. The total queue time in the cluster queues was about 5 minutes.

6. Conclusion

We have presented a novel approach to reparallelize and to migrate OpenMP applications between clusters of different size and architecture. This helps to make the boundaries of individual clusters in a computational grid less visible. A user can start an application at an arbitrary cluster in the grid. When the time slice is about to be exceeded, a checkpoint can be created. The application can either migrate to another cluster or restart on the current system with a new reservation. Reparallelization automatically adapts to the new number of available processors. Reparallelization is restricted to (1) well-formed OpenMP programs, and (2) type-safe programming languages.

Benchmarking shows that the reparallelization imposes little overhead and scales as expected. When the number of threads is changed, the new parallelization achieves speed-ups that are comparable to the regular speed-up behavior of the application with that number of processors. The overhead of inserting extra code for adjustment points is almost negligible compared to the overall runtime. The same holds for the overhead of checkpointing the application state.

References

[1] A. Agbaria and W. Sanders. Application-Driven Coordination-Free Distributed Checkpointing. In *Proc. of the 25th IEEE Intl. Conf. on Distributed Computing Systems*, pages 177–186, Columbus, OH, USA, June 2005.

[2] G. Aggarwal, R. Motwani, and A. Zhu. The Load Rebalancing Problem. In *Proc. of the 15th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 258–265, San Diego, CA, USA, June 2003.

[3] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *Intl. Journal of High Performance Computing Applications*, 15(4):345–358, Winter 2001.

[4] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[5] J. Bull, M. Westhead, M. Kambites, and J. Obdržálek. Towards OpenMP for Java. In *Proc. of the 2nd European Workshop on OpenMP*, pages 98–105, Edinburgh, U.K., September 2000.

[6] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, August 1991.

[7] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkley National Laboratory, 2003.

[8] T. Ellahi, B. Hudzia, L. McDermott, and T. Kechadi. Transparent Migration of Multi-Threaded Applications on a Java Based Grid. In *Proc. of the IASTED Intl. Conf. on Web Technologies, Applications, and Services*, Alberta, Canada, July 2006. no page numbers.

[9] R. Fernandes, K. Pingali, and P. Stodghill. Mobile MPI Programs in Computational Grids. In *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 22–31, New York, NY, USA, March 2006.

[10] C. Huang, O. Lawlor, and L. Kale. Adaptive MPI. In *Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 306–322, College Station, TX, USA, October 2003.

[11] S. Kalaiselvi and V. Rajaraman. A Survey of Checkpointing Algorithms for Parallel and Distributed Computers. *Sadhana*, 25(5):489–510, October 2000.

[12] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. JaMP: An Implementation of OpenMP for a Java DSM. *Concurrency - Practice and Experience*. To appear.

[13] J. Kovács and P. Kacsuk. Server Based Migration of Parallel Applications. In *Proc. of the 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 30–37, Linz, Austria, 2002.

[14] S. Neogy, A. Sinha, and P. Das. Distributed Checkpointing Using Synchronized Clocks. In *Proc. of the 26th Ann. Intl. Computer Software and Applications Conf.*, pages 199–204, Oxford, U.K., August 2002.

[15] OpenMP Application Program Interface, Version 2.5, May 2005. `http://www.openmp.org/`.

[16] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter Techn. Conf.*, pages 213–223, New Orleans, LA, USA, January 1995.

[17] B. Ramkumar and V. Strumpen. Portable Checkpointing for Heterogeneous Architectures. In *27th Intl. Symp. on Fault-Tolerant Computing - Digest of Papers*, pages 58–67, Seattle, WA, USA, June 1997.

[18] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *Intl. Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[19] L. Smith, J. Bull, and J. Obdrẑeálek. A Parallel Java Grande Benchmark Suite. In *Proc. of the 2001 ACM/IEEE Conf. on Supercomputing*, pages 97–105, Denver, CO, USA, November 2001.

[20] SUN Microsystems. Java Object Serialization Specification, November 1998.

[21] M. Süß and C. Leopold. Implementing Irregular Parallel Algorithms with OpenMP. In *Proc. of Euro-Par*, pages 635–644, Dresden, Germany, August 2006.

[22] S. Vadhiyar and J. Dongarra. Self Adaptivity in Grid Computing. *Concurrency - Practice and Experience*, 17(2–4):235–257, February/April 2005.

[23] R. Veldema, R. Hofman, R. Bhoedjang, and H. Bal. Runtime Optimizations for a Java DSM Implementation. In *2001 Joint ACM-ISCOPE Conf. on Java Grande*, pages 153–162, Palo Alto, CA, USA, June 2001.

[24] R. Veldema and M. Philippsen. Near Overhead-free Heterogeneous Thread-migration. In *Proc. of the IEEE Intl. Conf. on Cluster Computing*, pages 145–154, Boston, MA, USA, September 2005.

[25] D. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Number 1725 in Lecture Notes in Mathematics. Springer, 2000.

[26] D. Zotkin, P. Keleher, and D. Perkovic. Attacking the Bottlenecks of Backfilling Schedulers. *Cluster Computing*, 3(4):245–254, December 2000.