Reparallelization techniques for migrating OpenMP codes in computational grids

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@msbezd.de, stefan-gabriel@gmx.net

SUMMARY

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 kinds and numbers of processors. Benchmarks show that reparallelization and migration impose average overheads of about 4% and 2%, respectively.

1. Introduction

While offering novel computing opportunities, the boundaries between individual clusters of a computational grid are still visible to users. In addition to the problem of heterogeneity (e.g., different architectures and different interconnects), the user is faced with a cluster’s job scheduling mechanism that assigns computing resources to jobs. The scheduler asks the user for an estimation of the job’s runtime when the job is submitted to the cluster. From all submitted jobs, the scheduler then creates an execution plan that assigns the jobs to nodes. 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 only a few CPUs increase the cluster’s utilization, while long-running jobs or jobs that require many CPUs often cause
unproductive reservation holes [32]. To be fair to waiting users, the job manager terminates jobs that exceed their claimed resource limit. This causes computation loss.

However, it is difficult to provide an exact estimation of a job’s runtime, as often runtime depends on the application’s implementation, the input data, locality issues, available compilers, compiler optimization levels, and many more. Moreover, a job’s runtime is influenced by unpredictable environmental issues in the cluster (e.g. the load of the network, which in turn depends on a cluster’s overall load). Current practice provides two “solutions” to a user. First, a user can request a too long time slice and accept the penalty of an increased waiting time until the job eventually runs. As an educated guess, a user might double the estimated time or request the maximum permissible runtime to avoid losing results upon termination of the program. Second, the program is decomposed into a number of smaller phases that can then run within more predictable time boundaries. After each phase, the application saves to disk intermediate results computed so far. If the application is terminated after one of the phases, it can resume from the saved data and, thus, avoid losing work that was computed by earlier phases. However, this solution increases the application’s complexity and makes its implementation difficult to manage in most cases.

We propose a solution to this problem that does not suffer from the above mentioned drawbacks. In addition, our solution also makes Grid computing more transparent. If an OpenMP application is about to exceed the reserved time, it is automatically checkpointed and transparently migrated to either a new local reservation or to a reservation on another accessible (possible remote) system. If another, more powerful, cluster becomes available the application might migrate as well to increase the application’s computing speed.

Our solution consists of two orthogonal parts: (1) OpenMP [21] programs can transparently alter the number of threads during the execution of a parallel region, and (2) checkpoint-based migration between clusters is supported that 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 [29], a shared memory emulation for Java on clusters. Jackal’s compiler supports JaMP [16], 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 approach and the distributed checkpointing algorithm. Section 5 presents the performance of both OpenMP reparallelization and migration. Future work is discussed in Section 6; Section 7 concludes.

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. Categories (2) and (3) are related as migration often relies on checkpointing.
Although the OpenMP specification allows for altering the number of threads that process a parallel region [21], in existing implementations except ours this 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 [27] offers deferred cancellation. While new threads may not be created, threads can be scheduled to exit at certain cancellation points.

MOSIX [5], Sprite [8], 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 between 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 [4] and DGET [10] can adapt to changes in the computing environment and can acquire or release nodes while an application is running. However, both enforce their 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 [28] the application has to register its to-be-checkpointed data at a user-level checkpointing library. Similar to our approach, the application is migrated by checkpointing and restoring it; 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. In [19], an extension to the MPI programming model is proposed to inform the MPI library about how to redistribute data and work when the number of processes is changed. Our approach does not necessarily force the programmer to accept a new programming model, as our OpenMP programs may remain unchanged. P-GRADE [17] checkpoints and migrates MPI/PVM applications. However, the application’s degree of parallelism is fixed after a migration. The mobile MPI programs of [11] and AMPI [13] 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.

In general, checkpointing [15] forms the basis of most migration approaches. Checkpointing libraries such as *libchkpt* [22], or kernel modules such as BLCR [9] that dump the address space of a process to disk cannot be used in heterogeneous environments. Porch [23] follows a compiler-based approach to provide heterogeneous checkpointing, but is limited to single processes. Multi-process checkpoints may be created by [24, 1, 20], and many more. LAM/MPI [24] 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 checkpoint initiations that achieve a consistent global state. In [20], 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.
3. Reparallelization

Whereas 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. With over-decomposition, the number of tasks that are multiplexed is fixed before the application is started. Increasing the number of CPUs eventually results in a one-to-one mapping of tasks to CPUs. Further increases of processing speed cannot be achieved anymore since the number of CPUs exceeds the number of tasks.

Our approach overcomes this limitation by changing the number of threads 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. 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 the reparallelization of work-sharing constructs. Then we examine the differences between the different OpenMP loop schedule types. Finally, we study adjustment issues of reductions and parallel regions.

3.1. Repartitioning of Work-sharing Constructs

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

According to the OpenMP specification [21], size and shape of the iteration space of an OpenMP for construct are known up-front before the loop starts. The iteration space can then be divided into chunks of a fixed size that are processed by the threads.

A dynamic reparallelization must take into account what fraction of the iteration space has already been 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, there can be completed chunks, partially processed chunks, and unprocessed chunks. As for partially completed chunks, there may be finished and unfinished iterations. Moreover, a partially computed chunk might also contain unfinished, but already started iterations.
If a thread is removed at an adjustment point in the body of a work-sharing construct, another thread has to take on 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 thread. Standard compiler analysis is used to identify this set of live variables [3]. 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. If a chunk is resumed, it is necessary to jump to the code location of the adjustment point instead of starting from the beginning of the parallel region. A jump table is used to branch to the appropriate adjustment point (START_i).
1 barrierIncrement();
2 boolean finished = false;
3 while (! barrierGoalReached() || partialBlocksAvailable()) {
4 partialBlock = popPartialBlock();
5 if (partialBlock != null) {
6 barrierDecrement();
7 startLabel = partialBlock.getLabel();
8 goto JUMP_TABLE;
9 }
10 }
11 barrierWait();

Figure 4. Reparallelization barrier at work-sharing constructs.

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 thread 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 threads has changed. When a thread takes on a partial chunk for adjustment point $i$, it 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 contains 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 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 threads. While the compiler uses the common code transformation template discussed in Section 3.1 for all types, there are issues specific to the schedule type of the loop.

If no loop schedule type is specified, the iteration space is divided such that every thread receives exactly one chunk. If an adjustment adds a new thread, at least one new chunk is needed as well to make use of the added thread. 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 counting the unprocessed iterations, dividing them by the new number of threads, and then assigning the same number of unprocessed iterations to each of the threads.

With the dynamic loop schedule, threads request small chunks from a global work pile. Thus, new threads can participate in the computation without further effort of redistributing chunks. Each new thread starts to work by requesting the next uncomputed chunk from the global work pile. The same applies to guided loop schedules. For these loops, the chunk size is computed as
a function of the number of remaining chunks and the current number of threads. Hence, new threads also request a chunk from the work pile and the runtime system automatically adapts the chunks size according to the OpenMP specification.

In a static loop schedule, all chunks are assigned to the threads at the beginning of the loop in a round-robin fashion. Therefore, this assignment has to be updated when the number of threads changes. Since a new assignment can only be computed if it is known which chunks have already been processed, every thread memorizes the list of completed chunks. Reassignment is done in two steps. First, all chunks are assigned as usual to the threads in a round-robin fashion. Since completed chunks are marked, they can be skipped later on. Step 1 potentially creates an unbalanced load since some threads receive more unprocessed chunks than others. Step 2 relocates unprocessed chunks from overloaded threads to undersupplied 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 threads only receive one uncomputed chunk each. To improve load-balancing, step 2 moves two chunks to undersupplied 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
double sproduct(double[] a, double[] b) {
    double s = 0.0;
    //omp parallel for reduction(+:s)
    for (int i = 0; i < a.length; i++) {
        //omp adjust
        s += a[i] * b[i];
    }
    return s;
}

Figure 6. OpenMP example to compute the scalar product of two vectors.

terminated inside the loop body stores its partial results in the partial block descriptor. When another thread takes on this partial chunk, it can no longer simply copy all the live variables from the partial block descriptor. Instead of overwriting it, the reduction value has to be merged with the thread’s partial result.

Fig. 6 shows an example of an OpenMP-parallel calculation of the scalar product between two vectors a and b. At the adjustment point in line 6 the variables i, a, b, and s are live. Whereas i, a, and b are regular variables, s is a reduction variable that is reduced by the addition operator. When the currently executing thread is removed from the team, the live variables are stored in the partial block descriptor for later restoration when another thread takes on the work of the removed thread. The resuming thread copies the values of i, a, and b and overwrites its incarnation of these variables. However, s needs a special treatment. The resuming thread already computed a partial sum in its incarnation of s while it processed its chunks of the loop. Hence, when s is restored from the partial block descriptor, overwriting s would cause a loss of partial results. The solution is to sum up the thread’s existing result and the result that comes with the partial block descriptor, thus, effectively preserving both intermediate results.

3.4. Context for Added Threads

When a thread takes on a partial chunk, all live variables are contained therein. On the other hand, when new 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 threads. For that purpose, the set of live variables is stored by the master thread directly before entering the construct and a copy of it is loaded by the added threads.

In Fig. 7, a variable c of a class called Collector is created in the parallel region by every thread and is later 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 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.
void example() {
  //omp parallel
  { Collector c = ...;
    int z = ...;
    //omp for
    for (int i = ...) {
      //omp adjust
      compute(i, c, z);
    }
  }
}

Figure 7. Variable context at an adjustment point.

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. 8, parallelization is done by OpenMP, but work distribution is explicitly implemented by the programmer. The example code introduces an indirect data dependence from the thread ID to array[i]. Due 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 if these functions are used within reparallelizable regions. As OpenMP provides means to implement most applications with pure OpenMP constructs only, this is not a severe restriction. Fig. 9 shows the correctly parallelized version of Fig. 8.

b) If threads are added to or removed from a parallel region, the total number of times every statement of the region is executed can change. This might affect the correctness of the application if the number of executions is crucial to application semantics. For example, assume that a thread is removed at the barrier in Fig. 10. 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. The compiler then does not add implicit adjustment points and forbids manually placed adjustment points.

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 semantics. For example, when a variable is used to store the current thread ID (as shown in Fig. 8, line 4), the new thread would receive the ID of the master thread in that variable when the new thread’s context is
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 + sz; i++) {
        // computation using array[i]
    }
}

Figure 8. Dependence to the thread ID.

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

Figure 9. Corrected example of Fig. 8.

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

Figure 10. Limitations example.

loaded. The programmer has to be aware of this issue and deal with it accordingly. However, since in most OpenMP programs the explicit use of thread IDs 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. Another possible solution to migration is to migrate only one process at a time
(sometimes called live migration), i.e., continuously migrating the application to the new target without stopping computation. However, for two reasons, this method is not feasible. First, as most parallel applications communicate frequently between processes, this method would deteriorate application performance while migrating, as the communication has to cross the high-latency, low-bandwidth WAN connection. Hence, stopping the application effectively causes the same overall performance penalty as the live migration. Second, a continuous migration does not blend well with the current way of reserving computing time at a cluster. For a live migration, the old and new (possibly remote) reservations have to be active at the same time, which cannot be guaranteed in all cases; meta-schedulers that guarantee co-scheduling are still an open research topic [7, 14, 18]. Hence, only freezing the application state in a checkpoint provides a general solution to the inter-cluster migration problem.

Checkpointing saves the computational state of an application [15]. The state can then be transferred to another cluster, on which the application is resumed and continues computation. We have presented a compiler-based approach for migrating threads in heterogeneous clusters in [30]. It provides a means to checkpoint the computational state of a single thread, to move the state to another machine of a potentially different architecture, and to continue the computation there. For migration of OpenMP applications, we added a coordinated checkpointing algorithm for multi-process, multi-threaded 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 that run on a compute cluster.

4.1. Thread Checkpointing

A generic stack frame format to store the current stack of a thread is the basis of platform-independence in [30]. For each call-site of a function, the compiler creates so-called check pointers and uncheck pointers. Checkpointers map the architecture-specific stack frame of the function at that call-site to a generic, machine-independent format. In turn, uncheck pointers load a function’s stack frame from the generic format and recreate the function’s native stack frame.

The computational state at a given call-site is determined 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 stack-frame format consists of a set of tuples of the form (UDS, value). As described in [30], 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 preceding 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. 11. The details can be found in [30].

As an example of how to construct the UDS, let us consider a Java function (Fig. 12) 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. 11, 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
R1 “A = constant” → string(A) = “C:<constant>”.
R2 “A = B” → “string(A) = string(B)”.
R3 “A = call” → “string(A) = “call:<index of call in all calls inside the containing function>”.”
R4 “A = param(X)” → “string(A) = “P:<index of param(X)>”.”
R5 “A = object<access(expression, field)>” → “string(A) = “access:field” and recurse into expression.”
R6 “A = B op C” → “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 11. UDS construction rules.

Figure 12. Java checkpointing example.

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 checkpointing approach. A thread is checkpointed by sequentially calling the check pointers 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 [26], 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 check pointer does not necessarily need to wait for the disk to catch up. As soon as the checkpoint 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 threads but also execute several Java service threads such as the garbage collector and the finalizer thread that both are required by the Java runtime. There is also a set of Jackal-specific threads that are used to handle network traffic and to perform
monitoring tasks in the Jackal runtime system. 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. The coordinated algorithm does not impose a visible runtime overhead, as the time needed to run the coordinated protocol is negligible compared to the time needed to write the application’s heap to disk. If the node count becomes too large for a coordinated protocol it can later easily be replaced by a cheaper protocol.

A single-process application is checkpointed by saving both the state of all threads and the heap to disk. This is accomplished by collecting all threads in a special barrier (indicated by the gray horizontal line in Fig. 13(a)), 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 checkpointing algorithm and enters the barrier. Otherwise, it proceeds with the regular synchronization code. To enter a barrier in case no `Thread.sleep()` and the like is reached, our compiler adds calls to `createCheckpoint()` at certain code locations. For example, calls to the checkpointing function are added at the end of complex functions before returning and at the end of loop bodies. In the inner-most loop of a loop nest, the call is omitted, as adding a call would add too much overhead to the loop.

Threads that are already blocked, i.e., 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`. Catching the exception, the threads check for an active checkpoint and enter the local barrier if necessary. Upon leaving the barrier, 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. This ensures correct semantics of the blocking `wait` methods in Java even in the presence of our checkpointing algorithm.

Multi-process checkpointing is achieved by means of coordinated checkpointing [15]. To request a checkpoint, the master node first stops all local threads by setting up a barrier. It then sends a broadcast to all other nodes (Fig. 13(b), solid arrows). Because of the first-in first-out property of Jackal’s communication layer, this broadcast message pushes all data...
messages (indicated by dashed arrows) through the network to their respective receivers. As soon as another node receives a checkpoint request message, it sets up a local barrier as well and informs its blocked threads about the checkpoint request. All other 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 (Fig. 13(c)). Then, all local threads start to write the checkpoint. The barriers are torn down as soon as the stacks of the threads and the local heap have been completely handed off to the service thread. Hence, the application automatically continues with computation after the checkpoint is completed; no additional protocol is needed.

Our algorithm implements a race-free stop-the-world approach. Since all threads are stopped by barriers, no modifications occur to their respective stacks. The processes’ heaps also remain unchanged during a checkpoint for the same reason. As the network is cleared of in-transit data messages that might render a checkpoint invalid because of lost messages and orphaned messages [15], no rollback action is necessary at the time the application resumes from the checkpoint.

4.3. Interaction with Reparallelization

When the application resumes from a checkpoint, the number of nodes that execute the program can change. Hence, the application has to adapt to the new number of available CPUs and has to be reparallelized accordingly. 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, that is, a process that only executes the Java-specific service threads and the internal Jackal threads.

The checkpointing system interacts with the reparallelization runtime by means of the observer pattern [12]. The reparallelization runtime registers an observer that is notified when the application resumes from the checkpoint. The observer is passed both the new number of nodes/CPUs and the process mapping at restart time. 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. To avoid extensive and unnecessary data migration, the reparallelization runtime layouts the threads in a way that the existing threads are kept close to the nodes they were initially spawned in. Unavoidable data migration is automatically handled by Jackal’s DSM runtime. The DSM runtime determines which nodes continuously access data and migrates the data to them. However, data redistribution causes a singular drop of performance until the data distribution has become stable again (see Section 5.1).

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 increase of runtime of each benchmark to the same code without adjustment points and/or checkpointing.

5.1. Benchmarks

For our evaluation we implemented a compute intensive Lattice-Boltzmann Method (LBM) [31] benchmark. In addition, we use JOMP’s OpenMP/Java port of the JGF benchmarks [25, 6]. We skip section 1 of the benchmark suite since it solely contains micro-benchmarks for individual OpenMP directives, such as the creation of parallel regions. From sections 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 uses cellular automata to simulate fluids. Space and time are discretized and normalized. In our case, LBM operates on a 3D domain divided into 120×120×120 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 10,000×10,000 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 800×800 pixels. 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 a 1D array.

5.2. Overheads

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

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

Inserting adjustment points into the SOR kernel decreases performance by up to 15.9% (see Table I). 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 2–3% can be observed. This speedup is caused by 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 threads.

Fig. 14 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 about 2 occurs when the number of threads is halved (middle). This closely matches the speedup behavior of LBM (on the right). The runtime peak 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 or 260 MB) 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 memory access latencies of the NUMA implementation. A NUMA implementation that allows for the migration of data is desirable to avoid a durable performance penalty after reparallelization because of high communication costs due to wrong data placement.
A similar result is achieved for SOR. As Fig. 15 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 100 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 speedup 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 face a slow-down of 1.9 when the thread count is halved. The runtime peak for Crypt and Raytracer is lower, as the data set that needs to be redistributed is smaller.

5.4. Migration

We demonstrate the feasibility of our migration approach by manually migrating LBM from the cluster at Erlangen, Germany, to a cluster at the Vrije Universiteit in Amsterdam, the Netherlands, and back. The cluster in Amsterdam is equipped with dual Intel Pentium 3
CPUs with 1 GHz, 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. At the time the migration has to be triggered, a checkpoint was created automatically. We then interrupted LBM by sending a signal from a monitoring shell script. The checkpoint file was copied to the target system and a new job reservation was manually submitted to it. On the way back, the same actions were performed.

Fig. 16 shows the runtime per LBM time step. After the 16th iteration, we have interrupted and manually migrated LBM from Erlangen to Amsterdam. The number of CPUs was 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. Transferring the checkpoint of LBM between the clusters roughly took 50 seconds. The bandwidth for transferring the 128 MB checkpoint was roughly 2.2 MB/sec. The total queue time in the cluster queues was about five minutes.

6. Future Work

In this paper, we have presented an approach to automatically reparallelize an OpenMP application and to checkpoint a Java application. Although our checkpointing solution is automatic in a sense that reparallelization and checkpointing can be handled without the programmer’s help, the migration was performed manually.

Our current research focuses on the migration strategy to overcome the need to manually interrupt the application and to then copy the checkpointing file. We investigate a centralized bidding system, in which each accessible cluster runs a light-weight daemon that continuously monitors the load on the local system. In regular intervals, the daemon reports a bid that indicates what compute power the cluster can deliver at what time in its future execution plan. Based on the incoming bids, the Jackal runtime system selects the most suitable cluster.
as the target for migration, i.e., the one that delivers the highest computing power at the earliest time. This repeats until the application has finished.

As most clusters are hidden behind firewalls or are located in private networks, we are also working on establishing a virtual network that spans several grid sites. The virtual network will be organized as a loosely coupled peer-to-peer network in which accessible clusters may join as they become available to the user. We plan to use the virtual network as the main medium to transport the load information to the application. In addition, the checkpointing state will be transferred through the virtual network to eventually migrate the application to the target system.

7. Conclusion

We have presented a novel approach to reparallelize and to migrate OpenMP applications between clusters of different size and architecture. This helps to conceal the boundaries of individual clusters in a computational grid. 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. Although reparallelization is restricted to well-formed OpenMP programs and type-safe programming languages, it automatically adapts a variety of OpenMP programs to the new number of available processors.

Benchmarking shows that reparallelization imposes little overhead and scales as expected. When the number of threads is changed, the new parallelization achieves speedups that are comparable to the regular speedup 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


Copyright © 2000 John Wiley & Sons, Ltd.
Concurrency Computat.: Pract. Exper. 2000; 00:1–7
Prepared using cpeauth.cls


