An Introduction to Balder – An OpenMP Run-time Library for Clusters of SMPs

Sven Karlsson

Department of Microelectronics and Information Technology
Royal Institute of Technology, KTH, Sweden
Sven.Karlsson@sven.karlsson.name

Abstract. In this paper a run-time library, called Balder, for OpenMP 2.0 is presented. OpenMP 2.0 is an industry standard for programming shared memory machines. The run-time library presented can be used on SMPs and clusters of SMPs and it will provide a shared address space on a cluster. The functionality and design of the library is discussed as well as some features that are being worked on. The performance of the library is evaluated and is shown to be competitive when compared to a commercial compiler from Intel.

1 Introduction

OpenMP has during the last few years gained considerable acceptance as the shared memory programming model of choice. OpenMP is an industry standard and utilizes a fork-join programming model based on compiler directives [1, 2].

The directives are used by the programmer to instruct an OpenMP aware compiler to transform the program into a parallel program. In addition to the directives, OpenMP also specifies a number of run-time library functions.

To use OpenMP, an OpenMP aware compilation system is thus needed. The compilation system generally consists of a compiler and an OpenMP run-time library. The library is not only used to handle the run-time library functions as defined by the OpenMP specification but also to aid the compiler with a number of functions that efficiently spawn threads, synchronize threads, and help share work between threads.

In this paper, an open source OpenMP run-time library, called *Balder*, is presented which is capable of fully handling OpenMP 2.0 including nested parallelism [2]. The library supports not only single SMPs efficiently but also clusters of SMPs making it possible to do research on extensions to the OpenMP specification in the areas of SMP centric and cluster centric extensions. A compiler, called OdinMP, targeting the library is already readily available [3, 4]. A more detailed description of the OdinMP transformations is available on OdinMP's homepage [4]. I will not discuss the API of Balder in detail in this paper and instead I refer the interested reader to the aforementioned website. The source code of the library is also available from the same website.

The library is highly portable and there are currently ports available to several processor architectures such as x86 and ARM and to several different operating systems including Unix variants and Windows versions. Balder is currently at version 1.0.1.

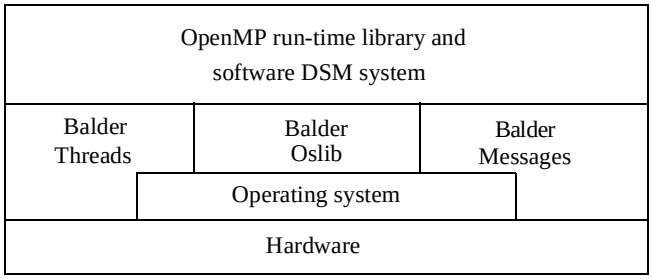

Fig. 1. Overview of the design of the Balder library

The rest of the paper is organized as follows. Section 2 provides a brief overview of the library while section 3 provides information of the sub-libraries that Balder builds on. Section 4 describes the OpenMP run-time itself and presents some details on the implementation of important primitives while section 5 discusses work-in-progress and future features for the Balder run-time. Experimental results are presented in section 6. The paper is summarized in section 7.

2 Overview of the Run-time Library

The run-time library is implemented in ANSI C [5] and provides the following functionality:

- Full implementation of all OpenMP 2.0 intrinsic functions, i.e., the run-time library functions described by the specification.
- Efficient handling of threads including thread creation, and thread synchronization.
- Parallel for-loop primitives to aid the compiler when transforming work sharing constructs.
- Support for OpenMP's threadprivate variables.
- Support for OpenMP's copyprivate clause.
- A built-in *software DSM*, software distributed shared memory, system to achieve a shared address space on a cluster.
- Memory management for the shared address space.
- Support for shared stack storage.

The library builds on previous experience [3, 6]. It is designed to be highly portable and to achieve portability it is designed as a layered system. Most of the functionality outlined in a previous paper is implemented [7].

The library utilizes three sub libraries: *Balder Threads*, *Balder Oslib*, and *Balder Messages* as can be seen in figure 1. These libraries provide thread services, operating system services, and cluster messaging services respectively.

The software DSM system and OpenMP run-time library is then implemented on top of the sub-libraries making it possible to implement the run-time in ordinary ANSI C without any dependencies on processor architecture or operating system features. In essence, this means that porting Balder is a matter of porting the well-defined primitives in the three sub-libraries.

2.1 Related Work

Balder is complete re-write of the run-time described in a previous paper which only supported uni-processor nodes [6]. No code has been borrowed or taken from that prototype or other OpenMP run-time libraries. Balder differs from previous efforts in that it is designed from the ground up to be both efficient and portable. In a previous paper, the Balder run-time was mentioned with the OdinMP compiler [3]. In that paper the focus was on the compiler and the design of Balder was not discussed.

In the next section, I will continue with a description of the run-time sub-libraries.

3 The Balder Sub-libraries

The three sub-libraries are instrumental in achieving a high degree of functionality and portability. A layered design approach has, whenever possible, been used even in the sub-libraries to aid porting efforts.

3.1 Balder Messages

Balder Messages is a packet-based, network technology independent messaging library with support for prioritized communication. The library is described in previous work [7, 8] and I will here only provide a very brief introduction. The transmission and reception primitives in the library use a data format based on linked lists for the packets. This facilitates scatter-gather I/O. It is also possible to allocate memory for packets in network hardware buffers so as to achieve zero-copy communication. Finally, there exist primitives for both synchronous and asynchronous communication as well as active message communication [9].

The library is divided into several parts. One part handles the construction of packets as linked lists, another handles flow control and queuing of packets and the last is the network back-end. There exist network back-ends for UDP and MPI [10]. Back-ends with partial support for Myrinet, LAPI, and System V's shared memory primitives exist [11].

3.2 Balder Oslib

Balder relies heavily on the virtual memory management system of the operating system. The virtual memory systems of operating systems differ from each other in both APIs and provided services. The Balder Oslib sub-library acts as a virtualization layer and provides an operating system independent API. There is support for:

- Creation of a virtual memory region which is guaranteed to be at the same address range on all cluster nodes.
- Setting of access permissions on individual virtual memory pages.
- Catching of page-faults and similar exceptions.
- Timers and timing management.

In addition, Balder Oslib implements a registry under which arbitrary data structures can be filed using strings as keys. This registry is used to store global system options and data structures and is instrumental in making it possible to modulize Balder properly without being hampered by artificial cross-module dependencies.

The Oslib is the sub-library which is the least layered. This is largely due to the virtual memory systems differing so much between operating systems. The Windows port, for instance, requires a completely different implementation than Unixes and cannot share any code with other ports.

3.3 Balder Threads

Balder Threads provides an efficient processor architecture independent multithreading API [12]. Balder Threads has primitives for:

- Thread creation and destruction.
- Thread synchronization using monitors, monitor signals, and barriers.
- Work queues.
- Stack frame creation so that arbitrary functions can be called.

Balder Threads uses a system of assembly macros that describes the processor architecture. A port to a new processor architecture is in most cases a matter of adapting the macros of an existing port.

The assembly macros describe how a function stack frame is organized, how functions are called, e.g. which parameters are passed on the stack and which are passed in registers, and provide implementations for important low-level primitives. Balder Thread uses POSIX threads as underlying thread library but implements efficient synchronization primitives using the mentioned low-level primitives [13]. The primitives used by Balder Threads are test-and-set, fetch-and-add, and a memory fence operation. Most modern processor architectures require that memory fence operations are embedded into lock and barrier operations or they will otherwise not work. This is not the only reason for providing memory fences as a low-level primitive. The memory fence operations are also used to implement the support for the OpenMP flush directive. In the absence of a required primitive, in the form of macros, Balder Threads will first revert to an implementation which only uses test-and-set and then, if no test-and-set primitive is provided, to POSIX threads. The memory fence operation is a no-operation on processor architectures that do not require memory fences.

Using this scheme of portable synchronization primitives based on low-level assembly macros, Balder Threads is able to achieve a synchronization overhead close to an order of magnitude less than POSIX threads. The synchronization primitives are generally based on test-test-and-set with a time-out so as to avoid excessive busy wait.

Balder Threads only provides primitives which can be used by threads running on the same cluster node. The OpenMP run-time layer provides primitives which can be used across cluster nodes.

4 Software DSM System and OpenMP Run-time Library

The OpenMP run-time library is built on-top of the sub-libraries and can thus be written completely architecture independent. It provides support for a shared address space on clusters via a software DSM system, and support for high-level OpenMP primitives as outlined in section 2.

The software DSM system uses the virtual memory management system to provide, on a cluster, a shared address space which acts as a shared memory. The system does not rely on any particular hardware except a decent virtual memory system and a network interconnect system.

In Balder, the software DSM system is built on-top of Balder Oslib and Balder Messages. The software DSM system uses home-based lazy release consistency, HLRC [14]. The reason for using HLRC is HLRC's robustness and relative simplicity. Some special features

of the HLRC variant used in Balder will be mentioned when discussing the support functions for OpenMP.

As mentioned in section 2, the OpenMP run-time library provides a number of primitives in different areas of functionality. I will now go through these areas and provide an overview of the functionality provided and how the implementation is done.

4.1 Parallel Regions

OpenMP is a fork-join programming model and the basis of parallelism are parallel regions as defined by compiler directives. The compiler will transform each parallel region into a function which is then executed in parallel by a number of threads. The compiler then inserts a call to a run-time library function, which handles creation of threads and execution of the function, into the program. The run-time library function is the basis for parallelism and is called in__tone_spawnparallel.

Balder uses a pool of threads to avoid excessive and time consuming thread creation and destruction operations. Threads are never destroyed and are only created when no threads are left in the pool. All threads in the thread pool wait on a work queue as provided by Balder Threads.

Internally the implementation of in__tone_spawnparallel is straight-forward. It merely performs book-keeping, adds work to the thread pool's work queue, the calling thread executes the parallel region, and then waits for any spawned threads to finish executing the parallel region. After finishing, the spawned threads returns to the thread pool and can be reused in other parallel regions.

Each cluster node has its own thread pool and so message passing is used to hand out work when running on a cluster. The cluster code also make use of a limitation to simplify the message passing and the memory coherence protocol. Balder does not currently allow nested parallelism when running on a cluster although different approaches to lift the limitation are being investigated. All nested parallel regions are serialized yielding one level of parallelism. This is allowed by the OpenMP specification and does not break OpenMP compliance. Balder does, however, fully support nested parallelism when running on one single SMP node.

4.2 Lock Functions

The OpenMP specification describes a number of lock functions as run-time library functions. To handle these function, a set of distributed lock primitives have been implemented on top of Balder Threads and Balder Messages. The lock primitives are un-fair in the sense that if a lock is held on a cluster node, the threads on that node get precedence over other threads when setting the lock. This reduces the network activity and the latency in case of lock contention [15].

Intra-node synchronization is performed with Balder Thread primitives while inter-node synchronization is performed with message passing.

If the system is running on one single node, the lock primitives revert back to the efficient primitives in the Balder Thread sub-library, thus avoiding the overhead of the distributed lock algorithm.

4.3 Barriers

The threads executing a specific parallel region form a thread team. The threads in a thread team can be synchronized with barriers and the OpenMP run-time provides a function for barrier synchronization.

Inter-node barriers are performed in two phases. First there is an intra-node barrier as provided by Balder Threads. Next, when all threads internal to a node are synchronized, message passing takes place which synchronizes all the nodes to each other using a centralized barrier algorithm. No threads are allowed to proceed until the second phase is finished.

The barrier operation is an operation with particularly high overhead on clusters. The overhead, is however, overlapped with memory coherence operations so as to not waste CPU resources. No message passing or memory coherence activities are performed when only one cluster node is used.

4.4 Worksharing Primitives

The worksharing constructs in OpenMP can all be mapped onto parallel for-loops and so Balder only provides support for such loops. The primitives essentially split a range of iterations into smaller pieces which then are executed in parallel by different threads.

It is, when running on one single cluster node, straight-forward to implement such primitives and in Balder the fetch-and-add primitive as provided by the Balder Thread assembly macros is used, if available, to minimize thread synchronization.

A cluster implementation is, however, much more complicated and easily leads to excessive message passing. A trick is used to reduce the message passing. The entire iteration space is first divided statically among the cluster nodes so that each cluster node receives a piece of the iteration space proportional to the number of threads, taking part of the parallel for-loop, that are executing on the node. These smaller pieces are then divided and handed out to the threads. This way no message passing is needed to implement parallel for-loops at the expense of potentially worse load imbalance. Load balancing algorithms, to reduce any load imbalance, are planned but not implemented

This trick is OpenMP compliant as the static assignment of iterations performed follows all rules for the scheduling of for-loops stated in the OpenMP specification.

4.5 Advanced Data Clauses

The OpenMP specification defines several data clauses which describe how different variables are handled. One such data clause is the threadprivate data clause which defines a variable to be private to a thread and keep its value between parallel regions. Such variables are called *threadprivate variables*. This essentially means that the threadprivate variables cannot be stored on the stack but must be associated with the threads themselves.

The run-time provides, for this, a thread-local storage space and two primitives to access the storage space. One primitive is used to allocate and initialize space and another is used to retrieve a pointer to the space.

The compiler generates code which at startup defines the thread-local storage. The storage space is, however, not allocated or initialized until it is used.

All accesses to threadprivate variables are changed by the compiler so that the pointer to the storage space is retrieved using the second primitive and all accesses are performed relative to that pointer. Allocation and initialization of the storage space will be performed in the primitive before the pointer is returned if no space has been allocated for the thread.

Another data clause is the *copyprivate* data clause. It makes it possible to broadcast the value of a variable private to a thread to the other threads in the same thread team. The copyprivate clause has been added in the second revision of the OpenMP specification to aid programming of applications utilizing nested parallelism.

A set of primitives has been added to the run-time library to implement the copyprivate data clause. These primitives make it possible for the broadcasting thread to send a variable via the run-time library to receiving threads and then wait for the receiving threads to properly receive the variable. For efficiency, the primitives are devised so that several variables can be sent and received after each other.

Naturally, message passing is used for inter cluster node broadcasts.

4.6 Handling of Shared Memory

The software DSM system is providing a shared address space across the cluster which can be accessed as a shared memory. To manage this address space, the run-time library provides memory allocation and de-allocation functions similar to ANSI C's malloc and free [5]. These are used instead of malloc and free when running on a cluster. The software DSM system is inactive when running on a single node, i.e., a single SMP, and so the normal heap as provided by the operation system is used.

A few advanced code transformations are required to run OpenMP applications on clusters. The transformations involve the handling of shared global variables and shared stack variables. The OdinMP compiler has a command line option which if enabled forces the compiler to perform these transformations. The default is to not perform the transformations as they are not needed when generating code for SMPs.

Shared global variables must be allocated in the shared address space when running on a cluster. The OdinMP compiler can emit code to do the allocation and it also transforms declarations of, and accesses to, shared variables so that shared variables are accessed through pointers pointing to allocated memory regions in the shared address space.

In OpenMP, variables located on a thread's stack can also be shared. This can occur if a parallel region is started. The thread that starts the parallel region, i.e., executes the in__tone_spawnparallel primitive, is called the master thread. Variables on the master thread's stack can be shared among the threads in the spawned thread team.

Balder handles this by implementing one single shared stack located in the shared address space. One stack is enough as, on clusters, only one level of parallelism is allowed which means that only the thread that executes the serial portions of the OpenMP application can spawn parallelism.

The OdinMP compiler can transform the application code to make use of the stack. It changes how variables are allocated so that the potentially shared variables are allocated on the shared stack and it inserts code that builds stack frames on, and removes them from, the shared stack upon entry and exit from functions respectively.

One shared variable is in the example in figure 2. The example is simplified for brevity. The transformed code is, albeit simplified, presented in figure 3.

The code inserted by OdinMP is very much similar to what a compiler emits at function entry and exit to build and remove stack frames. One thing differing is, however, the checks to see if the function is being called from a serial or parallel region of the code. This is necessary as the shared stack must only be used from a serial portion of the code. A frame pointer is inserted as a local variable and used for accessing the shared variable.

```
void f(void)
{
  int shared_variable;    /* The shared variable */

  shared_variable=5;      /* The shared variable is accessed. */
}
```

Fig. 2. An example with a shared variable

```
void f(void )
{
  struct in__tone_c_dt0
  {
     int shared_variable;
  };    /* The declaration of the stack frame which is used on the
           shared stack. */

        /* The shared stack must only be used in the serial portions of
           the code. The declaration below makes space on the thread's
           stack to be used in parallel regions. */
  struct in__tone_c_dt0 in__tone_c_dt0;
        /* The variable below is true when executing in a parallel
           region. in__tone_in_parallel() is a run-time library call
           that returns 1 iff the calling thread is executing in a
           parallel region. */
  const int in__tone_sdsm_in_parallel=in__tone_in_parallel();
        /* The declaration below is for the frame pointer. The frame
           pointer is set to either the shared stack or the stack frame
           on the thread's stack. */
  struct in__tone_c_dt0 * const in__tone_sdsm_i_framep
           =in__tone_sdsm_in_parallel ? &in__tone_c_dt0:
        /* The shared stack pointer is called in__tone_sdsm_stackptr.
           It is below subtracted to make space for a new frame. This
           piece of code will only be executed if executing in a serial
           portion of the code.*/
     (struct in__tone_c_dt0 * const ) (in__tone_sdsm_stackptr=
        in__tone_sdsm_stackptr-sizeof(struct in__tone_c_dt0 ));
        /* Below is the access to the shared_variable. It is now done
           through the frame pointer. */
  in__tone_sdsm_i_framep->in__tone_c_dt0.shared_variable=5;
        /* When leaving the function, either at the end of the function
           or with return, the shared stack pointer must be updated so
           as to remove the frame if executing in a serial portion of
           the code. The code below does that. */
  if (!in__tone_sdsm_in_parallel)
      in__tone_sdsm_stackptr=in__tone_sdsm_stackptr+
                             sizeof(struct in__tone_c_dt0 );
}
```

Fig. 3. Transformed example with shared variable

In OpenMP, the flush directive is used to enforce consistency. The flush directive is a memory fence operation that controls when memory updates are conveyed and when memory operations are performed.

The application programmer must, essentially, insert a flush directive before an access or update of any shared variable that could have been updated by another thread and a flush directive must be inserted after an update for the update to be conveyed to other threads. Most OpenMP directives such as the barrier directives have implied flush directives and this reduces the need to insert extra flush directives. The OpenMP directives are defined so that extra flush directives are only needed in some very special cases, e.g. when implementing thread synchronization not using the OpenMP synchronization primitives.

The semantics of the flush directive is implemented in Balder just like in an earlier prototype and a more in-depth description is available in [6]. A special distributed lock is used to implement the flush directive. The lock is not accessible from the OpenMP application but is internally handled by the run-time library just like any other OpenMP lock. A flush directive consists of a set of the lock followed by a release. Information on memory updates are piggy-backed onto the lock. When setting the lock, the received information is used to invalidate memory contents so as to drive coherency. The received update information is merged with information on locally performed memory updates and then sent with the lock when the lock is released to a remote node.

The implied flush directives can be and are in most cases optimized. The barrier directive, as an example, has an implied flush directive but the coherency information is piggy-backed on the barrier message passing, thus removing the use of the special lock mentioned above.

4.7 OpenMP Intrinsic Functions

All the OpenMP 2.0 intrinsic functions are implemented in Balder. The implementation is rather straight-forward and mainly involves inquiring or updating the internal state of the run-time library.

5 Advanced Features

Balder has a few planned features currently under implementation. Apart from prefetch and producer-push [16, 17], support for a fine-grain Software DSM system is under implementation as outlined in the next section.

5.1 A Compiler Supported Hybrid Fine-grain/Coarse-grain Software DSM System

The OdinMP compiler can gather information from the source code of OpenMP applications which can be used to further optimize the performance of said applications on Balder. This can be used to insert coherency checks into the compiler output so as to reduce the granularity of the coherency while still being able to fall back on a page-based system.

In short this means that whenever a shared variable can potentially be accessed the compiler needs to make sure that there is code inserted, prior to the access, which assures the shared variable is cached locally. The key point here is that only the shared variable itself, and not the virtual memory pages on which the variable is located, has to be locally cached. This reduces the latency of remotely requesting shared data and also reduces the average memory access latency of shared variables thus increasing the performance, i.e., reducing

the execution time, of applications running on the Software DSM system provided the overhead of the inserted coherency checks is small enough.

OdinMP is being augmented to insert the mentioned coherency checks and two primitives is being added to Balder. The coherency checks are based on a check-in/check-out scheme where a piece of shared address space is requested and then later returned. Prototypes for the two primitives are:

```
void *in__tone_sdsm_checkout_memory(
                    void* shared_address,
                    unsigned int length);
void in__tone_sdsm_commit_memory(
                    void* shared_address,
                    void* local_address,
                    unsigned int length);
```

The in__tone_sdsm_checkout_memory function is used to request an up-to-date version of a memory region. The memory region is defined by the parameters shared_address which is the shared address to the region and length which is the length in bytes of the region. The primitive returns a pointer to a copy of the memory region.

The in__tone_sdsm_commit_memory primitive is used to hand a previously requested copy back to the system and commit changes. It takes as parameters the shared address in shared_address, the pointer to the copy of the region in local_address, and the length, in bytes, of the region in length.

6 Experimental Results

Some experiments have been conducted so as to evaluate the performance of Balder. These experiments are not exhaustive but they do give an indication of Balder's performance. The cluster parts of Balder are not evaluated as they are being tested and are not ready to be evaluated yet.

A dual Pentium-III workstation running Linux version 2.4.25 was used as experimental platform. The processors were running at a clock rate of 1 GHz.

The EPCC micro-benchmark suite was used in the experiments [18]. The EPCC micro-benchmark suite is a set of benchmarks which measure the overhead of individual OpenMP constructs and thus also the run-time library. The benchmarks were compiled with OdinMP version 0.284.1 and GCC 3.3.4. The Balder library version 1.0.1 was used and was also compiled with GCC 3.3.4.

For comparison, the same set of benchmarks were compiled with the Intel C/C++ compiler version 8.0 and run on the experimental platform. The Intel compiler supports C/C++. The highest possible optimization level was used in both compilation systems. The Balder library cannot currently be compiled with the Intel compiler.

The overheads in microseconds for the most common OpenMP constructs as reported by the EPCC micro-benchmarks are summarized in table 3. The overheads are presented with their 95% confidence interval.

The overheads in the first three rows are for one single parallel region, parallel for-loop, and barrier respectively. The lock and unlock primitives row is the overhead of setting and then releasing a single lock once.

The overheads of the primitives in the Balder library are very competitive for barrier and lock synchronization. The overhead of parallel for loops are much higher for the Balder run-

Table 3. Overheads in microseconds of common OpenMP constructs

OpenMP Construct	Intel compiler	Balder with OdinMP
Parallel construct	1.43 +/- 0.11	2.91 +/- 0.15
For construct	0.79 +/- 0.17	2.93 +/- 0.30
Barrier construct	0.48 +/- 0.19	0.49 +/- 0.12
Lock and unlock primitives	0.48 +/- 0.33	0.47 +/- 0.12

time. The reason for this is the fact that OdinMP is a source-to-source compiler and cannot do aggressive optimizations of the parallel for-loops as the Intel compiler can as it is compiling to object code.

The overhead of parallel regions are also higher for the Balder run-time. One contributing factor is the transformation of parallel regions as outlined in section 4.1. The functions created out of the parallel regions by the OdinMP compiler can take arguments. These arguments are managed by the in__tone_spawnparallel run-time primitive. The arguments are copied once more than actually needed on an SMP during the handling of the arguments and the creation of stack frames performed in the run-time library. The extra copying performed is, however, necessary when running on clusters. I'm investigating to see if this can be improved in future versions of Balder.

The differences in overheads between code generated by the Intel compiler and the OdinMP/Balder combination is very small. The overheads are in the same range as found in a previous study [3]. It was found in the same study that small differences in overheads are unlikely to influence end performance of OpenMP applications. The overheads measured thus suggests that Balder paired with OdinMP should be very competitive to commercial compilation systems.

7 Summary

This paper provides an overview of the current status of the Balder, OpenMP run-time library. The organization of the library is presented and the functionality and design of the different modules are described. Some selected parts of the implementation are discussed. Planned future and work in progress for the Balder library and the OdinMP compiler is presented. Some experimental data is presented which shows the Balder library to be competitive when compared to a commercial OpenMP compiler.

Acknowledgements

The research in this thesis has been in part financially supported by the Swedish Research Council for Engineering Sciences under contract number, TFR 1999-376, and by the Swedish National Board for Industrial and Technical Development (NUTEK) under project number P855. It was also partially financed by the European Commission under contract number IST-1999-20252.

Nguyen-Thai Nguyen-Phan has implemented parts of the work-sharing, memory allocation and distributed lock primitives. He has been instrumental in testing the library. The

API used by Balder is a super-set of the API for OpenMP run-time libraries developed during the Intone project by the Intone project partners. Eduard Ayguadé, Marc González, and Xavier Martorell at UPC in Spain were particularly helpful in that effort.

References

1. OpenMP Architecture Review Board, *OpenMP specification*, C/C++ version 1.0, Oct. 1998
2. OpenMP Architecture Review Board, *OpenMP specification*, C/C++ version 2.0, Mar. 2002
3. S. Karlsson, and M. Brorsson, A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing, in *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), Nov. 2004*
4. S. Karlsson, *OdinMP homepage*, http://www.odinmp.com, retrieved on May 1st 2005
5. Information Technology Industry Council, *ISO/IEC 9899:1999 Programming languages - C second edition*, American National Standards Institute, 1999
6. S. Karlsson, S-W. Lee, and M. Brorsson, A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory, In *Proceedings of 9th International Conference on High Performance Computing (HiPC 2002*), Dec. 2002, pp. 195-206
7. S. Karlsson and M. Brorsson, An Infrastructure for Portable and Efficient Software DSM, In *Proceedings of 1st Workshop on Software Distributed Shared Memory (WSDSM '99)*, Rhodes, Greece, June 25, 1999, also available from Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden
8. S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, In *Journal on Cluster Computing*, 6 (2): 161-169, Apr. 2003
9. T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, Active Messages: a Mechanism for Integrated Communication and Computation, In *Proceedings of the 19th International Symposium on Computer Architecture*. Gold Coast, Qld., Australia. May 1992. pp. 256-66
10. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, Jun. 12, 1995
11. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. *Myrinet: A gigabit-per-second local area network.* IEEE Micro, 15(1):29--36, Feb. 1995
12. S. Karlsson, A portable and efficient thread library for OpenMP, in *Proceedings of EWOMP'2004*, Oct. 2004
13. IEEE, *IEEE std 1003.1-1996 POSIX part 1: System Application Programming Interface*, 1996
14. Y. Zhou, L. Iftode, and K. Li., Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. in *Proceedings of the 2nd Operating Systems Design and Implementation Symposium*, Oct. 1996
15. Z. Radovic, and E. Hagersten, Hierarchical Backoff Locks for Nonuniform Communication Architectures, in *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003
16. M. Karlsson and P. Stenström, Evaluation of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems, In *Journal of Parallel and Distributed Computing*, vol. 43, no. 7, Jul. 1997, pp.79-93
17. S. Karlsson and M. Brorsson, Producer-Push - a Protocol Enhancement to Page-based Software Distributed Shared Memory Systems, in *Proceedings of the 1999 International Conference on Parallel Processing (ICPP'99)*, Sep. 1999, pp. 291-300
18. J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sep. 1999, pp. 99-105