

Portable Support and Exploitation of Nested Parallelism in OpenMP

Panagiotis E. Hadjidoukas Laurent Amsaleg
IRISA/INRIA, Rennes, France
{phadjido,lamsaleg}@irisa.fr

Abstract

In this paper, we present an alternative implementation of the NANOS OpenMP runtime library (NthLib) that targets portability and efficient support of multiple levels of parallelism. We have implemented the runtime libraries of available open-source OpenMP compilers on top of NthLib, reducing thus their overheads and providing them with inherent support for nested parallelism. In addition, we present an experimental implementation of the workqueuing model and the parallelization of a data clustering algorithm using OpenMP directives. The asymmetry and non-determinism of this algorithm necessitate the exploitation of its nested loop-level parallelism. The experimental results on a SMP server with four processors demonstrate our efficient OpenMP runtime support.

1. Introduction

Although nested parallelism is allowed by OpenMP [12], most OpenMP implementations do not provide the necessary support and thus nested parallel regions are always executed by a single thread. The Intel C and Fortran OpenMP compilers and the Fujitsu PRIMEPOWER Fortran compiler provide this functionality. The default Omni compiler [14] supports a limited form of nested parallelism, however Omni/ST [16], an experimental version of Omni equipped with the StackThreads/MP library, implements efficiently nested irregular parallelism. The NANOS compiler [1] also supports nested regions and OpenMP extensions for processor groups. The runtime support of the NANOS Compiler is provided by an efficient user-level threads library (NthLib), which was initially developed in the context of the NANOS project [10].

The main contributions of this paper are the following:

- An alternative implementation of the NANOS runtime library is presented. Although it shares the

same architecture with the native implementation, it follows some different design decisions targeting portability and performance tricks related to unstructured multilevel parallelism.

- The runtime support of available open-source OpenMP compilation environments is implemented on top of NthLib without modifications to the compilers and the code they produce. The exploitation of user-level multithreading and some additional features introduced in NthLib reduce the OpenMP runtime overheads.
- Portable runtime support for nested parallelism is provided by a common research platform to OpenMP compilers.
- An experimental implementation of the OpenMP workqueuing model [15] on top of NthLib is also presented, as a proof of concept with respect to runtime support.
- Finally, the parallelization, using OpenMP directives, of a hierarchical data clustering algorithm that necessitates the exploitation of multilevel parallelism, is described.

The rest of this paper is organized as follows: Section 2 summarizes the design decisions and features of our implementation of NthLib. Section 3 describes the implementation of available open-source OpenMP libraries on top of NthLib. Section 4 presents our parallel data clustering algorithm. Experimental results are included in Section 5. Finally, Section 6 discusses our ongoing work.

2. Runtime Library

The NANOS runtime library (NthLib) [11] provides two different primitives to spawn parallelism, depending on the hierarchy level in which the application is working: the deepest level is generated with work descriptors, a data structure that contains pointers to a function and its arguments and results in work execution through a function call without

allowing further spawning of parallelism. Higher levels are generated using nanothreads, a different interface which provides threads with a stack. Nanothreads provide an address space for private variables and can be used at all levels. Their descriptor is an extended version of work descriptor and is always allocated in the stack of the associated user-level thread, using a single memory allocation call in case their recycling queue is empty. Nanothreads are submitted for execution in ready queues, while work descriptors are inserted in special locations in shared memory.

The alternative implementation of NthLib presented in this paper serves as a supplement of the native one. The main features that differ in our case are the significantly higher portability and the adoption of a lazy stack allocation policy. For the rest of this document the term NthLib will refer either to this alternative implementation or to some general features present at both cases, and any reference to native NthLib is mentioned explicitly.

The high portability stems from the use of the POSIX standard. Instead of using native kernel-threads, the virtual processors are system-scope POSIX threads. This feature improves NthLib's interoperability with third-party libraries and facilitates the co-existence of OpenMP and POSIX threads in the same program. In our case, user-level threads are provided to NthLib by the Underlying Threads Library (UthLib) [6], which replaces the assembly based QuickThreads package. Context-switching is implemented using `setjmp-longjmp` calls or `ucontext` operations. An underlying thread is actually a stack where the work, i.e. routine, of a nanothread descriptor is executed. Since UthLib supports thread routines with a single argument, nanothread routines are executed through an appropriate proxy routine that receives as argument the corresponding descriptor.

Another important difference is the adoption of a lazy stack allocation policy: the data structures for nanothreads and work descriptors are identical and the stack for a nanothread is allocated just before its execution. Initial motivation of this design decision was its minimal memory consumption and the easier management of parallelism on Software Distributed Shared Memory (SDSM) clusters [7, 13]. Since shared memory operations are very expensive on clusters, work descriptors are inserted in the runtime queues. In order not to harm the modularity of our implementation, we follow the same approach on shared-memory multiprocessors. Lazy stack allocation has allowed us to introduce *stack handoff*, an optimization that almost equalizes the runtime overheads between nanothreads and work descriptors.

According to peer-to-peer scheduling, a finished nanothread picks the next descriptor, creates an underlying thread/stack for that descriptor and switches to that. Normally, the underlying thread is recycled from an appropriate queue. Using stack handoff, a finished nanothread simply re-initializes the underlying thread by replacing its descriptor with the new one and jumps directly to the nanothread's execution proxy routine. By allocating the descriptors from the parent's stack, the activation of the recycling mechanism for descriptors is avoided too.

A critical performance issue in multithreading environments that still lacks portability is that of synchronization: our implementation uses the spin locks and the atomic primitives available in native NthLib. These locks can be alternatively mapped to POSIX Threads mutexes or spinlocks, and they are also used to protect the ready queues. This is also the default case in native NthLib, which optionally supports lock-free queues. Finally, we have introduced a hybrid implementation of barriers between nanothreads that combines busy-waiting and blocking, according to the number of threads and the level of parallelism they belong to. This issue is further discussed in the next section.

3. OpenMP Runtime Support

OpenMP Compilers and NthLib. Most open-source research OpenMP compilers translate C or/and Fortran programs with OpenMP directives into code suitable for compilation with a native compiler linked with their OpenMP runtime library, which usually supports only a single-level of parallelism. We have managed to provide the required runtime support by implementing a corresponding software layer on top of NthLib. Although this layer could be implemented from scratch, starting from the compiler-produced code, the source code availability allowed us to apply a straightforward porting of these OpenMP runtime libraries on top of NthLib.

Initially, we applied our approach to the runtime library of the Omni OpenMP compiler. The successful integration of NthLib in this compiler motivated us to continue with the runtime libraries of the OMPi [4], ORC [3] and Intone OpenMP compilers [8]. In this paper, we focus only on the first two compilers (Omni, OMPi). The implementation of the ORC and Intone OpenMP runtime libraries was straightforward since the former is similar to that of Omni's (version 1.4), while the latter was implemented in native NthLib, in the context of the Intone project. NthLib was used to provide runtime support, single-level though, to a

modified version of the NANOS compiler and to the OdinMP C compiler.

Omni supports both OpenMP C and Fortran77 and its latest version (1.6) provides two alternative implementations of the OpenMP runtime library for shared memory multiprocessors. The default library (libompc) is based on POSIX Threads while the second one (libompst) uses the StackThreads library. The former is the most widely used due to its portability while the latter supports nested parallelism but unfortunately lacks portability and modularity. The implementation of Omni's runtime library on top of NthLib is equivalent for both versions and combines their main features: portability, user-level multithreading and nested parallelism. Omni's locks have been mapped to NthLib's locks and its thread pooling has been replaced by the virtual processors of NthLib, while generated work is represented with nanothreads. NthLib's descriptors have been extended in order to include information used in Omni's library: for instance, the descriptor of a nanthread that spawns parallelism contains all the necessary parental information the spawned threads need access to. Another issue related to user-level threading and nested parallelism was the applicability of pure busy-waiting for synchronization. It was resolved by extending the barriers between nanothreads so as to operate with an arbitrary number of threads in a team and multiple levels of parallelism. This functionality is also important when a large number of threads is spawned. Native NthLib assumes always that dynamic parallelism is enabled and thus the number of spawned threads never exceeds that of available processors.

OMP*i* is a recent experimental C compiler for OpenMP and provides the first publicly available implementation of the version 2.0 of the standard. For this runtime library, which makes extensive use of the POSIX Threads API, we followed exactly the same approach as for Omni. Additionally, we implemented appropriate POSIX-like barriers and condition variables in NthLib. Finally, we introduced a slight modification into the compiler in order to generate code that uses the abstract data type for NthLib's locks instead of hard-coded POSIX Threads mutex declarations.

Management of Nested Parallelism. Due to its lightweight user-level threads and the lazy stack allocation policy, NthLib can support efficiently a large number of threads and multiple levels of parallelism. Our initial approach for handling nested parallelism within the implemented OpenMP runtime libraries is a variation of the all-to-all scheme. Specifically, we

distribute the descriptors spawned at the first level of parallelism to all processors, inserting them at the end of the ready queues. Moreover, we insert the descriptors of inner levels at the front of the local ready queue that belongs to the processor they were created on. In our implementation, an idle virtual processor still extracts descriptors from the front of its local ready queue but, in contrast with native NthLib, steals from the end of remote queues, favoring a virtual processor to exploit the potential data locality of inner levels of parallelism. Although Omni/ST allows an idle processor to issue random requests for work stealing, it requires the intervention of the remote virtual processor, which introduces runtime overhead and reduces portability. The latter feature is further reduced due to the dependence of Omni/ST on the StackThreads compiler, a patched version of GNU C. On the contrary, work stealing in NthLib is performed asynchronously and the access of the virtual processors to the remote queues is performed in a uniform way. Moreover, our software configuration allows the integration of any native compiler.

Workqueueing Model. The OpenMP workqueueing model is a flexible mechanism for specifying units of work that are not pre-computed at the start of the worksharing construct. To demonstrate the inherent runtime support of NthLib to this execution model, we have developed an experimental source-to-source translator that maps the parallel `taskq` and `task` pragmas to calls of the NANOS API. Although the translator is limited to programs where tasks are single function calls and no clauses are used with the directives, it is adequate for a wide variety of applications. According to [15], parallel `taskq` creates a single queue and spawns worker threads that execute the available tasks. Since NthLib maintains its own queues and workers, this pragma simply initializes the execution environment by adding one dependency to the current nanthread. For each task, the dependencies are increased by one and a nanthread descriptor is created and submitted for execution in the local ready queue. Finally, at the end of parallel `taskq`, the current nanthread suspends its execution, waiting the created tasks to finish. Our proposal for the management of nested parallelism is also applicable to nested parallel task queues.

4. Parallel Data-Clustering Algorithm

Data clustering is one of the fundamental techniques in scientific data analysis and data mining. The problem of clustering is to partition the data set into

```

1. find_nearest_neighbor(int i, int *idx, double *dist) {
2.   min_dist = +∞, min_idx = -1;
3.   for (j = 0; j < i; j++) {
4.     - if (entry j has been invalidated) continue;
5.     - if ((dist = compute_dist(i, j)) < min_dist)
6.       { min_dist = dist; min_idx = j };
7.   }
8.   *idx = min_idx; *dist = min_dist;
9. }
10.
11. update_nearest_neighbors(int pair_low, int pair_high) {
12.   for (i = pair_low; i < N; i++){
13.     - if (entry i has been invalidated) continue;
14.     - if (entry i had neighbor pair_low or pair_high)
15.       find_nearest_neighbor(i, &nnb[i].index, &nnb[i].dist);
16.     - else if (pair_high < i)
17.       if ((dist = compute_dist(pair_low, i)) < nnb[i].dist)
18.         { nnb[i].index = pair_high; nnb[i].dist = dist; }
19.   }
20. }

```

Figure 1. Pseudocode of the update phase in CURE

segments (called clusters) so that intra-cluster data are similar and inter-cluster data are dissimilar. In this section, we present a parallel implementation of CURE (Clustering Using REpresentatives) [5], a well-known and efficient hierarchical data clustering algorithm, using OpenMP. Despite its efficiency, the worst-case time complexity of CURE is $O(n^2 \log n)$, where n is the number of points to be clustered. The parallelization of the algorithm aims at enabling its direct use on very large data sets and, to the best of our knowledge, it is the first time that OpenMP has been applied to such an application.

PCURE (Parallel CURE) uses an array of records, holding information about the size, the centroid and the representative points of each cluster. According to the hierarchical algorithm, every data point is initially considered as a separate cluster with one representative, the point itself. At the initialization phase, the algorithm computes the closest cluster for each cluster. Next, it starts the clustering, merging the closest pair of clusters until only k clusters remain. When two clusters are merged, we store the information for the merged cluster in the entry of the first cluster and simply invalidate the second one. The algorithm also maintains per-cluster information about the index of the closest cluster and the minimum distance to it. To avoid duplication of this information the algorithm searches for the closest neighbor of a given cluster only in entries with a smaller index.

Figure 1 presents in pseudocode the most computation demanding routine of the clustering algorithm, which corresponds to the update of the nearest neighbors. Special features of this algorithm are its asymmetry and non-determinism, due to the way the nearest clusters are computed and the gradual decrease of valid entries (clusters). The efficiency of the parallel

implementation strongly depends on the even distribution of computations to the processors. This, however, is a challenging task due to the peculiarities of the algorithm. Our experiments confirmed that the update phase cannot scale efficiently if only one level of parallelism is exploited, regardless of the scheduling policy and the chunk size used. Therefore, we have also parallelized the loop in the `find_nearest_neighbor` routine.

5. Experimental Results

We performed our experiments on an Intel Pentium III system with four processors at 550 MHz, with 512KB cache memory and 1GB of main memory, running Linux (2.6.6) with the Native Posix Threads Library (NPTL). As OpenMP compilers we have used Omni 1.6 (libompc) and OMPi 0.8.1 and as native compiler GNU gcc 2.3.2.

Figure 2 presents the synchronization overheads for the parallel and for OpenMP constructs, measured using the EPCC microbenchmarks [2]. These two constructs are representative of the OpenMP overheads related to threading. We observe that the overhead of parallel is lower for both compilers when the runtime support is provided by NthLib. On the other hand, when the number of threads is less or equal to the number of processors, the for construct exhibits higher overhead for native OMPi, due to the blocking POSIX barrier. However, when the number of threads increases, which can also occur in the case of nested parallelism, the overhead of native Omni's busy-wait barrier is significantly higher.

Figure 3 depicts the performance speedups over the sequential version of four benchmarks (CG, FT, LU, MG) of the NAS Parallel Benchmarks suite [9], ported

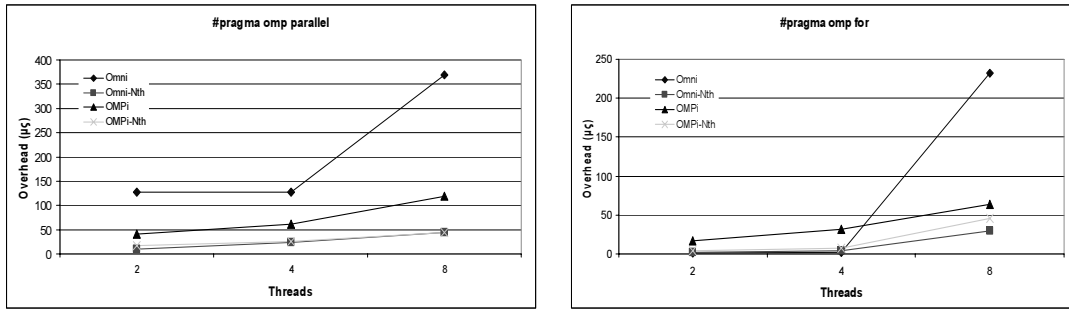


Figure 2. OpenMP overheads (parallel, for)

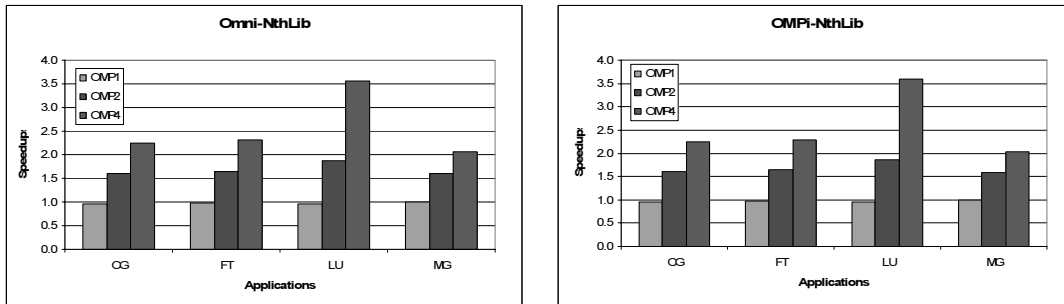


Figure 3. Performance of NAS benchmarks

to OpenMP C by the Omni group. Despite their different runtime overheads, all configurations result in very similar execution times. Therefore, results using the native runtime libraries are not included. Figure 4 shows the performance results of four benchmarks parallelized using the proposed workqueuing model. BlockLU and Raytrace come from the Splash benchmark suite [17], while FFT and Queens are two of the workqueuing benchmarks used in [15].

For our last experiment, we use PCURE as a benchmark in order to demonstrate our runtime support of nested parallelism and to provide a comparison with other OpenMP environments. For both loops of the update phase we use the static policy with chunk equal to 10. We used a small number of records (5000), of dimension 24, from a data set that contains image descriptors. Since these are organized in spherical clusters, we use a single representative per cluster. The clustering stops after 4000 steps. In Figure 5, we present the performance speedups obtained for the update phase of PCURE. The first two columns correspond to the use of the Omni and OMPI compilers with their runtime libraries built on top of NthLib. The third column (WQ) is an equivalent implementation of PCURE based on the workqueuing model. Finally the last two columns illustrate the performance speedups

when the Intel (version 8.0) and Omni compilers are used. Omni uses its Posix Threads based library and the size of the thread pool (OMPC_NUM_PROC) is double the number of requested threads (x 2 levels of parallelism). We observe that the application scales only when the runtime support is provided by NthLib. The workqueuing version performs slightly better because it avoids some explicit execution barriers. We also observe that the Intel and Omni compilers provide better results when 2 threads are used. In this case, the total number of threads that execute the two nested loops is equal to the number of physical processors. For 4 threads per loop, the speedup drops mostly due to the contention of the 8 total worker threads and the increased overheads of the runtime libraries.

6. Ongoing work

We are currently working on an alternative and portable implementation of the NANOS CPU Manager. This will allow us to apply the whole runtime infrastructure of the NANOS project to other OpenMP compilation systems. Regarding data clustering, our goal is to cluster very large data sets by running PCURE on clusters of SMPs.

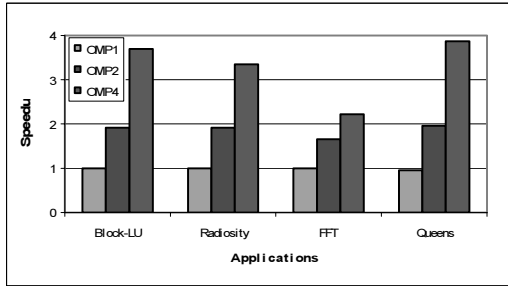


Figure 4. Performance of Workqueuing

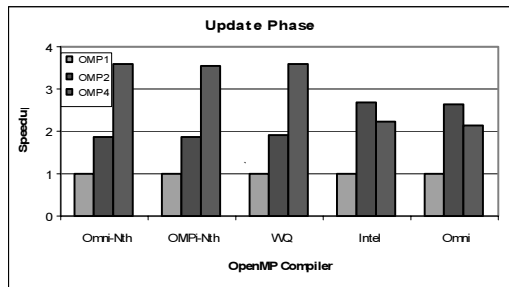


Figure 5. Performance of PCURE

Acknowledgments

Many thanks to all the partners in the NANOS and POP projects. We would also like to thank Dr. Clay Breshears for providing us the source code of the workqueuing examples.

References

- [1] E. Ayguade, M. Gonzalez, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*. Special issue on OpenMP, vol. 12, no. 12, pp.1205-1218, October 2000.
- [2] J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *Proc. of the European Workshop of OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.
- [3] Y. Chen, J. Li, S. Wang, and D. Wang. ORC-OpenMP: An OpenMP Compiler Based on ORC. In *Proc. of the Intl. Conf. of Computer Science (ICCS)*, Krakow, Poland, June, 2004.
- [4] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A portable C compiler for OpenMP V2.0. In *Proc. of the European Workshop on OpenMP (EWOMP '03)*, Aachen, Germany, September 2003.

[5] S. Guha, R. Rastogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large DataBases. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 1998.

[6] P.E. Hadjidoukas. UthLib: A Portable Non-Preemptive User-Level Threads Package. Technical Report HPCLAB-TR-230304, High Performance Information Systems Laboratory, University of Patras, March 2003.

[7] P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou. OpenMP Runtime Support for Clusters of Multiprocessors. In *Proc. of the Intl. Workshop on OpenMP Applications and Tools (WOMPAT '03)*, Toronto, Canada, June 2003.

[8] INTONE: Innovative Tools for Non Experts, IST/FET project (IST-1999-20252), <http://www.cepba.upc.es/intone/>.

[9] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. NASA Ames Research Center, Technical Report NAS-99-011, 1999.

[10] NANOS: Effective Integration of Fine-grain Parallelism Exploitation and Multiprogramming, ESPRIT Project No. 21097, <http://research.ac.upc.es/nanos/>.

[11] NANOS Consortium. M2.D2: n-RTL Implementation. NANOS project deliverable, April 1998.

[12] OpenMP Architecture Review Board. OpenMP Specifications. Available at <http://www.openmp.org>.

[13] POP: Performance Portability of OpenMP, IST/FET project (IST-2001-33071), <http://www.cepba.upc.es/pop>.

[14] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.

[15] E. Su, X. Tian, M. Girkar, H. Grant, S. Shah, and P. Peterson. Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures. In *Proc. of the 4th European Workshop on OpenMP*, Rome, Italy, September 2002.

[16] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Proc. of the Fifth Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR '00)*, Rochester, NY, USA, May 2000.

[17] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations, In *Proc. of the 22th Intl. Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.