Techniques Supporting `threadprivate` in OpenMP

Xavier Martorell, Marc Gonzàlez, Alex Duran,
Jairo Balart, Roger Ferrer, Eduard Ayguadé and Jesús Labarta
Barcelona Supercomputing Center - Computer Architecture Dept. - Technical Univ. of Catalunya
Campus Nord, Modul C6, c/Jordi Girona 1-3 08034 Barcelona Spain
{xavim, marc, aduran, jbalart, rferrer, eduard, jesus}@ac.upc.edu

Abstract

This paper presents the alternatives available to support `threadprivate` *data in OpenMP and evaluates them. We show how current compilation systems rely on custom techniques for implementing thread-local data. But in fact the ELF binary specification currently supports data sections that become threadprivate by default. ELF naming for such areas is Thread-Local Storage (TLS). Our experiments demonstrate that implementing threadprivate based on the TLS support is very easy, and more efficient. This proposal goes in the same line as the future implementation of OpenMP on the GNU compiler collection.*

In addition, our experience with the use of threadprivate in OpenMP applications shows that usually it is better to avoid it. This is because threadprivate variables reside in common blocks and they impede the compiler to fully optimize the code. So it is better to keep threadprivate as a temporary technique only to ease porting MPI codes to OpenMP.

1. Introduction and motivation

OpenMP [16] is a mature programming model for shared memory architectures, providing incremental parallelization over sequential codes. An OpenMP programmer usually starts from an initial sequential code, in C/C++ or Fortran and adds pragmas/directives to express the parallelism. A number of parallel constructs are offered, namely, parallel regions, loops, sections and combinations of them. Usually, it supports the exploitation of multiple levels of parallelism.

Nearly during the past decade, from the very beginning of OpenMP, we have been developing a threading library and a compilation infrastructure supporting this shared-memory programming model. NthLib [13] [14] [1] was designed to efficiently support multiple levels of parallelism, and as much fine grained as possible. The compilation infrastructure (Nanos Compiler [6]) was initially based on Parafrase-2 [17], and it is still used in the experiments for this paper. We are currently migrating to a new compilation infrastructure named Nanos Mercurium [2].

During all this time, there has been one OpenMP construction that has consistently resisted to be implemented in our environment. That was the *threadprivate* directive. Every attempt was giving too complex and ineficient implementations, or not general enough. We have also tried to make the work on the compiler side simple and let the run-time system to deal with most of the features. Also, initially, *threadprivate* constructions were rarely found in OpenMP applications. But this is not currently the case, specially since more and more applications written initially in MPI are being recoded using OpenMP. For this reason it was important to find a way to implement threadprivate in a general and efficient way.

Our current proposal is simple, both from the compiler and the runtime system perspective, mainly because it is based on a new feature provided by the Executable and Linking Format (ELF) specification, which allows to implement thread-local storage (TLS) efficiently.

The rest of the paper is organized as follows: Section 2 outlines the current support for the *threadprivate* directive found in several comercial compilers supporting OpenMP. This section also presents the related work. Section 3 presents our approach to implement *threadprivate* based on the TLS extension to ELF files. Section 4 shows the evaluation of our proposal with respect to not using the *threadprivate* directive at all. Finally, Section 5 concludes and presents future work.

2. Current support for *threadprivate*

Currently, OpenMP compilers support *threadprivate* variables by letting the runtime system to allocate a memory region for each thread. This is usually done during the initialization of the runtime system. The original variable is declared as a *common* area in the binary file. It is used as the place where the sequential code and the master thread access to find the values. The amount of

```
                 subroutine tpsimple (val)
                    integer val, tp_data
                    common /tp_area/ tp_data
              !$omp threadprivate (/tp_area/)
                    tp_data = val
                 end
```

```
  tpsimple:
      ...                        ; argument is r3 (pointer to val)
 or      r31,r3,r3                     ; put argument into r31
 bl       _xlGetThStorageBlock       ; get address of thread data
 or       r4,r3,r3                     ; into r4
 addis   r3,r0,$.TPK$.tp_area@ha   ; load address of common
 addi    r3,r3,$.TPK$.tp_area@l     ; block into r3
 bl       _xlGetThValue             ; get pointer to tp_data in r3
 lwz     r0,0(r31)                  ; do the real data copy
 stw     r0,0(r3)
...
```

Figure 1: (top) Fortran code, (bottom) assembly language code generation to access a threadprivate variable

memory to be allocated for each thread is determined from the size of such original variable.

The program code is then changed, so that every access to the *threadprivate* variable is done by getting a pointer to the thread area first and then applying an offset to access to actual value. The code transformation used in the IBM XL compilers is shown in Figure 1. This transformation must be done once for every different *threadprivate* common block accessed in a subroutine.

This way of implementing the *threadprivate* access has a large overhead compared to a usual variable access, residing either globally or in the stack. A simple microbenchmark consisting of the subroutine presented in Figure 2 has been developed to reveal this overhead. The subroutine (tps) does a single variable access for each invocation. This way, its execution time is heavily dependent on the way such an access is implemented, either as a regular variable or as a *threadprivate* variable. Accessing the variable as a *threadprivate* value has, in this particular situation, a cost which easily doubles the cost of accessing a variable in a regular common block. The effects of this overhead in the execution time are shown in Figure 3, using different operating systems, OpenMP compilers, and architectures. All executions are done with a single thread, so that only the overhead introduced by the different code generation is shown. In these tests, the subroutine shown in Figure 2 is executed 5 million times and the execution time is measured.

The experiments shown in this Figure are executed in a Linux platform based on Pentium Xeon [11] and the Linux Intel Fortran compiler 8.1 [10], a Linux platform based on POWER5 [8] and the IBM XLF 10.1 compiler [9], and a Solaris platform based on a ULTRA SPARC processor [22] and the SUNWspro 10 Studio [20]. All these environments use a similar technique for implementing *threadprivate* variables, as explained above in this section.

```
   subroutine tps (val)
     double precision val
     double precision num
     common /tp_area/ num
#ifdef TP
!$omp threadprivate (/tp_area/)
#endif

     num = num + val
   end
```

Figure 2: Subroutine optionally accessing a threadprivate variable or a variable in a regular common block

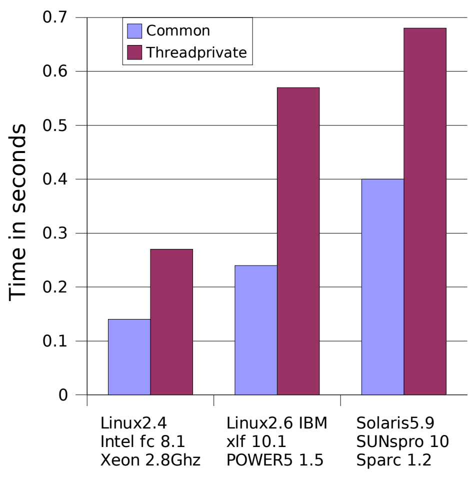

Figure 3: Overhead of accessing threadprivate data

As it can be observed, the results consistently show that the implementation of *threadprivate* variables in these three different architectures adds much more overhead with respect to an access to a common block. The execution time of the microbenchmark in each one of the architectures depends only on the processor speed. The benchmark is very small, it fits in the cache memory of the processors, so there are no memory effects. This is the reason why the Pentium architecture running at 2.8 Ghz. is the fastest executing this microbenchmark.

There exist some research OpenMP environments that use also similar techniques to implement the *threadprivate* directive. Among them, the Balder run-time system [12] is based on the OdinMP compiler to transform all accesses to *threadprivate* symbols towards an area previously allocated. This code transformation adds a call to the run-time system to determine a thread pointer to be used for such access. The Omni [19]compiler and runtime system also generates a code based on a runtime system call to get access to the threadprivate data.

From the perspective of a Fortran compiler, *threadprivate* values always reside in a common block. This is true even if the *threadprivate* directive is applied to individual variables, as it is allowed in OpenMP. This is to ensure that the values are kept across parallel regions. In C, they reside in a global variable. Fortran common blocks and C global variables (scalars, arrays, and structures) are the same from the perspective of the linker. This is the way C programs can access values in Fortran commons, and viceversa, allowing interoperability between these languages. So it is not a matter of the *threadprivate* directive, but a matter of optimizing access to common blocks and global variables. With variables residing in common blocks, the compilers are tied to the common block behaviour and they can not apply certain code optimizations: It is not possible to place a *threadprivate* scalar value in a processor register unless the compiler does interprocedural analysis: The code inside a function can cache the value in a register while it does not call another function. When a new call is reached, the value must be stored in its place in memory, so that the new function (possibly not available to the compiler, but to the linker) will get the initial value of the variable from the common block. *Threadprivate* simply makes more complex loading and storing the value from memory, increasing the overhead, as the previous experiment shows.

In the next section, we present an alternative implementation that reduces this overhead to a small fraction.

3. Implementing *threadprivate* using the ELF TLS specification

Starting in January 2002, the ELF binary file specification [3][23][18] includes the possibility to declare and use thread-local storage. Thread-local storage contains thread-local variables. A thread-local variable is implemented in such a way that each thread in the program can assign a different value to it. Each copy of the variable has a different address. It is the responsibility of the run-time system to allocate and manage the memory area to support the local storage of each thread. Programmers can easily define thread-local variables using the C language (non-standard) *__thread* attribute when declaring global or static variables [3]. Variables declared with this attribute are allocated by the compiler in two new object file sections: a `.tdata` section is used to place initialized TLS variables, and a `.tbss` section holds non-initialized variables. Any scalar, array or structure variable in C can be attributed as *__thread.*

While thinking in a way to implement the *threadprivate* directive for our OpenMP runtime system (Nanos), we thought that using this feature should provide several benefits:

- Ease the development of our prototype source-to-source compilers (NanosCompiler [6] and Mercurium [2]).
- Simplify code generation with respect to other techniques.
- Achieve better performance than current implementations.
- Achieve automatic interoperability between C /C++ and Fortran (assuming the compilers of these languages all use the same conventions for *threadprivate*).

In the next subsections, we describe the modifications introduced in the compiler and inside the run-time system.

3.1. Modifications to the compiler

The approach to achieve the previous benefits from the compiler was simple: every variable in a common block annotated as *threadprivate* should be attributed as *__thread* in the code generated by the compilers. This is what NanosCompiler and Mercurium do to translate the *threadprivate* directive. Any variable reference is kept the same. *__thread* variables will be allocated in the `.tbss` and `.tdata` sections and the ELF specification will take care of the rest.

This solution is fine in the case of C/C++: we can use native compilers supporting the *__thread* attribute in the target system to generate machine code (gcc supports it).

The problem appears in Fortran: even the Fortran specification 2003 [4] does not include any support for threading, so our compiler cannot generate anything to express thread-local storage. The Fortran 2005 specification , which is currently under discussion (and planned for 2008) comes with the solution. It includes a subset of Co-Array Fortran [15], in particular the possibility of expressing per-image variables, where each image can be a different thread running on the same address space, or a process running in a different node.

But meanwhile, we thought useful to look inside the gcc compiler [5] to see whether it could be easy to include some (partial) *threadprivate* support. At that time gcc 4.0 was already stable enough to be used with our benchmarks, so we selected it for examination.

It happens that gcc 4.0 includes gfortran, with good support for Fortran 95. As gfortran and gcc share the same backend for generating assembly code in each architecture, it seemed reasonable to modify it, including our proposed extensions.

The changes to gfortran were simple:

- Recognize a new token to express that a variable is thread-local.
- Mark the thread-local symbol with the same flag used in C.
- Allow the generation of common thread-local data. This was already supported by the compiler,

  but it was forbidden by a previous conditional statement.
- Add code generation support for `.tdata` and `.tbss` sections.

With these changes, gfortran generates common blocks into the `.tbss` and `.tdata` sections and the appropriate code to access them correctly. In addition, they are accessible from object files generated by GCC using variables with the same name and the *__thread* attribute. We know that the development of OpenMP in gfortran is going in the same direction [7].

With our current approach, the subroutine shown in Figure 2 is transformed in the intermediate and assembly code presented in Figure 4. The assembly code is presented in its Intel Pentium version.

In the Figure, we can observe that the code generated is much more simple compared to the code presented in Figure 1. In particular, as the thread has a dedicated register (gs) pointing to the TLS area, there is no need to call a function to obtain this pointer (_xlGetThStorageBlock in Figure 1).

3.2. Runtime system support

As our OpenMP runtime system (NthLib [1]) runs directly on top of the kernel-level abstraction offering the parallelism, it has been modified to create the kernel level entities with the TLS characteristics.

The usual case is the kernel-level thread, which can be a *thr* in Solaris, or a *clone* in Linux. In Solaris, just linking against the *librt* system library is enough to have available a memory region for each thread to act as the thread-local storage. The mechanism used is described in [21].

In the Linux implementations on Pentium and Power processors, we have modified NthLib to allocate a region of thread-local storage for each processor requested for OpenMP. This has been an easy task, as the address space of the parallel applications was already well organized. Figure 5 shows the resulting organization of the address space. We try to leave as much space as possible on the lower addresses (left part of the Figure) to the application code and data. We let the operating system map the dynamic libraries on its standard address. At the higher addresses (right hand side of the figure), we map the thread stacks and the thread-local storage (TLS) areas.

This is different from what we have seen in the Pthreads library in Linux, which maps the TLS areas inside the thread stacks. This has a strange effect for the programmer: the TLS size depends on the size of the pthread stacks, and not on the size of the actual `.tbss` and `.tdata` sections. This means that the pthreads library attribute *pthread_stack_size* is in fact controlling the size of both the stack and the TLS areas for each *pthread*.

```
SUBROUTINE tps (val)                        tps_:
 IMPLICIT  NONE                                push   %ebp
 DOUBLE PRECISION  val                         mov    $NUM_OFF, %edx
 DOUBLE PRECISION, TLS :: num                   mov    %esp, $ebp
 COMMON /tp_area/  num                         mov    8(%ebp), %eax
                                               fldl   (%eax)
  num = num + val                              faddl  %gs:(%edx)
END                                            fstpl  %gs:(%edx)
                                               pop    %ebp
                                               ret
```

Figure 4: Intermediate code (left) and assembly language (right) generated to access num as a threadprivate variable (gs is the thread register)

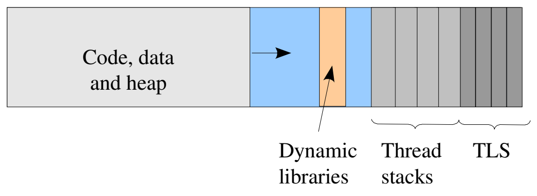

Figure 5: Structure of the virtual address space

After allocating the TLS areas and the stacks for each kernel thread, a simple machine depending code tells the operating system where is the TLS for each thread. The TLS ELF specification states that from now on, during execution, any TLS data should be accessed using a specific processor general purpose register. This register is the segment register *gs* in the Pentium architecture, and the *r13* general purpose register in the Power architecture. The OS loads the register and the execution continues normally.

With the previous modifications both in the compiler and the runtime system, it has also been possible to implement the *copyin* [16] clause for *threadprivate* variables. When *copyin* is specified, the master thread enters a barrier after spawning a parallel region, and waits while the slave threads access its TLS area to copy the values of the variables specified. After the copy, each thread joins the master in the barrier, and the parallel execution starts with all the threads having the same variable setup.

4. Evaluation

We have evaluated the TLS proposal for *threadprivate*, on one hand with the same microbenchmark used for the motivation, and on the other hand, using the NAS benchmarks written in Fortran which use the threadprivate directive (BT, SP, LU and CG). We have used Classes S and A of the NAS benchmarks to show how the technique influence the performance of applications with different working-set sizes.

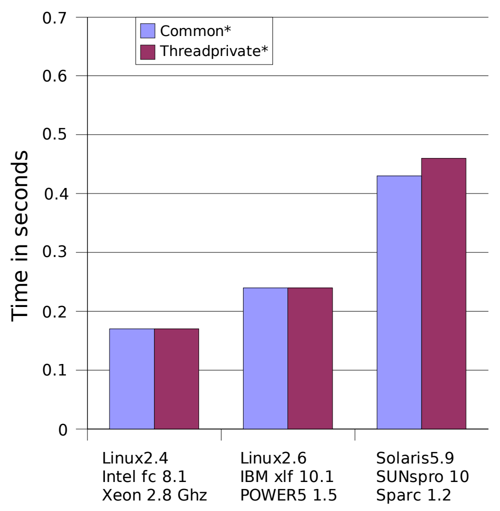

Figure 6: Performance comparison of the new threadprivate method

4.1. Evaluation on the microbenchmark

The results of the *threadprivate* microbenchmark generated using our modified gfortran compiler are presented in Figure 6. As the benchmark has been now compiled with gfortran, the results labeled *common** are a little bit different than the previous results presented, labeled *common* in Figure 3. The important observation here is that the performance to access *threadprivate* variables is now very similar to when accessing a regular common block.

4.2. Evaluation of the NAS benchmarks

We have taken the NAS benchmarks version 3.2, where four of the benchmarks written in Fortran use the *threadprivate* directive. The directive is used in different ways, so that different amount of data is privatized in a per-thread manner, depending on the benchmark. Table 1 Shows how BT and SP use *threadprivate* for two common structures (*work_1d* and *work_lhs*) containing around 10 different arrays. In addition, BT has 3 double precision scalar values on the *work_lhs* common structure. LU and CG use *threadprivate* to make some scalar values private. Those values are the amount of threads participating on the parallel region and the thread self identifier (obtained through the primitives *omp_get_num_threads()* and *omp_get_thread_num()*, respectively). In addition, CG has the *threadprivate* structure *urando* containing two double precision numbers to compute random numbers in a per-thread basis.

The evaluation of the benchmarks has been carried on a POWER5-based machine with 2 chips, giving a total of 8 hardware threads, running the Linux 2.6 operating system.

Benchmark	Scalars	Arrays
BT	*tmp1, tmp2, tmp3*	*work_1d, work_lhs*
SP	-	*work_1d, work_lhs*
LU	*threadinfo*	-
CG	*tinfo, urando*	-

Table 1: Usage of the *threadprivate* directive in the Fortran NAS3.2 benchmarks

Evaluation of the CLASS S version. To evaluate the proposal described in this paper, we decided to compare the performance obtained with the new implementation of *threadprivate* with the same application coded without the *threadprivate* directive. It is in fact easy to remove such directives from the source code in a way that the application is still running correctly and providing successful results: The arrays in BT and SP can be made just private, as there is no reuse of their values across parallel sections, so they must not be maintained.

In the case of the variables representing the number of threads and the thread identifier, in LU and CG, they are simply replaced by private variables that take the value of omp_get_num_threads() and omp_get_thread_num(), and that are assigned each time a parallel region using the variable starts. There are three parallel regions using this variables in LU and CG.

presents the evaluation of these benchmarks. It shows the megaflops per second obtained by the benchmark, depending on the number of hardware threads used and the technique used for privatization.

Initially, we started comparing the *threadprivate common* version with the *private-only* version. The three benchmarks SP, LU and CG had a similar performance, independently of the kind of privatization used. Observe that this means that accessing arrays in *threadprivate* with our technique offers similar performance to accessing private arrays located in the stack. On the other hand, the access to *threadprivate* scalar values is similar to getting their value using simple library calls like *omp_get_num_thread()*.

The BT benchmark was the exception: The performance of the *threadprivate* version was clearly lower, even when executed in a single processor. Observe in that the performance of *threadprivate* in bt.S with 1 thread is lower than 700 Mflops/s, while using private variables is 835 Mflops/s. After some thoughts and tests, we found that the problem was that mixing scalar and array variables in *threadprivate* commons was not a good idea. In particular, any variable residing in a common in Fortran has a *common association*, that may forbid some optimizations to the compiler. For this reason, we also present the intermediate version (labeled *threadprivate variables*), where the *threadprivate* directive is applied to the individual variables, instead of to the full common. In this case, the performance is closer to the *private-only* solution. The difference comes from the fact that a common for each *threadprivate* variable is still introduced in our code

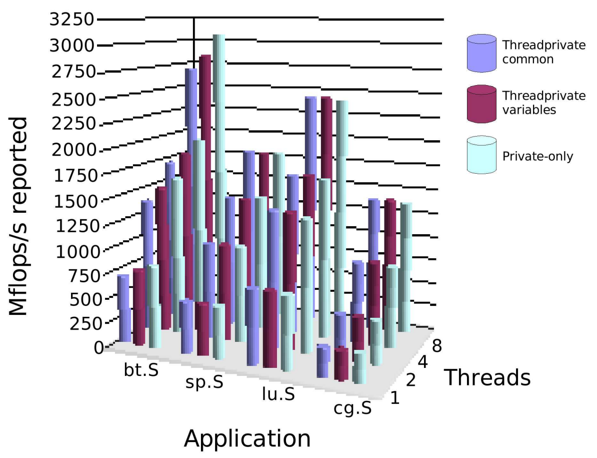

Figure 7: Evaluation of the NAS benchmarks (CLASS S)

transformation for such variables, while in the *private-only* solution they may be just allocated into a processor register.

Evaluation of NAS CLASS A. To check that the performance obtained in the CLASS S is maintained when the working-set of the benchmarks is increased, we present in Figure 8 the results for the CLASS A of the benchmarks.

In this Figure, we can observe that the behaviour of the benchmarks is very similar. BT reflects the same problem related to common association when making the common *threadprivate*. Still, forcing the disassociation of the variables in the intermediate approach (labeled *threadprivate variables*) is not so good as making them really private.

It is interesting to comment that the performance of all the benchmarks is reduced when executed in a small number of threads, compared to the class S version. The cause of this is that, as the working-set increases, the execution suffers from an increment in the number of cache misses. This effect is nevertheless amortized through the parallelization in BT, SP and LU: With 4 to 8 threads, these class A applications perform better than the class S ones. It is not the same with CG, in which in class A, the amount of sharing increases with the increase in the working-set and the performance is comparatively degraded, with respect to the class S.

5. Conclusions and Future Work

In this paper, we have studied the current techniques used by various OpenMP compilers to implement *threadprivate* variables. A first conclusion is that *threadprivate* must be used with care. Do not consider that a *threadprivate* variable can be as efficient as a private

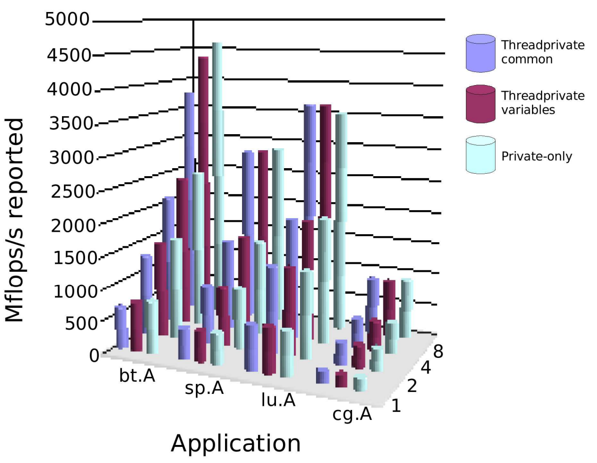

Figure 8: Evaluation of the NAS benchmarks (CLASS A)

variable. *Threadprivate* implies that variables must be placed in common blocks to be kept save across parallel regions. This may impede some optimizations, like keeping values in registers and the code will be slower. So use *threadprivate* only when the variables involved need to keep their values across functions and parallel regions.

In addition, we have proposed an implementation of *threadprivate* based on thread-local storage (TLS), a support introduced relatively recently in the ELF specification. We have implemented such proposal in the gfortran compiler and our OpenMP runtime system (Nanos). The evaluation of this proposal on a microbenchmark shows that the fine-grain overhead is highly reduced and the measurements are close to those obtained when accessing private variables. Also, it is shown that the performance obtained from the NAS benchmarks (Classes S and A) are comparable to the ones using strictly private variables. Nevertheless, it is also important to note that using *threadprivate* variables, and depending on the way they are finally placed on common blocks, there are applications (e.g. BT in our experiments) that suffer from the lower level of optimization achieved by the compiler.

Our current work is the porting of this feature to our new line of compilers, Nanos Mercurium, supporting both C and Fortran95, and achieve full interoperability between both languages. We will also maintain the modifications to gfortran across different versions of the compiler, while possible, with the goal to continue being able to work in a source-to-source way. In this line, adopting a syntax in accordance with Co-Array Fortran for our intermediate language seems the appropriate direction to follow to allow Fortran applications to be conscious about threads.
We would also like to continue with the evaluation of the proposal using other kinds of applications (e.g., integer, multimedia applications) to have a better understanding

with respect the differences of keeping the data private or making it *threadprivate*.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract CICYT-TIN2004-07739-C02-01.

References

[1] E. Ayguadé, M. Gonzàlez, X. Martorell and G. Jost, " Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications", International Parallel and Distributed Processing Symposium (IPDPS'2004). Santa Fe, NM (USA), April 2004.

[2] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé and J. Labarta, "Nanos Mercurium: A Research Compiler for OpenMP", European Workshop on OpenMP (EWOMP'04), pp. 103-109, Stockholm, Sweden, October 2004.

[3] U. Drepper, "Elf Handling for Thread-Local Storage", Version 0.20, Red Hat Inc., Feb. 8, 2003. http://people.redhat.com/drepper/tls.pdf

[4] International Fortran standards committee, "Draft International Standard (DIS) for Fortran 2003", ISO/IEC DIS 1539-1:2004(E), ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1601.pgf.gz

[5] R.M. Stallman and the GCC Developer Community, "Using the GNU Compiler Collection", Nov. 2005, http://gcc.gnu.org/onlinedocs/

[6] M. Gonzàlez, E. Ayguadé, X. Martorell and J. Labarta, "Exploiting Pipelined Executions in OpenMP", 32nd Annual International Conference on Parallel Processing (ICPP'03). Kaohsiung, Taiwan. October 2003.

[7] R. Henderson, "Re: [gomp] OpenMP IL design notes", Tue, 3 May 2005 14:16:35 -0700, "http://gcc.gnu.org/ml/gcc/2005-05/msg00109.html

[8] IBM Corporation, "Support for AIX 5L and Linux servers", Nov. 2005, http://www.ibm.com/servers/eserver/support/unixservers/

[9] IBM Corporation, "XLFortran", Nov. 2005, http://www-306.ibm.com/software/awdtools/fortran/xlfortran/

[10] Intel Corporation, "Intel Fortran Compiler 8.1 for Linux", Nov. 2005, http://www.intel.com/software/products/compilers

[11] Intel Corporation, "Intel Xeon Processors", http://support.intel.com/products/processor/xeon/index.html

[12] S. Karlsson, "An Introduction to Balder – An OpenMP Run-time Library for Clusters of SMPs", First International Workshop on OpenMP (IWOMP05), Eugene, Oregon USA, June 1-4, 2005.

[13] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, "Nano-Threads Library Design, Implementation and Evaluation", Technical Report UPC-DAC-1995-33. Dept. d'Arquitectura de Computadors-Universitat Politècnica de Catalunya, September 1995.

[14] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, "A Library Implementation of the Nano-Threads Programming Model", Euro-Par'96, Lyon (France). Lecture Notes in Computer Science, Springer-Verlag. Vol. 1124, pp. 644-649. August 1996.

[15] R.W. Numrich, J. Reid, "Co-arrays in the next Fortran Standard", ISO/IEC JTC1/SC22/WG5 N1642, May 2005.

[16] "OpenMP Application Program Interface", version 2.5, May 2005, http://www.openmp.org/

[17] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, and Scheduling Programs on Multiprocessors", International Journal of High Speed Computing, 1(1), 1989.

[18] Y. Sade, M. Sagiv, and R. Shaham, "Optimizing C Multithreaded Memory Management Using Thread-Local Storage", International Conference on Compiler Construction (CC05), Edinburgh, Scotland, April 4-8, 2005.

[19] M. Sato, S. Satoh, K. Kusano and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster", First European Workshop on OpenMP, Lund University, Sweden, October 1999.

[20] Sun Microsystems, "Sun Studio 10 for Solaris Platforms", Nov. 2005, http://developers.sun.com/prodtech/cc/documentation/ss10_docs/index.html

[21] Sun Microsystems, "Linker and Libraries Guide", Solaris 10 Software Developer Collection, 2005. http://docs.sun.com/app/docs/doc/817-1984

[22] Sun Microsystems, "UltraSPARC Processors", Nov. 2005, http://www.sun.com/processors/UltraSPARC-IV/

[23] Tool Interface Standards (TIS) Committee, "Executable and Linking Format (ELF) Specification", May 1995. http://www.x86.org/ftp/manuals/tools/elf.pdf