Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors

Xavier Martorell, Eduard Ayguadé, Nacho Navarro,
Julita Corbalán, Marc González and Jesús Labarta
Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
cr. Jordi Girona 1-3, Campus Nord, Mòdul C6, 08034 Barcelona, Spain
{xavim, eduard, nacho, juli, marc, jesus}@ac.upc.es

ABSTRACT

This paper presents some techniques for efficient thread forking and joining in parallel execution environments, taking into consideration the physical structure of NUMA machines and the support for multi-level parallelization and processor grouping. Two work generation schemes and one join mechanism are designed, implemented, evaluated and compared with the ones used in the IRIX MP library, an efficient implementation which supports a single level of parallelism.

Supporting multiple levels of parallelism is a current research goal, both in shared and distributed memory machines. Our proposals include a first work generation scheme (GWD, or global work descriptor) which supports multiple levels of parallelism, but not processor grouping. The second work generation scheme (LWD, or local work descriptor) has been designed to support multiple levels of parallelism and processor grouping. Processor grouping is needed to distribute processors among different parts of the computation and maintain the working set of each processor across different parallel constructs.

The mechanisms are evaluated using synthetic benchmarks, two SPEC95fp applications and one NAS application. The performance evaluation concludes that: i) the overhead of the proposed mechanisms is similar to the overhead of the existing ones when exploiting a single level of parallelism, and ii) a remarkable improvement in performance is obtained for applications that have multiple levels of parallelism. The comparison with the traditional single-level parallelism exploitation gives an improvement in the range of 30-65% for these applications.

1. INTRODUCTION

Several current multiprocessor machines are based on building blocks (modules) consisting of 2 to 8 processors along with 128 to 512 Mb. of local memory. For instance, the SGI Origin 2000 [1] and the SUN Enterprise 10000 [2] are built using this approach. By joining several blocks, current machines can scale to a large number of processors. Although in these machines the physical memory is globally shared, the access from a processor to a remote memory location can be 2 to 3 times slower than the access to a local memory location due to the non-uniform memory access (NUMA) architecture or memory contention.

When the access to remote memory is slow, the processor utilization is very influenced by the amount of data shared among the processors. When data sharing increases, the actual processor utilization decreases due to the search for data in remote memory. Even when there are no data sharing problems at application level, the way the parallel tasks are supplied to processors and the way the threads join at the end of parallel regions are the aspects that have to be carefully tuned in order to achieve good performance. These aspects become specially important when support for both fine-grain and multi-level parallelism are required.

Current multiprocessor systems provide parallel execution environments mostly targeted to loop-level parallelism that try to obtain good processor utilization when running parallel programs. Current trends in the development of parallel execution environments include the possibility of expressing and exploiting multiple-levels of parallelism. The OpenMP [3] proposal for Fortran and C/C++ is a good example, although by now, no implementation of OpenMP is able to exploit multiple levels of parallelism.

We are working in the design and implementation of a parallel execution environment (called nano-threads [4]) which allows the exploitation of multiple levels of parallelism, along with some mechanisms to group processors for the execution of parallel regions. The execution environment is targeted to Fortran applications containing both parallel sections and loops. In this paper, several techniques for supplying work to processors and to implement thread joining in NUMA machines are first compared. Our proposed techniques are aware of the cache behavior to minimize memory conflicts, although they may incur overheads higher than existing implementations that only support a single level of parallelism. We use the performance counters of the MIPS R10000 processor [5] to keep track of the cache behaviour in the different implementations. Then, two applications (the SPEC95fp Hydro2D and the NAS APPBT) are used to validate the proposed fork/join implementations. Both applications benefit from exploiting multiple levels of parallelism, achieving higher speedup than their corresponding single-level versions.

The rest of the paper is structured as follows: Section 2 summarizes the functionalities related to thread forking and joining supported by current parallelization environments. Section 3 describes the proposed fork/join techniques, enlightening the differences with the existing ones. Section 4 presents an evaluation

The final publication is available at ACM via
http://dx.doi.org/ 10.1145/305138.305206

of the overhead introduced by the proposed techniques. Section 5 shows the structure of the applications used to validate our proposals and Section 6 presents their evaluation. Finally, Section 7 concludes the paper and presents future work.

2. CURRENT PARALLELIZATION ENVIRONMENTS

Current parallelization environments are based on highly tuned and customized thread packages. Each package provides mechanisms to spawn and join parallelism. The implementation of such mechanisms greatly influences the type of parallelism which can be exploited at application level. Current implementations do not allow the exploitation of more than one level of parallelism, because the run-time execution environment forbids spawning parallelism when already running in a parallel region.

For instance, in the SGI MP library, when the master thread spawns parallelism in a parallel construct, it sets the starting program counter and the arguments for the slave threads and assigns a sequence number to the parallel construct. It collects all this information in a common and fixed memory area, known as the work descriptor. Then, the slave threads pick up their work from this descriptor. They all participate in the parallel construct identified by the current sequence number, executing the same function with the same arguments. This mechanism supports all the work-sharing constructs defined in OpenMP-like extensions to sequential languages such as Fortran or C/C++. However, it restricts the parallelism that can be exploited by the application to a single level because the descriptor cannot be reused till the previous parallelism has been joined.

The SGI MP library provides two different implementations for thread joining. The first one, used by default, is a global join structure in shared memory (SHM). The slave threads use the global sequence number assigned to the current parallel construct to mark in the global join structure, located in the work descriptor, that the parallel work has been finished. Meanwhile, the master thread, after participating in the spawned work, waits for all the sequence numbers in the join structure to reach the current one. Usually, the join structure is distributed along several cache lines in order to minimize false sharing when the slave threads access it. The second implementation replaces the join structure by a counter on uncached atomic memory (FOP's [6]) based on specialized hardware present in the memory modules of the Origin2000 system. The use of a shared counter or a single joining structure and a global sequence number to implement thread joining or barrier synchronization also disables multi-level parallelism exploitation because only one thread may act as the master thread for managing the joining process.

The parallelization environment based on the SUIF compiler [7] is also based in a run-time package which restricts the parallelism to a single level. The limitation of the SUIF run-time library is also motivated by the use of a shared work descriptor to supply work to the slave processors and only one join structure with a global sequence number to identify which parallel construct is active.

The Illinois-Intel Multithreading Library (IML [8]) is also targeted to shared memory multiprocessors. This one does support multiple levels of general, unstructured parallelism. Application tasks are inserted in task queues before execution, allowing several task descriptions to be active at the same time. IML focuses on the design alternatives for implementing such task queues (centralized and/or distributed). The library is also in charge of mapping the tasks to the available processors and of load balancing issues.

There are other research projects targeting at the exploitation of multiple levels of parallelism. They focus on providing some kind of coordination support to allow the interaction of a set of program modules in the framework of data parallel programs for distributed memory architectures. For instance, [9] propose a library-based approach that provides a set of functions for coupling multiple HPF tasks to form task-parallel computations. The Fx [10] and PARADIGM [11] projects propose extensions to integrate task and data parallelism in an HPF environment. The use of task parallelism is proposed to improve the performance when data parallelism is not enough.

Our thread package implementation (NthLib [12]) is built based on the Nano-Threads Programming Model [13][14]. It is targeted to shared memory and supports multiple levels of parallelism. The user expresses the multiple levels of parallelism through nested OpenMP directives. The NANOS compiler [4] (based on Parafrase-2 [15]) analyses the directives and generates a hierarchical task graph structure which represents the application. Each level of the hierarchy represents one possible level of parallelism. Each nested parallel construct makes the hierarchy one level deeper. The compiler generates code based on this internal representation. At run-time, tasks are mapped to nano-threads and these ones are dynamically mapped one-to-one to the currently available processors. The words thread and processor are used indistinctly along the paper.

The NthLib package interface is designed in such a way that it provides different primitives to spawn parallelism, depending on the hierarchy level in which the application is working. The deepest level is generated using the most efficient thread creation primitives, based on work descriptors and the techniques presented in Section 3. This level is the one which contains the finest grained parallelism. Higher (less deeper) levels are generated using nano-threads, a different (more costly) interface which provides threads with a stack. Local variables residing in the thread stack can be maintained (as in a local address space) to be used by the threads executing inner levels of parallelism. In addition, the latter interface also provides support for the general unstructured parallelism found in the hierarchical task graph. A detailed description of this interface and its implementation can be found elsewhere [12].

3. THREAD FORK/JOIN TECHNIQUES

The main goal of our proposals for implementing efficient thread fork/join in the nano-threads environment is the support for multiple levels of parallelism and processor grouping. In this section, two different techniques for supplying work to processors and an improved thread joining scheme are presented.

3.1. Forking Threads

Forking threads efficiently at the inner-most level of parallelism is based on supplying a work descriptor to the participating processors. The work descriptor consists of a pointer to the function encapsulating the work that has to be executed and its arguments. When the same work descriptor is supplied to a group of processors, each one decides the portion of work that has to execute, based on the arguments, the number of processors working in the group and its own identifier inside the group.

3.1.1 Functionality

The forking techniques are GWD (Global Work Descriptors) and LWD (Local Work Descriptors), WD for short. The GWD can be used in spawning the inner-most level in a multilevel parallel application. It supports multiple levels of parallelism because it allows the coexistence of multiple opened parallel constructs, solving the limitation of a single work descriptor found in previous implementations. All processors share a single GWD structure, they all can simultaneously supply work to the GWD and they execute the same work. GWD is expected to perform comparable to existing highly tuned implementations when exploiting a single level of parallelism. Figure 1A shows the execution of a parallel construct consisting of a parallel loop, four independent sections (also containing parallel loops) and another parallel loop. In this case, the master thread executing each section spawns the parallelism associated to the parallel loops inside the section to all the processors. Since the same descriptor is supplied to all the processors for each parallel loop, each processor in the system will execute a chunck of iterations of all the loops (probably always the same, if the compiler exploits loop affinity).

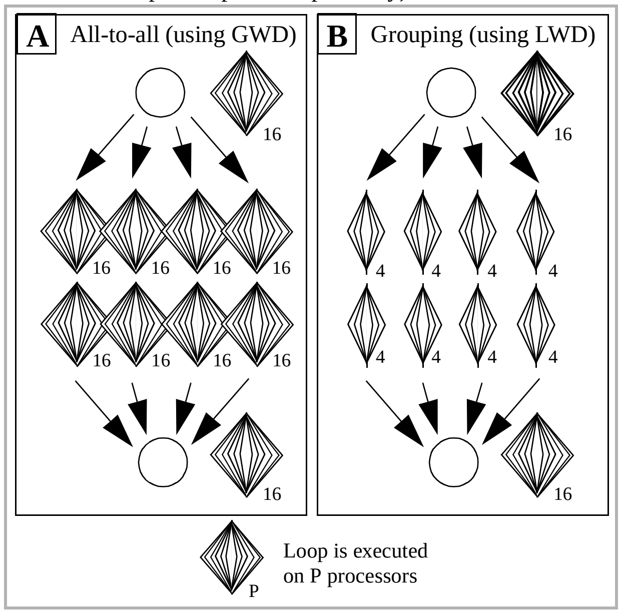


Figure 1: Two alternatives for the exploitation of multiple levels of parallelism

LWD is a more advanced technique that inherits most of the characteristics of the GWD and also allows processor grouping. Using processor grouping, the application is able to drive several processors to work on a independent task or set of tasks. Each group has a master and (possibly) several slave processors. The master processor starts the task and it is in charge of spawning the parallelism encountered inside it. After that, the slave processors cooperate with the master to execute the parallelism. At the end, the master processor waits for the slaves to complete the work. This execution model requires that any processor can supply work to any other processor. There exists one LWD structure for each processor, allowing different work to be supplied to different processors from different parts of the parallel application. In Figure 1B, the master thread executing each section spawns the parallelism associated to the parallel loops inside the section to just a subset of all the processors (in this case, each group of four processors is supplied with a different work descriptor). Due to the definition of these groups, now each processor executes a chunk of iterations for a subset of all the loops inside the parallel sections. However, for loops outside them, all processors cooperate in the execution of the parallel loops. Care should be taken into account about how this change in the structure of the parallelism can influence data locality.

As a result, the LWD overhead can be higher than that of GWD due to the individual supply of work descriptors, but it is expected that limiting the number of processors participating in an inner parallel construct makes worthwhile the exploitation of multiple levels of parallelism.

3.1.2 Implementation

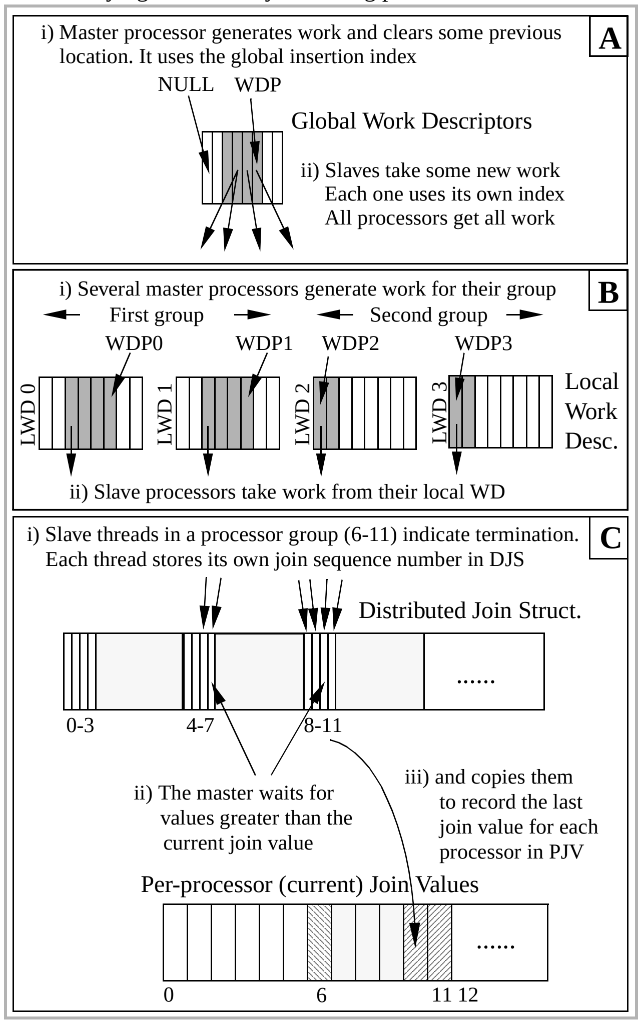
The two WD proposals are implemented as arrays of pointers to work descriptors, behaving as circular lists (see Figure 2, A and B). There is a shared GWD structure and one per-processor LWD structure. Each processor searches for work first in its own LWD and then in the GWD. The size of the WD structures is a multiple of the secondary cache line size and they are aligned to cache line boundaries to avoid false sharing. Both implementations use one-way communication from the master processor to the slave processors. This means that the master processor writes pointers and the slaves read them. After the master processor writes a pointer to a WD location, it takes advantage of having exclusive access to the cache line to also clear a previously used location. This one-way communication mechanism saves several cache misses and invalidations while generating work, thus speeding up part of the critical path of the run-time library. Each processor has its own local index to the WD structures to extract work from them. It knows that there is no work in a WD structure when the location pointed by its index is NULL. The master processor uses a global index (WDP) to store a new pointer in the next available location of the WD structures. The global index is necessary to allow several processors to add work at the same time. Mutual exclusion through load-linked and store conditional instruction sequences is used to update this global index.

For example, in Figure 2A, four work descriptor pointers (shaded area) are currently stored in the GWD, possibly from different parts of the application. Not all processors have executed the same number of descriptors and some of them are extracting work from different locations. Each processor has a local index for work extraction.

Figure 2B presents the implementation of the LWD. It shows four LWD structures for four different processors. In the example, two groups of two processors are already spawned. Each master processor (0 and 2) uses an index associated to each LWD to insert new work. This index is accessed in mutual exclusion to allow several processors to add work to a LWD at the same time. Again, clearing an already used location during work insertion improves the cache behaviour. Each processor waits for work in its LWD using a local index. Processors working in a group are consecutive and are identified 0 (the master), 1, 2, and so on.

3.2. Thread Joining

Like many implementations, our proposal for thread joining is based on a distributed structure, to minimize false sharing. However, we add a local sequence number for each processor and a per-processor join values array (PJV) to support multiple levels of parallelism. The per-processor sequence number allows that each individual processor participates in a different number of parallel regions before collaborating again in the same group due to a change in the structure of the parallelism. This is different from the previous implementations, where the sequence number was identifying the currently executing parallel construct.



i) Master processor generates work and clears some previous location. It uses the global insertion index
A
NULL
WDP
Global Work Descriptors
ii) Slaves take some new work
Each one uses its own index
All processors get all work
i) Several master processors generate work for their group
B
First group
Second group
WDP0
WDP1
WDP2
WDP3
LWD 0
LWD 1
LWD 2
LWD 3
Local Work Desc.
ii) Slave processors take work from their local WD
i) Slave threads in a processor group (6-11) indicate termination.
Each thread stores its own join sequence number in DJS
C
Distributed Join Struct.
......
0-3
4-7
8-11
ii) The master waits for values greater than the current join value
iii) and copies them to record the last join value for each processor in PJV
Per-processor (current) Join Values
......
0
6
11 12

Figure 2: Thread fork / join data structures

Figure 2C shows the proposed implementation. When a slave processor in a group terminates its work, it writes its per-processor join sequence number in the Distributed Join Structure (DJS). One cache line stores the sequence numbers of four processors. The master processor has a copy of each thread sequence value in its PJV. It waits for completion of the parallel work by looking at both ends of the DJS. When it detects that the value stored in DJS for one of the two current locations is greater than its local copy, it records the new value in PJV and proceeds to the next processor. Both the DJS and the PJV are allocated in the stack of the master processor in order to allow the existence of several master processors at the same time.

4. EVALUATION OF THE FORK/JOIN TECHNIQUES

The goal of this section is to evaluate the proposed fork/join techniques and compare them with the ones currently implemented in the SGI MP library. Later in this paper we show the usefulness of the multiple levels of parallelism exploitation and processor grouping. This will validate the fork/join techniques and justify the additional overhead they introduce.

All the experimental evaluation in this paper has been done on a dedicated SGI Origin2000 machine [16] containing 64 R10000 processors (250 Mhz., chip revision 3.4) and 8 Gb. of main memory. Each processor uses separate primary caches for instructions (32 Kb.) and data (32 Kb.) and a unified 4 Mb. secondary cache. The operating system is IRIX release 6.5. Unless noted otherwise, all benchmarks have been compiled with the MIPSpro FORTRAN 77 release 7.2.1.1m, using the following compiler flags: -Ofast=ip27 -LNO:prefetch_ahead=1:auto_dist=ON. These are the options used to compile the base versions of the SPEC95fp benchmarks.

A synthetic benchmark (called “overhead”) is used to evaluate the overhead introduced by the techniques presented in Section 3 when executing fine-grain parallelism. “Overhead” consists of 1,000 iterations spawning and joining parallelism. Several experiments have been carried out in the Origin2000 machine with a work size ranging from 1,000 to 1,000,000 iterations. From them, we have selected the most relevant ones: Experiment #2 executes 4,000 iterations, taking around 100 us.; Experiment #6 does 50,000 iterations, taking 1.2 ms.; And experiment #9 is 400,000 iterations, taking 10 ms. The benchmark is run on both the SGI MP library and NthLib.

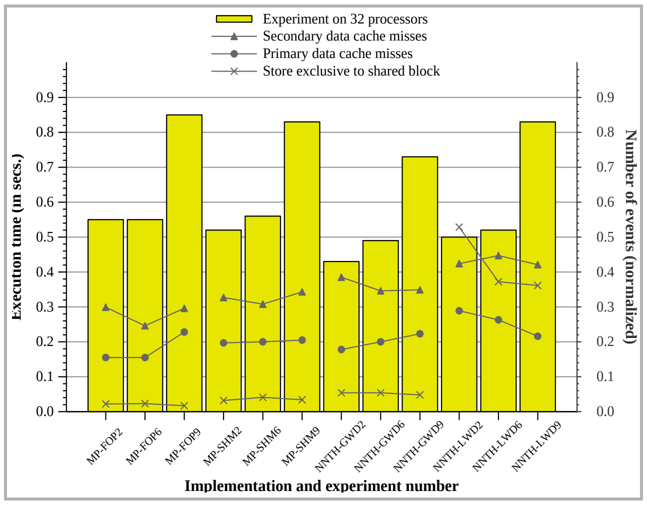


Figure 3: Evaluation of the fork/join mechanisms

Figure 3 compares the four techniques for work generation with respect to the execution time and cache behaviour. MP-FOP and MP-SHM stand for the two different joining mechanisms inside the MP library. NNTH-GWD and NNTH-LWD stand for the two different forking mechanisms with distributed joining in NthLib. Each bar represents the execution time of the benchmark on 32 processors, running the experiment indicated by the associated label. For instance, NNTH-LWD2 corresponds to the experiment #2 using the LWD technique. Along with the execution times of the benchmark, the plot presents the normalized numbers of primary and secondary data cache misses and the normalized number of store operations requiring exclusive access to a shared cache line. These events have been collected using the R10000 hardware event counters, through the *perfex* analysis tool, provided by SGI.

Execution times in Figure 3 show that all techniques are comparable with respect performance. Differences arise when comparing the behaviour of the cache. Observe that for the FOP, SHM and GWD techniques, the amount of cache events are similar. In LWD, the number of stores that reclaim exclusive access to a cache line increases. This is due to the way the work is supplied to processors, one to one. The store event is caused each time the master processor generates work, getting exclusive access to the cache line where it stores the pointer to the work descriptor. The cache line is, then, read by the destination slave processor, which requests shared access to the line. Thus, one cache line per processor exchanges twice its status each time work is generated. This movement between cache memories is not affecting the performance of the overhead benchmark, but it can affect the performance of parallel applications.

Figure 4 shows the execution times obtained when running the overhead benchmark (4096 iterations, from 1 to 64 processors). The results confirm that the GWD and LWD implementations are comparable to the MP library implementation. In addition, the figure also presents results for a multilevel version of the overhead benchmark. This version spawns a first level of parallelism consisting of four sections and an inner level of loop parallelism executing 1,024 iterations (using LWD), so the total amount of work is the same. This is represented by the bar labeled NNTH-2L2. It shows that the overhead of the multilevel version is comparable to the overhead of the single level one.

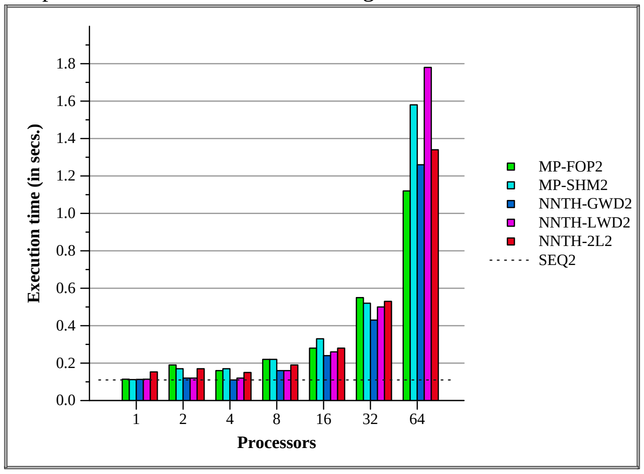


Figure 4: Fork/join evaluation (overhead)

Figure 5 shows the execution times for a parallel version of the SPEC95fp Swim application (with one level of parallelism and using the GWD technique) using from 1 to 64 processors. The execution times of the Swim application show that the GWD approach is comparable to the SGI implementation also when running complete applications.

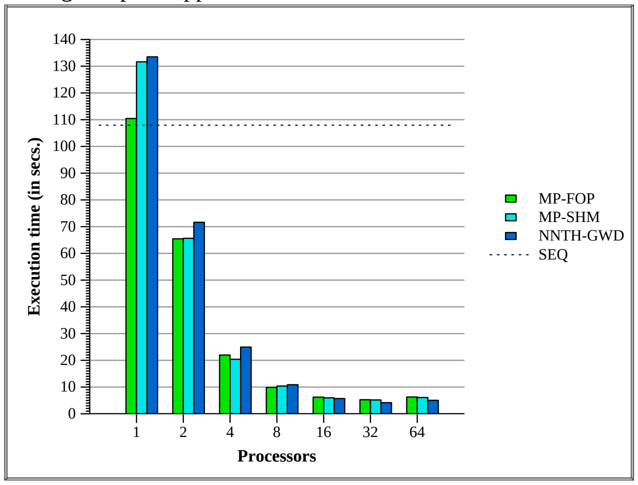


Figure 5: Fork/join evaluation (SPEC95fp Swim)

5. MULTI-LEVEL PARALLELISM EXPLOITATION

In the rest of the paper, we use two different applications to validate the implementation of the fork/join techniques presented in Section 3. First, we show that their efficiency is good enough to support multiple-levels of parallelism. A second result which is obtained is that multiple-levels of parallelism can be effectively exploited inside applications. The applications selected are the SPEC95fp Hydro2D and the NAS APPBT benchmarks. The structure of the applications is described in the following subsections.

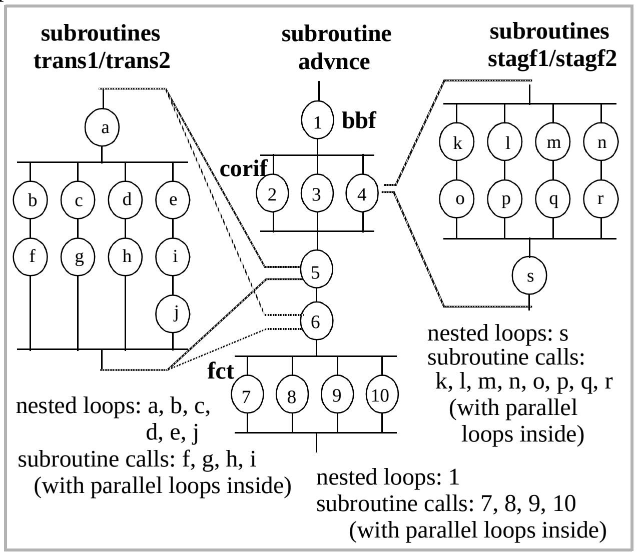
5.1. SPEC95fp Hydro2D Benchmark

The Hydro2D application solves the hydrodynamical Navier Stokes equations to compute galactical jets. The *reference* input for Hydro2D executes 200 iterations of a time-step loop.

Figure 6 shows the structure of one of the two computational steps in the main time-consuming routine (*advnce*) executed inside the time-step loop. Up to a maximum of three levels of parallelism can be detected in this application. In the *advnce* subroutine, a set of functions can be executed in parallel since they access different data (nodes *2*, *3* and *4*); these functions contain two levels of parallelism as shown in the right part of Figure 6. Nodes *5* and *6* corresponds to the invocation of functions *trans1* and *trans2*, respectively; both functions also contain two-levels of parallelism as shown in the left part of Figure 6. Nodes *7* to *10* contain a call to the *fct* subroutine. This subroutine has parallel loops.

Two different parallelization strategies have been tested. The first one exploits a single level of loop parallelism. The second one is a two-level version which exploits all the parallelism inside *corif*/ *stagf1*/*stagf2*, *trans1*/*trans2* and the parallelism of the parallel invocations to *fct* subroutine including parallel loops. Although nodes *2*, *3* and *4* can be also executed in parallel (so three levels of parallelism could be exploited), we decided to execute them sequentially in order to improve data locality. Spawning at the outer level distributes work to the master processors for each group. For instance, in the 16 processor execution, processors numbered 0, 4, 8 and 12 are the group master processors. At the inner level, each group master processor entering a new work-sharing construct spawns parallelism on its group. For example, processor 4 is going to spawn parallelism on processors 4 (itself), 5, 6 and 7.

As the application structure is regular, Hydro2D is useful to demonstrate how a well balanced allocation of the work at the different levels of parallelism to processors can result in a better processor utilization.



subroutines trans1/trans2
subroutine advnce
subroutines stagf1/stagf2
bbf
corif
fct
nested loops: a, b, c, d, e, j
subroutine calls: f, g, h, i (with parallel loops inside)
nested loops: 1
subroutine calls: 7, 8, 9, 10 (with parallel loops inside)
nested loops: s
subroutine calls: k, l, m, n, o, p, q, r (with parallel loops inside)

Figure 6: Multiple levels of parallelism inside the SPEC95fp Hydro2D benchmark

5.2. NAS APPBT Benchmark

The NAS APPBT benchmark [17][18] solves three sets of uncoupled block tridiagonal systems of equations, first in the *x*, then in the *y* and finally in the *z* direction. Each block contains 5x5 elements. These systems arise in many CFD applications.

The structure of the application is as follows: An iterative loop sequentially calls to routines *compute_rhs*, *x_solve*, *y_solve* and *z_solve*. The dependences in these routines determine which loops can be parallelized. For instance, *x_solve* carries the dependence in the first dimension being the loops that traverse the second and third dimension completely parallel; similarly, *y_solve* carries the dependence in the second dimension and *z_solve* in the third dimension. A possible strategy would be to parallelize the loop that traverses the third dimension in routines *x_solve* and *y_solve* and parallelize the loop that traverses the second dimension in routine *z_solve*. Although this parallelization strategy implies totally parallel loops, it suffers from the data movement overhead (transposition) that occurs when going from *y_solve* to *z_solve* and back again to *x_solve*.

The data movement overhead of the transposition can be avoided if the third dimension is also parallelized in *z_solve*; this requires the use of the OpenMP ORDERED clause and directive that forces the sequential execution of the distributed loop iterations. In order to allow a pipelined execution of the ORDERED dimension, loop blocking is applied. In this way, a chunk of iterations in processor *p+1* is executed when the same chunk of iterations finish its execution in processor *p*. Figure 7 shows the data distribution among processors and the resulting execution model for the one-dimensional parallelization in the *z_solve* routine. Although this introduces the overhead of blocking and synchronization, the overlap of different chunks in different processors can result in an improved performance.

When the number of iterations is small to fed a large number of processors, two dimensions are worth to be parallelized. In order to avoid data movement, our strategy parallelizes the second and third dimension in all the routines. This implies that two dimensions are executed in parallel in *x_solve*, but one of the two dimensions parallelized are executed in an ORDERED way in both the *y_solve* and *z_solve* routines. Figure 8 shows the data distribution performed among processors and the resulting execution model for the multidimensional parallelization in the *z_solve* routine.

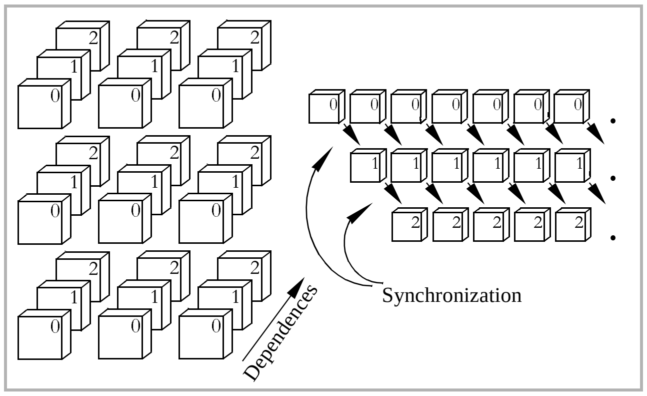


Figure 7: One-dimensional data distribution and pipelined ORDERED execution in the APPBT application

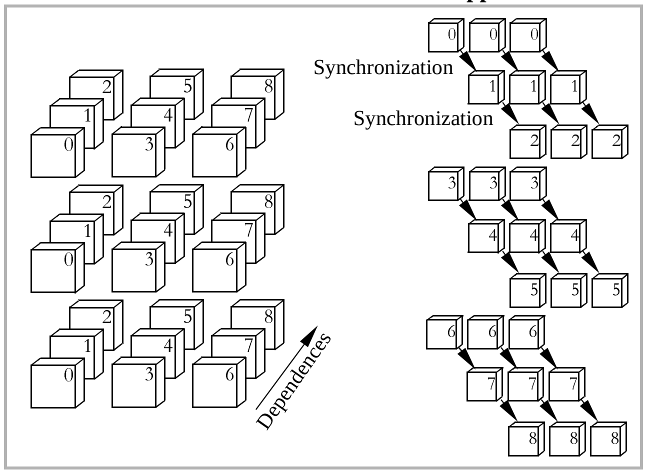


Figure 8: Two-dimensional data distribution and pipelined ORDERED execution in the APPBT application

6. MULTI-LEVEL PARALLELISM EVALUATION

6.1. SPEC95fp Hydro2D Benchmark

Four different versions of the Hydro2D benchmark have been executed using the *reference* input file, as provided in the SPEC benchmarks. Figure 9 shows the speedup obtained. Sequential execution time is 154.71 seconds, which agrees with the 154 seconds reported in the SPEC benchmark CFP95 summaries [19].

Bar labeled MP-SHM correspond to the speedup obtained when the application is run on top of the SGI MP library. In this case, the application has been manually parallelized using OpenMP directives to exploit loop parallelism and compiled with the native SGI compiler. Results are better than those reported in the SPEC summaries because of the manual parallelization. For instance, reported results using automatic parallelization through PFA (Parallel Fortran Analyzer) are 39.1 seconds on 8 processors and 35.9 seconds on 16 processors. Results obtained through manual parallelization are 32.7 seconds on 8 processors and 23.1 seconds on 16 processors. These results validate the parallelization we have manually performed on this application.

Bars labelled NNTH-1LVL have been obtained by executing the same loop level parallelization on top of the NthLib threads package using GWD. In this case, the application has been first parallelized using the NANOS compiler. The resulting parallel Fortran code, containing calls to the NthLib threads package is then compiled using the MIPSpro Fortran 77 compiler with the usual compilation options. As can be observed in Figure 9, the execution time of this version is worse than the SGI MP library version. The main reason for this poor performance is that the quality of the code generated by the SGI OpenMP compiler is better than the quality of the code generated for the NthLib. This is because the MIPSpro Fortran 77 compiler applies some code optimizations before and after the parallelization takes effect. This behaviour is inhibited in our approach because the task of parallelism extraction is done by the NANOS compiler, which is not applying any optimization before parallelizing. This difference in the quality of the generated code has a greater influence when the number of processors is small. However, up to a certain number of processors (32), the speedup obtained by the two one-level implementations is similar.

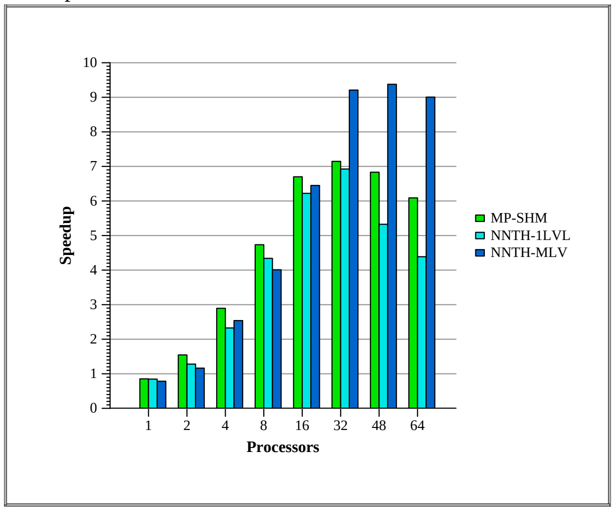


Figure 9: Execution time and speedup of the SPEC95fp Hydro2D benchmark

The last bar (labelled NNTH-MLV) is the result of the multi-level execution. The application exploits two levels of parallelism as explained in Section 5. Results show that the overhead introduced by the two-level parallelization is noticeable when running on a small number of processors. A fair comparison can be established with the NNTH-1LVL version, in which the quality of the code generated is the same. As a result, for up to 8 processors, the overhead of the two-level parallelization motivates an increment in the execution time below 10%. In this application, four different processor groups are established at the outer parallelization level, so at least 8 processors are required to effectively exploit multiple levels of parallelism. Four groups of 2 processors give an speedup of 4.0 compared to the 4.5 achieved by the one-level version running in 8 processors. When 16 processors (four groups of 4 processors) are used, nearly the same speedup is achieved by both NNTH versions. When using more than 16 processors, the one-level version is unable to scale, while the multi-level version scales till a speedup of 9.3 on 48 processors and giving an improvement of 30% in 32 processors with respect the single level version.

Processor grouping promotes scalability in this application because the parallel loops in the inner parallel level are distributed among less processors when executed in parallel. This causes each processor to keep a larger working set belonging to each matrix used in the loop body, thus reducing the amount of cache conflicts among the processors.

6.2. NAS APPBT Benchmark

Figure 10 shows the execution time and speed-up of the NAS APPBT benchmark. We have selected a small version (class W) of the APPBT application in order to better appreciate the influence of the parallelization overhead. All versions of the APPBT application have been compiled with -O3 compilation option instead of -Ofast=ip27 because this is the option used in the standard compilation of the NAS benchmarks in SGI machines. Due to the parallelization scheme and the application class, the application can be executed on as much as 24 processors.

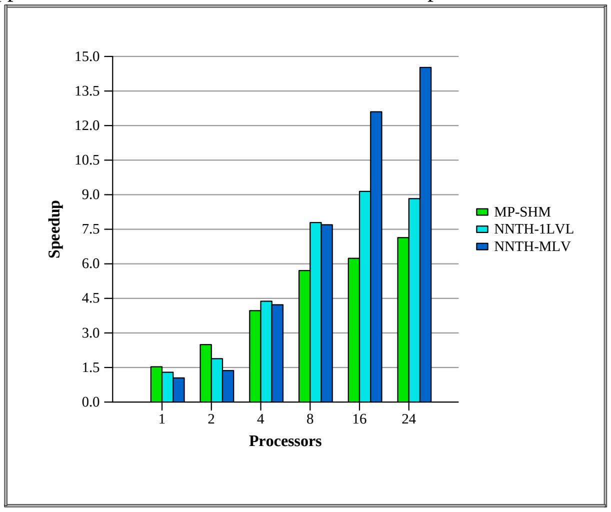


Figure 10: Execution time and speedup of the NAS APPTBT benchmark

In Figure 10, MP-SHM stands for the one-level version executed on the SGI MP Library. This result is provided as a reference. Two NNTH versions have been developed, one for single dimension parallelization (NNTH-1LVL) and another for multidimensional parallelization (NNTH-MLV), achieved through two-levels of parallelism (as shown in Figure 8).

Again, comparing the results in the nano-threads environment, the results on a small number of processors show that the multi-level version is worse than the one level version. This is due to the extra overhead introduced by the multi-level version. However, when using more than 8 processors, the multiple-level version achieves higher speedup, reaching 14.5 on 24 processors. The gain in the speedup reaches 65% with respect to the one dimensional parallelization.

7. CONCLUSIONS

Current parallelization environments for shared memory multiprocessor systems are based on highly tuned and customized thread packages offering the mechanisms required for thread creation (fork) and joining, thread identification and so forth. Such environments (either commercial or experimental) do not allow the exploitation of more than one level of parallelism.

This paper presents techniques for efficient thread forking and joining in parallel execution environments, taking into consideration the physical structure of NUMA machines and the support for multi-level parallelization and processor grouping. Two work generation schemes and one join mechanism are designed, implemented in the NthLib package, evaluated and compared with the ones used in the IRIX MP library, an efficient implementation which supports a single level of parallelism.

For a single level of parallelism, the MP library uses single data structures located at fixed memory locations that do not enable several processors to coordinate (spawn and join) parallel activities, thus restricting the parallelism that can be exploited at the application level. In order to overcome this lack of functionality, we provide per-thread data structures that enable the coordination of some slave threads with a master thread, set when spawning an outer level of parallelism. Per-thread data structures also allow multiple threads to coordinate parallel activities coming from different parts of the application at a time.

Both mechanisms are evaluated and validated using a synthetic benchmark for measuring their overhead, two SPEC95fp applications (Swim and Hydro2D), and one NAS benchmark (APPBT). The overhead benchmark and Swim (with a single level of parallelism around loops) allow us to conclude that the additional functionalities in our fork/join mechanisms do not result in a noticeable reduction of the thread package efficiency. The other two benchmarks expose multiple levels of parallelism and benefit from its exploitation. The comparison with the traditional one-level parallelism exploitation for these programs gives an improvement of 30% and 65% in the speedup of Hydro2d (on 32 processors) and APPBT (on 24 processors), respectively. These results validate the exploitation of multiple levels of parallelism based on the thread fork/join techniques described in this paper.

8. ACKNOWLEDGEMENTS

We would like to thank to the partners of the NANOS project for the insightful discussions on various aspects of this paper.

This work has been supported by the European Community under the ESPRIT project E21907 (NANOS) and the Ministry of Education of Spain (CICYT) under contracts TIC97-1445-CE and TIC98-0511.

9. REFERENCES

[1] Laudon, J. and Lenoski, D., "The SGI Origin: A ccNUMA Highly Scalable Server", Proc. of the 24th Int. Symposium on Computer Architecture (ISCA), pp. 241-251, June 1997.

[2] Charlesworth, A., "STARFIRE: Extending the SMP Envelope", IEEE Micro, Jan/Feb 1998.

[3] OpenMP Organization, www.openmp.org.

[4] Ayguadé, E., Martorell, X., Labarta, J., González, M. and Navarro, N., "Exploiting Parallelism Through Directives in the Nano-Threads Programming Model", 10th Workshop on Programming Languages and Compilers for Parallel Computing, Minneapolis (USA), August 1997.

[5] MIPS Technologies Inc., "MIPS R10000 Microprocessor User's Manual", version 2.0, January 1997.

[6] Cortesi, D., Raithel, J., Tuthill, B., "IRIX Device Driver Programmer's Guide", doc. 007-0911-120, SGI, 1998.

[7] Hall, M. W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Bugnion, E. and Lam, M.S., "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, December 1996.

[8] Girkar, M., Haghighat, M.R., Grey, P., Saito, H., Stavrakos, N., Polychronopoulos, C.D., "Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems", Intel Technology Journal, Q1 issue, February 1998.

[9] Foster, I., Kohr, D.R., Krishnaiyer, R., Choudhary, A., "Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface". Supercomputing'96, November 1996.

[10] Gross, T., O'Halloran, D. and Subhlok, J., "Task Parallelism in a High Performance Fortran framework", IEEE Parallel and Distributed Technology, vol. 2, no. 3, Fall 1994.

[11] Ramaswamy, S., "Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations", Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1996.

[12] Martorell, X., Labarta, J., Navarro, N. and Ayguadé, E., "A Library Implementation of the Nano-Threads Programming Model", In Europar'96, August 1996.

[13] Polychronopoulos, C.D., Girkar, M. and Kleiman, S., "nano-Threads: A user-level threads architecture", technical report, CSRD, Univ. of Illinois at Urbana-Champaign, 1993.

[14] Polychronopoulos, C.D., "Nano-threads: Compiler Driven Multithreading", In 4th Int. Workshop on Compilers for Parallel Computing, Delft (The Netherlands), December 1993.

[15] Polychronopoulos, C.D., Girkar, M., Haghighat, M., Lee, C., Leung, B. and Schouten, D., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Int. Conference on Parallel Processing (ICPP), St. Charles, Illinois, 1989.

[16] Silicon Graphics Computer Systems SGI, "Origin 200 and Origin 2000 Technical Report", 1996.

[17] Waheed, A., Yan, J., "Parallelization of NAS Benchmarks for Shared Memory Multiprocessors", Technical Report NAS-98-010, NASA, March 1998.

[18] Bailey, D., Harris, T., Saphir, W., Wijngaart, R., Woo, A. and Yarrow, M., "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA, December 1995.

[19] SPEC Organization, "The Standard Performance Evaluation Corporation", www.spec.org.