A Library Implementation of the Nano-Threads Programming Model

Xavier Martorell, Jesus Labarta, Nacho Navarro, Eduard Ayguade

Departament d'Arquitectura de Computadors (DAC)
Universitat Politecnica de Catalunya (UPC)
Gran Capita s/n, Campus Nord, Modul D6, 08071, Barcelona, Spain
{xavim, jesus, nacho, eduard}@ac.upc.es

Abstract. In this paper we describe the design and implementation of a user-level thread package based on the nano-threads programming model, whose goal is to efficiently manage the application parallelism at user-level. Nano-thread applications work close to the operating system to quickly adapt to resource availability.

The goal is to obtain an efficient parallel execution of the nano-threads by appropriately balancing the work assigned to each thread and the thread management overhead. Early experiments let us determine that the appropriate number of operations spread out among the threads to ensure less than 10% of overhead is around 800. Recent experiments show that this nano-thread granularity is fine enough to adapt easily to the system conditions, granting a reduced response time.

1. Introduction

Traditional user-level thread packages are used by programmers to parallelize by hand their applications. Those packages are mostly oriented to coarse-grain structured parallelism [7][12]. Nowadays, new thread packages evolve to give support to compiler generated parallel code. Compilers mainly manage one-level of structured parallelism (do-loops) and also coarse-grain parallelism. Research on new programming models is a topic of great interest for the success of general purpose parallel computing. The goal is that new thread packages could offer efficient execution of several level parallelism, less structured, more fine-grained and flexible than current ones.

In this paper we are going to describe the design and implementation of a user-level thread package based on the nano-threads model [10]. Our environment assumes that applications (e.g., C or FORTRAN programs) are automatically decomposed by a parallelizing compiler. The compiler identifies the maximum parallelism of the application through data and control dependence analysis and generates an intermediate representation of the parallel application in the form of a hierarchical task graph

This work has been supported by the Ministry of Education of Spain (CICYT) under contracts TIC95-0492 and TIC94-0439.

(HTG)[2][6]. We plan to use the Parafrase-2 compiler [9] to generate executable code from the HTG intermediate representation.

2. Objectives

Objectives of this paper are to study the viability of the nano-threads parallel programming model demonstrating that

+ It is possible to build an efficient implementation of a nano-thread package to manage the application parallelism at user level.
+ The run-time overhead related to the creation and management of parallel threads can be kept very low, so that efficiency of parallel processing at fine-granularity levels does not depend on library management.
+ The supported granularity is fine-grained enough to ensure a good adaptation to the resource availability.

Section 3 outlines the design of the nano-threads package. Section 4 presents its evaluation. Finally, section 5 outlines some future work.

3. The Nano-threads Library

The execution of a nano-threaded application consists in the execution, in some order and preserving dependencies, of the functions generated from the HTG. An approach is to build a user-level library of routines grouping the services needed by these functions: the nano-threads library.

In the library, the required simplicity in thread management is obtained by managing only one fixed size structure (the nano-thread structure). It contains all the nano-thread information including its descriptor and its stack. Among other attributes, the nano-thread descriptor contains a counter of unresolved dependencies for the nano-thread and a reference to a successor nano-thread.

The nano-thread creation primitive allocates (or recycles) and initializes the nano-thread structure. The library also offers simple primitives to control the (already) unresolved dependencies, to manage the user-level ready queue, and to help in the implementation of parallel loops.

The core of the library contains the nano-thread scheduling loop that searches for work in the ready queue. This loop is executed by as many kernel threads as processors are allocated to the application. Also, the library is able to adapt to variations in the number of processors allocated to it.

More information about the current implementation of the nano-threads library can be found in [5].

4. Evaluation

We have implemented the nano-threads library on top of the Mach 3.0 microkernel [1]. We use a 4 i486 (33 Mhz) multiprocessor architecture

(DEC433MP) with 32 Mb. of main memory and 256 Kb. of coherent cache at each processor. Our implementation of the nano-threads library is based on the Quick Threads package [3].

Execution times for the basic nano-thread library primitives remain below 15 us. Starting a nano-thread costs 25 us. in the given architecture. This can be compared with the time required to perform a floating point addition (0.15 us) and a floating point multiplication (0.27 us).

4.1. Granularity Experiments

A first evaluation of the nano-threads package was done using a matrix by vector product application. Figure 1 shows the application execution times and speed-up on 2, 3 and 4 processors. The main goals of the experiments were the study of the nano-thread granularity and its effect on the execution time and speedup of the application.

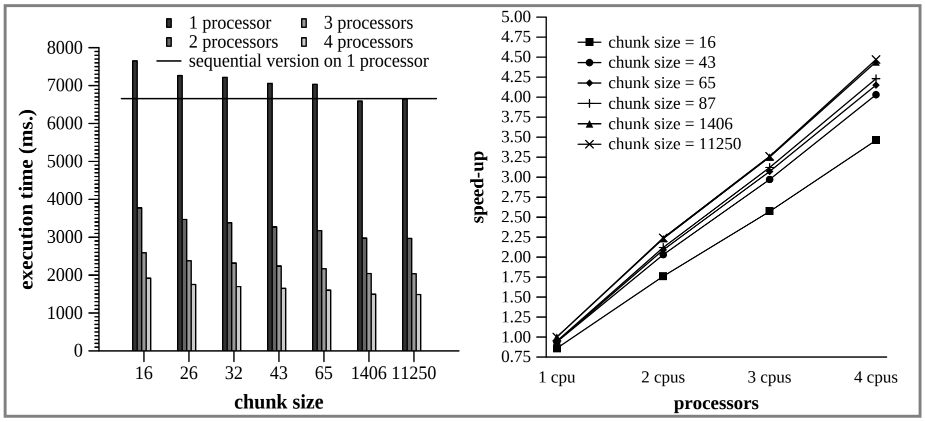

Figure 1: Execution time and speed-up for the matrix by vector product.

The application we used worked with a big matrix and a small vector (9 elements). The chunk size assigned to each nano-thread lets the application to span a wide range of granularity levels. It can use chunks containing from 1 to 11250 matrix rows (with 18 fp-ops for each row).

In terms of operations executed by each nano-thread, the results indicate that the minimum number of floating point operations to be performed by a nano-thread should be greater than 800 (43 iterations * 18 fp-ops/iteration = 774 fp-ops) to ensure an overhead lower than 10% of the sequential execution time.

Application speedups reflect the cache influence. Four processors provide four times more cache than a single one. This reduces the number of cache misses in the overall execution, thus producing a super-linear speedup in some experiments.

4.2. Adaptability Experiments

The nano-threads programming model is defined to adapt to the underlying number of processors. This means that the operating system can add and remove processors to/from applications. The steps followed by the kernel preemption mechanism are:

+ The operating system decides to reassign a processor from an application to another. It requests a processor to the source application and waits for the response (the reaction time).
+ The application detects the request and it releases a processor at a safe point (at the end of a nano-thread), avoiding the preemption of work, which may be in the critical path.
+ The operating system transfers the processor to the destination application.
+ In case the source application does not respond in the given reaction time, the operating system steals a processor from it, independently of the work it is doing.

Two different techniques to release and return processors have been implemented: first, using the thread suspend/resume primitives; and second, modifying the priority of the kernel threads associated to the application. The measurements presented here are taken using the first technique.

We have used a Jacobi iteration based on a matrix of 480 rows. The resolution spends 250 iterations. Two work generation schemes are tested: first, a fixed chunk size (2 and 24 rows); and second, varying the chunk size in a guided self-scheduling style. Elapsed execution times are given in figure 2. Observe the overhead introduced by the fixed small chunk size (2) compared with the *gss* generation style, using 1, 2 and 3 processors.

We have implemented a high-priority user-level process which simulates the operating system scheduling, stealing and returning processors from/to the application. Shared memory is used for communication between the kernel simulator and the application. In the experiments, one processor is preempted from the application and assigned to it periodically for the same amount of time (*t-ncpuchange*). A wide range in *t-ncpuchange* is explored to cover from quick movement of processors (40 ms.), to larger periods used in other scheduling works (4 s.) [12]. The system simulator is configured to set no reaction time or a reaction time of 3 ms.

In general, applications that react in time always perform better than applications that do not. A small chunk size (2 rows) is a good warranty of response. *Gss* applications do not respond because they generate some too large chunks for each iteration.

When the *t-ncpuchange* is small, fixed chunk size (2 or 24) performs better than *gss* because the latter generates bigger chunks. Generally, preemption can occur in the middle of the execution of the chunk, and not only the application looses one processor, but a great amount of work blocks until the processor returns. Also, this is the reason why applications with big chunk sizes perform worse when the *t-ncpuchange* is larger.

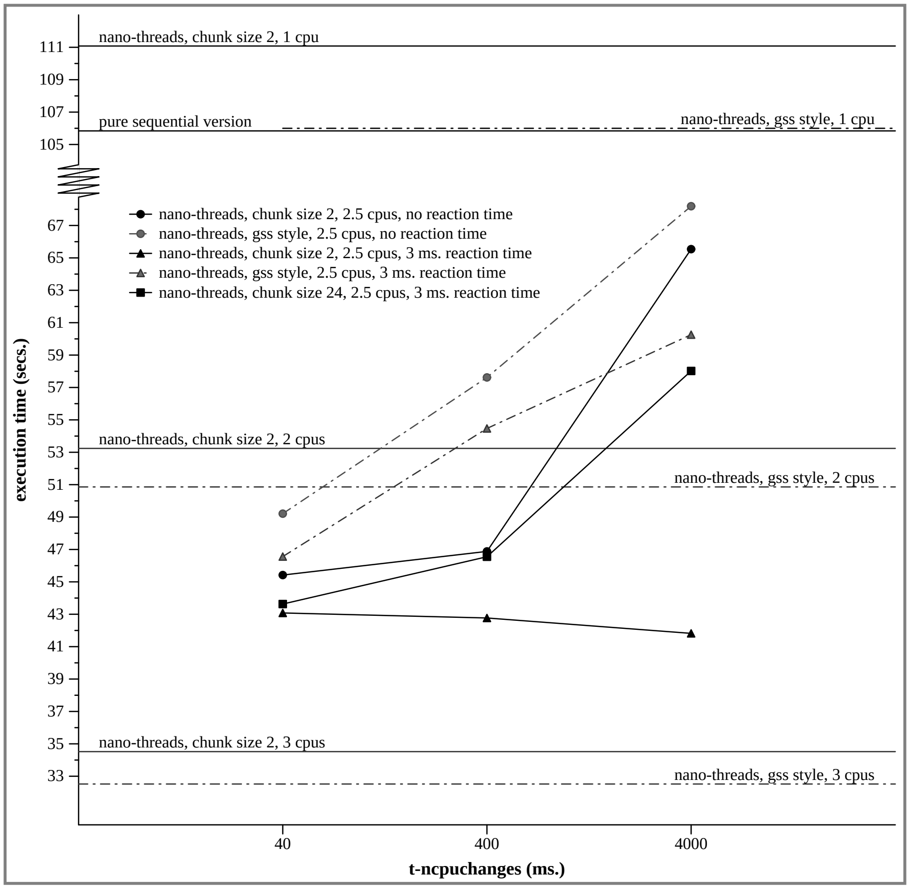

Figure 2: Effect of the processor assignment in the execution time.

5. Future Work

More experiments, measurements and behavioural studies are needed. We plan to trace the execution of applications based on the library to determine where and when the processors become idle or the management done by the library does not fit the application requirements. We are implementing an instrumented version of the library that interfaces to a visualization tool (PARAVER [8]). This tool will enable us to see the actual run-time behaviour. It will also be interesting the study of the influence of the operating system virtual memory management mechanism in the performance of the applications.

We want to test new user-level scheduling policies; for example, dynamic chunk sizes based on trapezoid self-scheduling, affinity scheduling, etc. [4], and to determine the quality of adaptability of the library to larger variations in the number of processors. The results will be very useful to apply to a multi-user environment.

The nano-threads library is now being ported to other architectures: in particular the SGI Power Challenge and the DEC Alpha AXP. We also plan to port the library on top of the Chorus microkernel [11].

6. Acknowledgements

We would like to thank Constantine D. Polychronopoulos for the initial discussions about nano-threads and the possibilities of implementation.

7. References

1. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M.: Mach: A New Kernel Foundation for UNIX Development. Proc. of the Summer 1986 USENIX Conference, July 1986.
2. Girkar, M., Polychronopoulos, C. D.: Automatic Extraction of Functional Parallelism from Ordinary Programs. IEEE Trans. on Parallel and Distr. Systems, Vol. 3, No. 2, March 1992.
3. Keppel, D.: Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.
4. Markatos, E. P., LeBlanc, T. J.: Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. Proc. of the Supercomputing-92, 1992, pp. 104-113.
5. Martorell, X., Labarta, J., Navarro, N., Ayguadé, E.: Nano-Threads Library Design, Implementation and Evaluation. Dept. d'Arquitectura de Computadors - Universitat Politècnica de Catalunya. Technical Report: UPC-DAC-1995-33, September 1995.
6. Moreira, J. E.: On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors. PhD. thesis, Department of Electrical and Computer Engineering, Univ. of Illinois at Urbana-Champaign, 1995.
7. Mueller, F.: A Library Implementation of POSIX Threads under UNIX. 1993 Winter USENIX, January 25-29, 1993, San Diego, CA.
8. Pillet, V., Labarta, J., Cortés, T., Girona, S.: PARAVER: A Tool to Visualize and Analyse Parallel Code, WoTUG-18, pp 17-31, Manchester, April 95.
9. Polychronopoulos, C. D., Girkar, M., Haghighat, M. R., Chia Ling Lee, Leung, B., and Schouten, D.: Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. International Journal of High Speed Computing, Vo. 1, No. 1, 1989.
10. Polychronopoulos, C. D.: *nano*-Threads: Compiler-Driven Multithreading. CSRD Technical Report, 1993.
11. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., et al.: Overview of the Chorus Distributed Operating System. Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures, April 1992.
12. Tucker, A., Gupta, A.: Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. ACM Operating Systems Rev., Vol 23 Num 5, Dec. 1989.