References

[Ande 89] T. Anderson, E. Lazowska, H. Levy, “The Performance Implication of Thread Management Alternatives for Shared-Memory Multiprocessors”, ACM Trans. on Comp. Vol. 38 No. 12, Dec. 1989

[Ande 92] T. Anderson, B. Bershad, E. Lazowska, H. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM Trans. on Comp. Sys. Vol. 10 No. 1, Feb. 1992

[Ghos 93] K. Ghosh, “Experimentation with Configurable, Lightweight Threads on a KSR Multiprocessor”, Georgia Institue of Technology: Technical report GIT-CC-93/37

[LeBl 89] T. LeBlanc, “Memory management for large-scale numa multiprocessors”, Department of Computer Science: Technical report*311

[Rüde92] Ulrich Rüde, “On the multilevel adaptive iterativ Method”, SIAM journal on scientific and statistical computing, Vol. 15, 1994

In the beginning, all threads corresponding to a grid point have the state “suspended”. A set of start points is calculated. These points are those with the highest local error (in the example these points are the grid points at both inlets of the faucet). In the next step all threads in the start set become active. An active thread calculates it’s local error. If this error is greater than a given threshold, the corresponding grid point is calculated. Because this could change the local error of all threads responsible for points in the neighborhood, those threads have to be activated. After this activation, the thread suspends himself. The algorithm terminates, if all threads have the state “suspended”.
The adaptive method is easy to implement with a threads library. The runtime system will distribute the threads to the processors available in the system. But there is the hard problem of a bad data locality leading to huge time penalties for the processors in a NUMA architecture. To avoid this, prefetch and poststore operations have to be mixed with computation statements in the code to overlap computation and data transfer to local caches. For this optimal mix of computation and data management you have to provide

- a high memory locality of data by storing thread control information (Thread Control Block TCB) and thread related data together in the same memory area
- a scheduler interface to be able to prefetch data, which will be necessary in the near future in the processor cache, and to poststore data, which might be used by other processors in the near future.

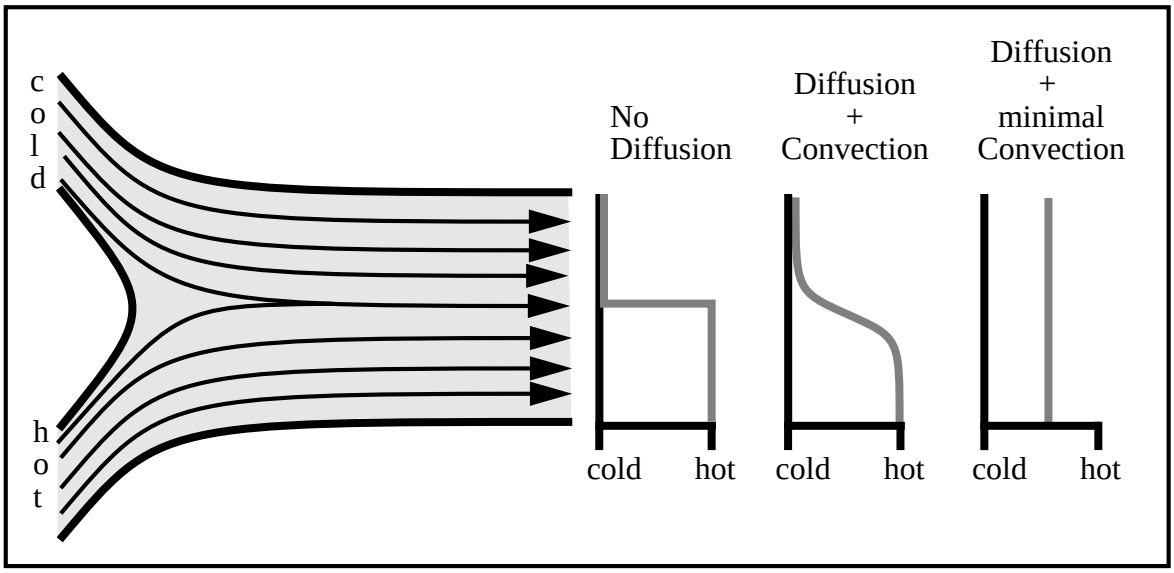
Like the application programer in the example above, a compiler doing automatic parallelization can use the fast threads package to produce fine granular code which will be managed by the runtime system.

2 How to implement fast threads?

The specification and implementation of a fast threads package for NUMA architectures could be the core of a successful cooperation between the CONVEX Corporation, the chair of operating systems and the bavarian consortium for high performance computing (FORTWIHR). The CONVEX SPP would be the appropriate hardware to realize this project.

1 Why using fast user level threads?

To demonstrate the advantage of the threads programming model, we discuss possible implementation techniques for adaptive numerical methods on unstructured grids. Using a message passing approach you have the problem of grid partitioning, data distribution and load balancing. Switching to the shared memory model with parallel processes and shared memory segments, you have still the partitioning and load assignment problem. One approach to solve these problems is the use of a fast threads package with a smart user level interface which enables a concise programing style. To demonstrate this style we user an example in the area of fluid mechanics:
Cold and hot fluid streams through a faucet. We are interested in the temperature at the outlet of the faucet depending on diffusion and convection.



After a discretisation of the problem area we assign each grid point to a thread. All threads perform a kind of code like the one below.

```
func working_thread(myself, threshold)
    while (1)
        prefetch read_only Next-TCB
        prefetch read_only TCBs of Next-TCB's Neigbors
        if (error(myself) > threshold)
         prefetch exclusive TCBs of Neigbors
         localpoint = calculatepoint(neighborpoints)
         activate(neighbors)
         poststore Neighbors
        end if
        suspend(myself)
    end while
```

Techniques for Building a Fast Threads Package on NUMA Architectures

Frank Bellosa

email: bellosa@informatik.uni-erlangen.de

University Erlangen-Nürnberg
Chair of Operating Systems

Abstract

Operating system abstractions do not always reach high enough for direct use by a language or applications designer. The gap is filled by application-specific runtime environments. Typical arguments for their use include complete user-level control over threads scheduling and possibilities regarding the customization of threads synchronization or communications constructs. Especially on NUMA architectures an interface between scheduler and application is essential to overlap computation and memory transfer.

We think about a nonpreemptive user-space threads package with an application interface. The application should be able to get information about scheduling decisions of the runtime system to invoke prefetch operations. Furthermore efficient machine dependent code for creating, running and stopping threads has to be provided by the runtime system. By separating the notion of execution (starting and stopping threads) from threads allocation and scheduling, changing scheduling policies can be as simple as using different function pointers and can be done efficiently at runtime. Thus details of the threads package are not fixed, but can instead be tuned to the needs of the application. To implement this package we want to follow a two level approach: The lower level consists of assembler code for fast thread initialization and context switching. The upper level is a toolbox for building application specific schedulers and synchronization operations. The kernel threads provided by the operating system represent the “virtual processors” of the runtime system. This kind of threads package can only work efficiently, if we use gang-scheduled kernel threads in a multiuser environment or individual-scheduled kernel threads in an environment with just one running application on each processor set.

A fast threads package on NUMA architectures is the prerequisite for an easy implementation of adaptive numerical methods on unstructured grids. A first approach for an implementation is given in the next section.

Last but not least, a fast threads package can be the support library for a compiler doing automatic parallelization.

Techniques for Building a Fast Threads Package on NUMA Architectures

Frank Bellosa

February 1994 TR-I4-6-94

Technical Report

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)