

# Hybrid Parallelization with Dynamic Thread Balancing on a ccNUMA System

Alexander Spiegel and Dieter an Mey

Center for Computing and Communications, RWTH Aachen University, Germany

{spiegel,anmey}@rz.rwth-aachen.de

## Abstract

*SMP Clusters with fat nodes offer an interesting capability for large applications that employ a hybrid parallelization model: to improve load balance, the number of threads can be increased in order to speed-up busy MPI processes or decreased to slow down idle MPI processes, provided these processes reside on the same SMP node. We developed a library which performs this thread adjustment automatically during program execution. On a Sun Fire 15K ccNUMA machine the library also improves data locality by taking advantage of a low level API provided by the Solaris 9 Memory Placement Optimization feature. Experimental results demonstrate remarkable speed-ups with minimal programming effort.*

## 1. Introduction

Hybrid parallelization using message-passing and multi-threading with OpenMP or auto-parallelization on clusters of shared-memory computers is not always profitable [1, 3], yet it offers interesting opportunities [2], particularly on SMP clusters with fat nodes. If an application employs a hybrid parallel programming model utilizing both MPI and threads (from now on we call such an application a "hybrid application" for simplicity), the number of threads can be increased in order to speed-up busy MPI processes or decreased to slow down idle MPI processes, provided these processes reside on the same SMP node.

It may not always be easy to find out the optimal distribution of threads to the MPI processes, however, and the optimal distribution may change in the course of the runtime of an application. Therefore, we developed a dynamic thread balancing (DTB) library which performs this thread adjustment automatically, requiring the user just to insert one extra MPI call in the code to trigger this feature.

The PMPI profiling interface, which is part of the MPI specification and therefore part of each compliant implementation, offers an easy and portable way of

intercepting calls to the MPI library. The DTB library uses this interface to capture the user time of each MPI process and the time spent in MPI routines. This information is evaluated to adjust the number of threads for each MPI process, taking care not to oversubscribe the number of processors dedicated to the application. The major features and the implementation of the DTB library are described in section 2.

First results of experiments performed on a Sun Fire SMP cluster with the new multi-zone version of the BT code of the NAS Parallel Benchmark suite and the FLOWER Navier-Stokes solver have been described in [7]. Here in section 3 we concentrate on further experiments carried out with the FLOWER code taking the ccNUMA architecture of the Sun Fire 15K into account. Section 4 concludes with a summary and an outlook on future work.

## 2. The Dynamic Thread Balancing (DTB) Library

### 2.1. Usage of the DTB Library

The DTB library has to be linked between the user program and the MPI library. One or more calls to the newly provided routine `MPI_Pcontrol` have to be inserted in suitable spots of the program code in order to regularly – typically once per iteration - trigger the steering mechanism. Thus, programming effort for the user of the DTB library is minimal. The calls of the `MPI_Pcontrol` routine are ignored when the DTB library is not linked. The compute time and the MPI overhead between two successive calls of each MPI process are measured and evaluated to adjust the number of threads.

The DTB steering mechanism shifts threads between the MPI tasks residing on the same node with the OpenMP `omp_set_num_threads` call. If freed threads stay in a busy waiting state, they still consume

compute cycles. Therefore they have to be put to sleep completely. On Solaris we hence set the environment variable `SUNW_MP_THR_IDLE` to sleep. For portability reasons we would appreciate the availability of an additional parameter of the `omp_set_num_threads` runtime function as part of the OpenMP specifications. Of course the more MPI processes reside on one node and the more CPUs are available on a node, the better the algorithm is able to shift threads between the MPI processes and to improve load balancing.

When using multiple SMP nodes, the initial distribution of MPI processes onto the nodes is very important: If all the busiest processes are started on the same node, than the DTB algorithm has little to gain. It is thus advantageous to first execute a shorter profiling run - with the balancing mechanism of the DTB library turned off - and then use an optimized distribution for the production run.

## 2.2. Prediction of the Runtime of the MPI Process

In each iteration the number of threads and the compute time is stored for each MPI process. This information is then used to calculate an approximation function  $T(t)$  of the runtime of each MPI process depending on the number threads  $t$

$$T(t) = T_s + T_p / t + T_o * t$$

with

- $T_s$  = serial part of the runtime
- $T_p$  = parallel part of the runtime
- $T_o$  = parallelization overhead

by using the least square method to determine  $T_s$ ;  $T_p$  and  $T_o$ .

If the program behaves smoothly, the corresponding system will have a unique solution with non-negative components. If this is not the case, for example because a loaded system impacts the program performance, we try to find a solution of a simplified approach by setting one or even two of the parameters to zero. In any case, in an overloaded system any increase of the number of threads will diminish the performance and the DTB library will therefore most likely reduce the thread number.

In order to adapt to a changing runtime behaviour, an aging factor, adjustable by an environment variable, is employed to give more recent timing measurements a higher weight.

At program start we can activate a warm-up phase in order to speed-up the initial balancing process. Thereby we quickly gather three values of  $T(t)$  for the

approximation formula. If warm-up is disabled, all processes are started with only one thread and all remaining threads are put into the thread pool (see below).

## 2.3. The DTB Steering Algorithm

A pool of unused threads is managed by the following steering mechanism:

- Search for the MPI process taking maximum compute time (*maxwork\_task*) (This process has to be accelerated.)
- Search for the MPI process with more than one thread and minimum predicted compute time with one less threads (*minwork\_task*) (This process will suffer at least from losing one thread.)
- If the thread pool is empty and the predicted compute time of *minwork\_task* with one less thread is less than the current and predicted compute time of *maxwork\_task*, reduce the number of threads of process *minwork\_task* by one and increase the thread pool (This process may loose one thread, without slowing down the whole application.)
- If the thread pool is not empty and if the predicted compute time of process *maxwork\_task* with one more thread is less than the current maximum compute time, increase the number of threads of process *maxwork\_task* and reduce the thread pool by one. (The busiest process really profits from an additional thread.)
- Else if the predicted compute time of process *maxwork\_task* when loosing one thread is less than the current maximum compute time, decrease the number of threads of process *maxwork\_task* and increase the thread pool by one. (The busiest process does not scale well. In fact, one less thread might be profitable.)

## 2.4. Limitations of the DTB Library

The DTB library currently is limited to the masteronly hybrid programming method (described, for example, in [3]), in which only the master thread calls the MPI library routines outside of any parallel regions. As DTB uses the PMPI interface, other MPI profiling tools like Vampirtrace can not be combined with the DTB library.

### 3. Experimental Results

#### 3.1. Computing Environment

Timing experiments were performed on a Sun Fire 6800 and on a Sun Fire 15K at RWTH Aachen University. The Sun Fire 6800 is equipped with 24 UltraSPARC III Cu processors running at 900 MHz and the Sun Fire 15K contains 72 processors of the same kind. Both machines contain identical system boards with 4 CPUs plus local memory each. But whereas the Sun Fire 6800 has a rather flat memory system, the Sun Fire 15K has a ccNUMA architecture with a ratio of 1:2 concerning the latency of local versus remote memory accesses due to a different cache coherency protocol.

The production environment was Solaris 9 8/03 with Memory Placement Optimization (MPO) feature enabled and we used the Sun ONE Studio 8 compilers. The Sun HPC ClusterTools 5 package includes a fully thread-safe MPI 2 implementation. See [8] for more details about the Sun Fire Servers and MPO.

#### 3.2. The FLOWer Navier-Stokes-Solver

In an ongoing project sponsored by the German Research Council (DFG), scientists of the Laboratory of Mechanics of RWTH Aachen University are simulating PHOENIX, a small scale prototype of the Space Hopper, a space launch vehicle designed to take off horizontally and glide back to earth after placing its cargo in orbit (see [4-6]). The corresponding Navier-Stokes Equations are solved with FLOWer, a flow solver developed at the German Aerospace Center (DLR).

The code is parallelized with the CLIC-3D communication library which encapsulates all the MPI communication. As all information exchange is block oriented, the number of blocks limits the number of MPI tasks. When less MPI processes are started than blocks are used, FLOWer distributes the blocks over the processes according to the grid sizes in an effort to equalize computational load. Underneath the coarse-grained parallelization with CLIC-3D/MPI, all but one of the compute intense loop nests can be efficiently auto-parallelized using threads.

In table 1, the runtimes of a job running on the Sun Fire 6800 with six MPI tasks, each using 2 threads, is shown in detail. It can be seen that the compute time of the various processes differs by a factor of over two.

**Table 1. FLOWer on one Sun Fire 6800, 6 processes with 2 threads each, runtime of the single MPI processes in seconds**

Rank-ID	Compute Time	MPI Overhead
0	258.5	253.3
1	182.8	329.0
2	210.6	301.2
3	187.5	324.3
4	186.0	325.8
5	497.6	14.2

In table 2, we show the impact of adding more threads, with or without employing the DTB library. We noticed that an increase of the number of MPI tasks not necessarily improves performance, which probably has to do with the fact that a greater number of MPI tasks leads to a greater communication overhead overall. Whereas increasing the number of threads from 2 to 3 pays off in any case, which indicates that the loop-level parallelization scales well, the last measurement with 4 threads for each of the 6 MPI processes does not perform well, if DTB is turned on. This is most likely due to the fact that the system is slightly overbooked by 6 times 4 threads plus additional system processes running at the time of this test run on 24 physical CPUs.

**Table 2. FLOWer on one Sun Fire 6800, runtime with and without DTB in seconds**

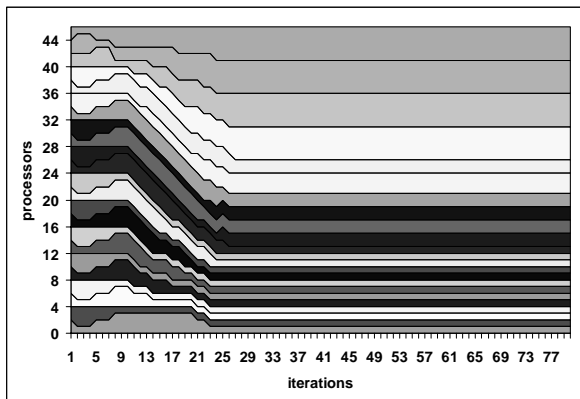
#processors	#MPI processes	#threads	without DTB	with DTB
12	6	2	499.8	350.6
14	7	2	606.9	450.7
16	8	2	511.0	317.0
18	6	3	408.8	286.3
21	7	3	474.3	347.1
24	6	4	363.2	314.0

However, more importantly, the use of the DTB library resulted in substantial improvements in any case. Given that FLOWer is a code used in production in many projects, these results show the benefit of hybrid parallel programming and the performance potential that can be gained from dynamic thread balancing schemes on large SMP nodes.

On the Sun Fire 15K we investigated the behaviour of the ccNUMA architecture. The Solaris 9 operating environment uses the first touch policy by default, if the Memory Placement Optimization (MPO) feature is enabled. The DTB mechanism suffers from the fact, that, if the number of threads is increased for a busy process, an additional thread might be started on a different processor board then slowing down the whole

process, because this thread has to access all its data remotely. As FLOWer uses a static mesh of grid points, our strategy was to turn off the first touch mechanism and use random placement instead for the first phase of the computation. Once the distribution of threads stabilized, we cause all data to move to the very board where it is accessed the next time. Unfortunately this mechanism is by no means standardized. We had to use the MPO APIs provided by Solaris 9.

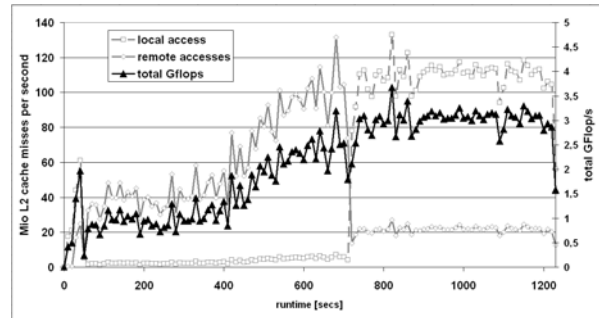
Figure 1 shows the distribution of threads to the 23 MPI processes over the iteration numbers. Each of these processes starts off with 2 threads, thus occupying 46 processors of the Sun Fire 15K in total. During the first 10 iterations the warm-up phase can clearly be identified. In the following 15 iterations the threads are shifted such that the MPI processes 20, 21, 22, and 23 on the upper part of the graph get 5 threads, whereas the MPI processes depicted on the lower part run with only one thread at the end. At about iteration 30 the DTB library detects that the thread distribution remains stable, triggers the “next touch” mechanism and turns off the steering.



**Fig. 1. FLOWer on one Sun Fire 15K: thread distribution over the iteration number**

Figure 2 nicely demonstrates the success of this strategy. The number of L2 cache misses satisfied by local and remote memory accesses has been measured with hardware performance counters and is plotted over the runtime. During the first 700 seconds there are by far more remote misses than local misses. Then the “next touch” mechanism is turned on and the DTB steering mechanism is turned off to save overhead and suddenly the situation completely changes. Now the number of local misses is much higher than the number of remote misses. The figure also contains a line indicating the speed of the calculation measured in

total GFlop/s. Once the memory is mainly accessed locally the speed increases by about 20 percent.



**Fig. 2. FLOWer on one Sun Fire 15K, random memory placement at the beginning and then explicit migration after about 700 seconds: local and remote cache misses and GFlop/s**

#### 4. Conclusions

By automatically varying the number of threads per MPI process based on timing results of the running program we offer an easy-to-use opportunity to improve the load balance of hybrid applications on clusters of shared-memory machines with fat nodes. The PMPI profiling interface of MPI allows us to easily capture relevant timing information, and as a result, a potential user only has to add a subroutine call in a few places of the code to trigger the dynamic thread balancing (DTB) mechanism.

Essential for the profitability of this technique was the control over the behaviour of the surplus threads such that they no longer block CPUs by spin-waiting. This has been achieved by a Solaris specific environment variable. Here a standardized way of controlling the behaviour of unused threads - for example as an additional parameter of the `omp_set_num_threads` runtime function - would be adequate.

Experimental results with the FLOWer CFD solver show that the use of the DTB library results in substantial improvements. The programming effort required to use the DTB library is minimal and it can be employed in a very flexible fashion as the number of additional threads is concerned.

First experiments taking the ccNUMA architecture of the Sun Fire 15K architecture into account revealed very positive results. Means to explicitly move data to where they are used are unfortunately not standardized (for example by an `omp nexttouch(variable list)` directive would be helpful), so that we had to employ a low level interface of Solaris 9.

In future work, we will investigate to what extent we can improve the robustness of the DTB mechanism during the warm-up phase and on loaded systems. We also plan to use the library in simulations with adaptive grid refinement.

## 5. References

1. Jost, G., Jin, H., an Mey, D. and Hattay, F.: Comparing the OpenMP, MPI and Hybrid Programming Paradigms on an SMP Cluster, NAS Technical Report NAS-03-019, NASA Ames Research Center, Moffet Field, CA, Novermber 2003. <http://www.nas.nasa.gov/Research/Reports/Techreports/2003/PDF/nas-03-019.pdf>
2. Bova, S.W., Breshears, C.P., Gabb, H., Kuhn, B., Magro, B., Eigenmann, R., Gaertner, G., Salvini, S. and Scott, H.: Parallel Programming with Message Passing and Directives. In: Computing in Science and Engineering, September 2001, pp. 22-37.
3. Rabenseifner, R.: Hybrid Parallel Programming on Parallel Platforms. EWOMP'03 - Fifth European Workshop on OpenMP, <http://www.rz.rwth-aachen.de/ewomp03/.omptalks/Tuesday/Session7/T01p.pdf>.
4. Hesse, M.K., Reinartz, B. and Ballmann, J.: Inviscid Flow Computation for the Shuttle-Like Configuration PHOENIX. Notes on Numerical Fluid Mechanics, Vol. 87, Eds. Chr. Breitsamter, B. Laschka, H.-J. Heinemann, R. Hilbig, Springer 2003, pp. 172-179.
5. Hesse, M.K., Reinartz, B. and Ballmann, J.: Numerical Investigation of the Shuttle-Like Configuration PHOENIX. High Performance Computing in Science and Engineering 2002, Ed. E. Krause, W. JÄager, Springer Verlag, ISBN 3-540-43860-2, pp. 379-390.
6. Hesse, M.K., Reinartz, B. and Ballmann, J.: Numerical Investigation of a Reusable Space Transportation System. In: Proceedings of the 3rd International Symposium on Atmospheric Reentry Vehicles and Systems, Arcachon/France, 24-27 March 2003.
7. Spiegel, A., an Mey, D. and Bischof, C.: Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing, PARA'04 - Workshop on the State-of-the-Art in Scientific Computing, June 2004. Proceedings to be published in the Springer series Lecture Notes in Computer Science
8. Solaris Memory Placement Optimization and Sun Fire Servers, Technical White Paper, March 2003. [http://www.sun.com/servers/wp/docs/mpo\\_v7\\_CUSTOMER.pdf](http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf)