What multilevel parallel Programs do when you are not watching: A Performance Analysis Case Study comparing MPI/OpenMP, MLP, and nested OpenMP

Gabriele Jost[1]*, Jesus Labarta[2] and Judit Gimenez[2]

[1]NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 USA
gjost@nas.nasa.gov

[2]European Center for Parallelism of Barcelona-Technical University of Catalonia (CEPBA-UPC), cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain
{jesus,judit}@cepba.upc.es

1 Extended Abstract

With the current trend in parallel computer architectures towards clusters of shared memory symmetric multi-processors, parallel programming techniques have evolved that support parallelism beyond a single level. When comparing the performance of applications based on different programming paradigms, it is important to differentiate between the influence of the programming model itself and other factors, such as implementation specific behavior of the operating system (OS) or architectural issues. Rewriting a large scientific application in order to employ a new programming paradigm is usually a time consuming and error prone task. Before embarking on such an endeavor it is important to determine that there is really a gain that would not be possible with the current implementation. A detailed performance analysis is crucial to clarify these issues.

The multilevel programming paradigms considered in this study are hybrid MPI/OpenMP, MLP, and nested OpenMP. The hybrid MPI/OpenMP approach is based on using MPI [7] for the coarse grained parallelization and OpenMP [9] for fine grained loop level parallelism. The MPI programming paradigm assumes a private address space for each process. Data is transferred by explicitly exchanging messages via calls to the MPI library. This model was originally designed for distributed memory architectures but is also suitable for shared memory systems.. The second paradigm under consideration is MLP which was developed by Taft [11]. The approach is similar to MPI/OpenMP, using a mix of coarse grain process level parallelization and loop level OpenMP parallelization. As it is the case with MPI, a private address space is assumed for each process. The MLP approach was developed for ccNUMA architectures and explicitly takes advantage of the availability of shared memory. A shared memory arena which is accessible by all processes is required. Communication is done by reading from and writing to the shared memory. Libraries supporting the MLP paradigm usually provide routines for process creation, shared memory allocation, and

* The author is an employee of Computer Sciences Corporation.

process synchronization. The third paradigm employed in our study is the usage of nested OpenMP directives. Even though the nesting of parallelization directives is permitted by the OpenMP standard, it is not supported by many compilers. The NanosCompiler [3] was developed to show the feasibility of exploiting nested parallelism in OpenMP and is used in our study. The NanosCompiler accepts Fortran-77 code containing OpenMP directives and generates plain Fortran-77 code with calls to the NanosCompiler thread library NthLib [6]. NthLib supports multilevel parallel execution such that inner parallel constructs are not being serialized. The programming model supports several extensions to the OpenMP standard allowing the user to create groups of threads and to control the allocation of work to the participating threads.

We describe the performance analysis of the multi-zone versions of the NAS Parallel Benchmarks the NPB-MZ [12]. The purpose of the NPB-MZ is to capture the multiple levels of parallelism inherent in many full scale CFD applications. Multi-zone versions of the well known NAS Parallel Benchmarks [2] LU, BT, and SP were developed by dividing the discretization mesh into a two-dimensional tiling of three-dimensional zones. Within all zones the LU, BT, and SP problems are solved to advance the time-dependent solution. The same kernel solvers are used in the multi-zone codes as in the single-zone codes. Exchange of boundary values takes place after each time step. Reference implementations employing the MPI/OpenMP and the MLP programming paradigm are part of the benchmarks distribution. A discussion of the performance characteristics of these codes is presented in [5]. The nested OpenMP implementation we used in our study is based on NanosCompiler extensions and is discussed in [1].

Our tests are executed on an SGI Origin 3000 located at the NASA Ames Research Center. The SGI Origin 3000 is a ccNUMA architecture with 4 CPUs per node. The CPUs are of type R12K with a clock rate of 400 MHz, 2 GB of local memory per node, and 8 MB of L2 cache. The MLP implementations use the SMPlib library as described in [4]. The MIPSpro 7.4 Fortran Compiler is used to compile the hybrid codes and the NanosCompiler for the nested OpenMP code. The compiler options -mp -O3 and -64 are set in all cases.

We use the Paraver [10] performance analysis system which is being developed and maintained at CEPBA-UPC. It consists of a graphical user interface to obtain a qualitative view of the program execution and an analysis module for a detailed quantitative performance analysis. The Paraver tracing package OMPItrace [8] provides a simple but very flexible format. Traces are composed of state records, events, and communications, each with an associated time stamp.

We ran the different implementations of LU-MZ and BT-MZ for 20 iterations of various problem classes. In Figure 1 we show their scalability for benchmark class B which has an aggregate size of 304x208x17 grid points The reported performance is the best that was achieved over different combinations of processes and threads or thread groups in case of nested OpenMP. For LU-MZ the scalability of the MLP based implementation is clearly superior to MPI/OpenMP and nested OpenMP. For BT-MZ, the MLP and nested OpenMP show similar scalability while MPI/OpenMP lags behind for a large number of CPUs.

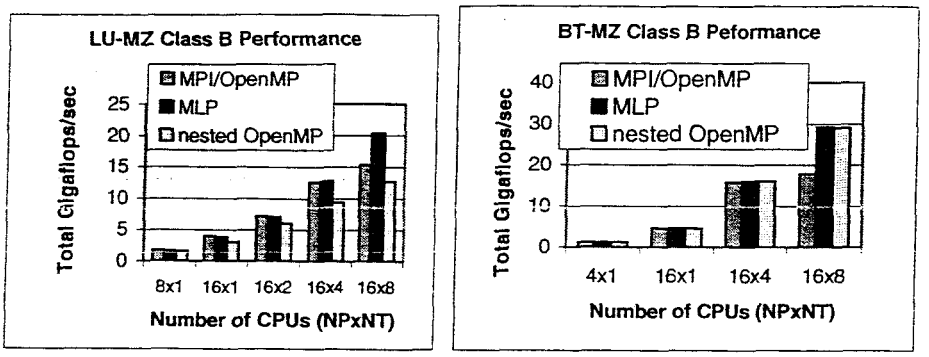

Figure 1: Performance of LU-MZ and BT-MZ Class B. The number of CPUs is indicated as the number of processes (NP) times the number of threads (NT).

We will first examine LU-MZ and investigate the performance difference between MLP and MPI/OpenMP. The performance difference between MPI/OpenMP and MLP on 128 CPUs can also be observe on 64 CPUs, employing 8 processes with 8 threads each. We obtained traces for the smaller configuration and computed various statistics. In summary, the statistics indicate the following:

- Some of the subroutines take considerably (50%) more time in MPI/OpenMP than in MLP.
- The routines taking more time in MPI/OpenMP are computation bound. Communication time is therefore not the issue for the time difference.
- There is no significant difference (< 0.001%) in the number of issued instructions between the MPI/OpenMP and MLP in the time consuming routines. Computational complexity is therefore not an issue for the time difference.
- The difference in the number of L2 cache misses is slightly in favor of MPI/OpenMP which has about 1% less misses.

A clue to the reason for the performance difference can be obtained by examining the duration of the OpenMP workshares in both implementations. An example is shown in Figure 2 which displays a zoom into a timeline view of the duration of workshares for 2 processes with 8 threads each. The duration of the workshares is indicated by a gradient color scale. The scale ranges form 30 ms (light shading) to 60 ms (dark shading) to make the difference between the different threads apparent. The MPI/OpenMP code clearly shows a bimodal pattern. depending on the thread number. Four of the eight threads of each process show shorter and four show longer durations. Those with short durations are very similar to the MLP case. Those with the long durations are responsible for the increased amount of time in certain subroutines in the MPI/OpenMP implementations. The difference in the number of instructions and L2 cache misses across the threads in the MPI/OpenMP code is less than 3% and can not account for the time difference. When jointly looking at the time taken by the OpenMP workshares within a given subroutine, the number of instructions and the number of L2 misses across threads we can estimate that the same number of L2 misses take a fairly different time on different threads. An explanation for this is, that some threads re-

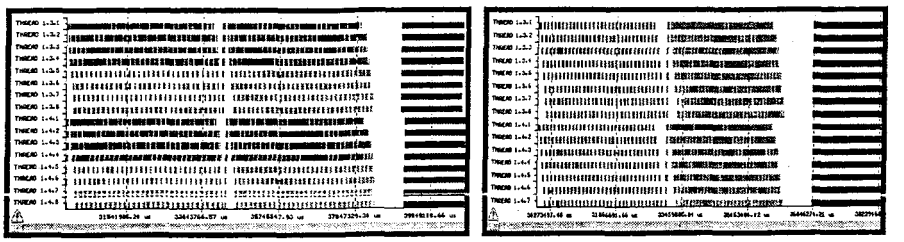

Figure 2: *Timeline view of the work sharing duration within MPI/OpenMP (left) and MLP (right) code for LU-MZ.* The view zooms in on 2 processes with 8 threads each. The thread number is indicated on the vertical axis, the time on the horizontal axis. Light shading indicates OpenMP workshares with short durations, darker shading indicates longer workshares. *MPI/OpenMP shows a bimodal pattern depending on the thread, which is not present in MLP*

quire remote memory access due to the way the data is placed relative to the CPU that they run on.

When combining MPI and OpenMP on the SGI Origin, care has to be taken how to place the threads onto the CPUs. By default the OS will place the MPI processes onto consecutive CPUs. When new threads are being forked they may end up running on CPUs far away from the master MPI process, which can potentially decrease the performance due to remote memory access time. There is the possibility for the user to specify gaps between the MPI processes to allow for future threads, but it seems that a gap of 8, necessary for the 8x8 run, was not handled correctly, either by the user or by the OS. The traces indicate that four of the threads are placed correctly on consecutive CPUs, but the other four are placed further away. The problem does not occur in the MLP code. The MLP programming paradigm was designed for ccNUMA architectures. During the start-up phase the MLP library issues system calls which pin a thread to a particular CPU for the duration of the run to assure efficient memory access. When using the same system call within the MPI/OpenMP implementation the performance was very similar to MLP (see Figure 4).

Next we investigate the lack of scalability of LU-MZ using nested OpenMP. We gathered a trace employing 16 groups of 8 threads. Figure 4 shows a timeline view that identifies the parallel functions executed by each thread. We can identify that 16 threads execute at the outermost parallelism level, each of them generating work for 8 threads. The first two iterations are very long and imbalanced. The following iterations are much faster and are relatively balanced. There is no algorithmic reason for a workload imbalance within the first two iterations, nor for the fact that they should consume more time than the following iterations. We suspect that due to the interaction between the NthLib and the OS it takes several iterations until the data is placed onto the appropriate memory modules. The BT-MZ benchmark implementation differs from LU-MZ in that it executes one time step before the actual iteration loop. The purpose is to place the data appropriately before the timing begins. We have added the execution of two time steps before the timed iteration loop to the LU-MZ code.

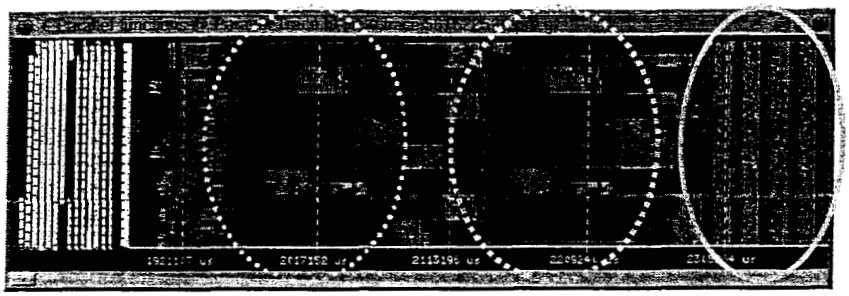

Figure 3: Time line view of LU-MZ using nested OpenMP. The view shows the time spent in different compiler generated parallel functions. Different shadings indicate different parallel functions. The dashed lines mark the first two iterations, the solid lines mark 5 of the following iterations

In Figure 4 we show the timings of Figure 1 and the timings obtained after adding the call to pin-to-node to MPI/OpenMP and after adding 2 time steps before the iteration loop to the nested OpenMP code. We conclude that after removing OS and runtime library effects, the performance of the three implementations is very similar. The performance increase was obtained with only minor changes to the user code. In the full paper we plan to include timing results for other benchmark classes and discuss load balancing issues.

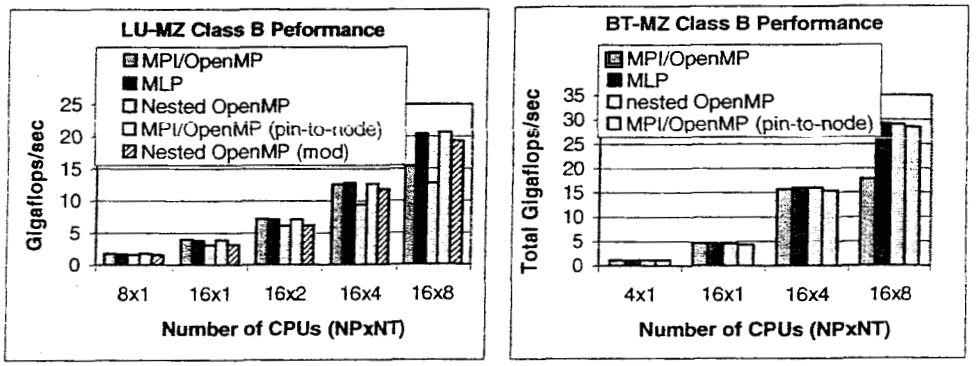

Figure 4: Performance of LU-MZ and BT-MZ Class B. The number of CPUs is indicated as the number of processes (NP) times the number of threads (NT)

Acknowledgements

This work was supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/AMTI, by the Spanish Ministry of Science and Technology, by the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA).

References

1. E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost, *Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications,* to appear in the Proceedings of IPDPS 2004, Santa Fe, New Mexico, USA, April 2004
2. D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0, RNR-95-020,* NASA Ames Research Center, 1995
3. M. Gonzalez, E. Ayguade, X. Martorell and J. Labarta, N. Navarro and J. Oliver. *NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP, Concurrency: Practice and Experience. Special issue on OpenMP. vol. 12, no. 12.* pp. 1205-1218. October 2000
4. H. Jin, G. Jost, *Performance Evaluation of Remote Memory Access Programming on Shared Memory Parallel Computer Architectures*, NAS Technical report NAS-03-001, NASA Ames Research Center, Moffett Field, CA, 2003
5. H. Jin, R. F. Van der Wijngaar, *Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks*, to appear in the Proceedings of IPDPS04, Santa Fe, New Mexico, USA, April 2004
6. X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta. *Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors.* 13th International Conference on Supercomputing (ICS'99), Rhodes (Greece). pp. 294-301. June 1999
7. MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich
8. OMPItrace User's Guide, https://www.cepba.upc.es/paraver/manual_i.htm
9. OpenMP Fortran Application Program Interface, http://www.openmp.org/.
10. Paraver, http://www.cepba.upc.es/paraver
11. J. Taft, *Achieving 60 GFLOP/s on the Production CFD Code OVERFLOW-MLP*, Parallel Computing, 27 (2001) 521
12. R. F. Van Der Wijngaart, H. Jin, *"NAS Parallel Benchmarks, Multi-Zone Versions,"* NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003