# Performance Analysis of Multilevel Parallel Applications on Shared Memory Architectures

Gabriele Jost[*], Haoqiang Jin
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 USA
{gjost,hjin}@nas.nasa.gov

Jesus Labarta, Judit Gimenez, Jordi Caubet
European Center of Parallelism of Barcelona-Technical University of Catalonia (CEPBA-UPC)
cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain
{jesus,judit,jordics}@cepba.upc.es

## Abstract

*In this paper we describe how to apply powerful performance analysis techniques to understand the behavior of multilevel parallel applications. We use the Paraver/OMPItrace performance analysis system for our study. This system consists of two major components: The OMPItrace dynamic instrumentation mechanism, which allows the tracing of processes and threads and the Paraver graphical user interface for inspection and analyses of the generated traces. We describe how to use the system to conduct a detailed comparative study of a benchmark code implemented in five different programming paradigms applicable for shared memory*

## 1 Introduction

With the advent of parallel hardware and software technologies users are often faced with the challenge to choose a programming paradigm that is best suited for the underlying computer architecture. 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. Multiple levels of parallelism can be achieved by the use of processes, multithreading, and a mixture of both. Which programming paradigm is the best will depend on the nature of the given problem, the hardware architecture, and the available software. When comparing the performance of applications based on different programming paradigms it is important to understand how the performance is influenced by what aspect of the paradigm. It is important to identify and quantify the performance metrics of a certain paradigm.

Quantification of performance characteristics can be only be achieved by a detailed analysis. The Paraver [14] visualization and analysis tool was developed with the goal to provide the user with means to obtain a qualitative global perception of the application behavior as well as a detailed quantitative analysis of program performance. Paraver allows the user to visually inspect trace files that have been collected during program execution. Traces are composed of state transitions, events, and communications, each with an associated time stamp. These three elements allow generating traces that capture the behavior of a programs execution. For our study we used the OMPTtrace module, which is optionally available with the Paraver distribution. The OMPItrace module allows dynamic instrumentation and tracing of applications with multiple levels of parallelism.

*The author is an employee of Computer Sciences Corporation

To this day there a very few published results containing detailed performance analysis data for multilevel parallel program. In our study we compare different implementation of one of the NAS Parallel Benchmarks [1] employing single and multiple levels of parallelism. We demonstrate how to conduct a detailed performance analysis to determine the impact of the programming paradigm on the scalability of the code.

The rest of the paper is structured as follows: In Section 2 we give a brief discussion of parallel programming paradigms applicable to shared memory architectures and identify their performance metrics. In Section 3 we explain how the Paraver/OMPItrace performance analysis system supports the different levels of parallelism. In Section 4 we perform a comparative case study using different implementations of a CFD benchmark kernel. In Section 5 we briefly discuss related work and summarize our conclusions in Section 6.

## 2 Parallel Programming Paradigms for Shared Memory Architectures

Most shared memory architectures provide hardware support for a cache coherent globally shared address space. Process communication is usually supported by software but leverages the hardware support for the shared address space. This provides support for various programming paradigms.

### 2.1 Process Level Parallelism

In our study we assume that each process has its own local memory. A well understood programming paradigm for process level parallelism is message passing. The computational work and the associated data are distributed among a number of processes. If a process needs to access data located in the memory of another process, it has to be communicated via the exchange of messages. MPI (Message Passing Interface) [10] is a widely accepted standard for writing message-passing programs.

The Shared Memory Access (SMA) programming paradigm is also based on the concept of processes with their own local memory, but it separates the communication and synchronization step. In this programming model, a special memory area is allocated which is accessible to all processes. Exchange of data is achieved by writing to and reading from the shared memory buffer. Data transfer between two processes is performed by only one side and does not require a matching operation by the other process. The correct ordering of memory access has to be imposed by the user through explicit synchronization. The MLP library [18], which was developed at the NASA Ames Research Center, supports this programming paradigm by providing routines for process generation, shared memory allocation, and process synchronization.

### 2.2 Thread Level Parallelism

Parallel programming on a shared memory machine can take advantage of the globally shared address space. Compilers for shared memory architectures usually support multi-threaded execution of a program. Loop level parallelism can be exploited by using compiler directives such as those defined in the OpenMP standard [13]. OpenMP provides a fork-and-join execution model in which a program begins execution as a single process or thread. This thread executes sequentially until a parallelization directive for a parallel region is found. At this time, the thread creates a team of threads and becomes the master thread of the new team. All threads execute the statements execute the statements until the end of the parallel region. Work-sharing directives are provided to divide the execution of the enclosed code region among the threads. All threads need to synchronize at the end of parallel constructs, they also may synchronize at the end of each work-sharing construct or at specific points enclosed by the parallel construct. Data transfer between threads is done by direct memory references. In a nest of loops, parallelization directives can be applied to several of the loop levels. This allows for the possibility to have groups of threads working on different chunks of data. In our study we consider nested OpenMP parallelism as mulitlevel parallelism, since grouping the threads imposes a hierarchy amongst them. Even though the nesting of parallelization directives is permitted by the OpenMP standard, it is at this point not supported by many compilers. The NanosCompiler [3] has been developed to show the feasibility of exploiting nested parallelism in OpenMP and is used in our study.

### 2.3 Hybrid Parallelism

Process level parallelism and thread level parallelism can be combined into a hybrid paradigm to exploit parallelism beyond a single level. The main thrust of the hybrid parallel paradigm is to combine process level coarse grain parallelism, which is obtained for example in domain decomposition and fine grain parallelism on a loop level which is achieved by compiler directives. The hybrid MPI/OpenMP approaches has been used in various applications ([8], [9]).

The SMA programming model also allows a combination of processes and threads. We refer to the hybrid SMA/OpenMP programming paradigm as MLP (Multi-Level-Parallelism). MLP has been successfully applied at the NASA Ames Research Center [18],

## 3 Support for Performance Analysis of Multilevel Parallel Programs

A trace-based visualization and analysis environment consist of two basic components: the tracing package and the visualization tool. The power of the system depends on the individual capabilities of each component as well as on the proper interoperation between them. The tracing tool must be able to capture events that accurately characterize the behavior of the program. The analysis tool has to post-process the trace information in order to display useful views and statistics.

### 3.1 The OMPItrace Module

In our study we used OMPItrace [12] to instrument all the programming paradigms targeted by our analysis. The basic OMPItrace mechanism traces OpenMP, MPI and hybrid MPI/OpenMP binaries without requiring source code modification or linking to special libraries.

The SGI version of OMPItrace uses DITools [16] as the dynamic instrumentation mechanism. Calls to the dynamically linked OpenMP and MPI run-time libraries are intercepted by properly modifying the Global Offset Table (GOT) at load time. Events are emitted into the trace on entry and exit of MPI calls along with the information of message size and source/destination. Entry and exit of OpenMP parallel regions and work-sharing constructs are also tagged. In this programming model, the code regions that correspond to the body of the parallel regions/loops are outlined by the compiler and encapsulated as parallel routines. OMPItrace intercepts entry and exit to these routines as well as the run time calls that assign work when called by these outlined routines. Besides tracing events, OMPItrace can read two hardware counters and emit their values to the trace. Reading the hardware counters poses a certain overhead, as it requires a system call. If hardware counters are not read, the overhead of the instrumentation at one probe is about

1 microsecond. Reading the hardware counters increases the overhead by about 20-23 microseconds.

The Paraver object model is structured in a three level hierarchy: Application, Task, and Thread. Each record is tagged with these three identifiers. This approach allows Paraver to support nested parallel programming models

For this study, OMPItrace had to be extended to support process generation and synchronization mechanisms different form those originally used for MPI and OpenMP. To support SMA and MLP it was necessary to add extensions to dynamically intercept calls to the UNIX fork routine. We also added support for the state records that indicate MLP synchronization primitives and provided an instrumented version of the MLP library to allow tracing.

## 3.2 Trace File Views

The traces collected during the execution of a program contain a wealth of information, which as a whole is overwhelming. We need to be able to screen out information to gain visibility of a critical subset of the data. In order to understand the impact of various aspects of hardware, operating system (OS), and program structure on the performance, it is also necessary to correlate extracted subsets of information with each other. This can be done through timeline graphical displays or by histograms and statistics. Paraver provides great the flexibility in designing displays hat are suitable for a particular problem. The timelines can be displayed at the level of individual threads but also at the level of processes. In he latter case, the values for each thread are combined to produce a value for the process. A user can specify through the Paraver GUI how to compute a given performance index from the records in the trace and then save it as a configuration file. These configuration files can then be used by to immediately pop up a view of the selected performance index. For example, it is possible to show when MPI calls are made, when parallel functions are being executed by each thread or what the MIPS or cache miss rates are for a given time.

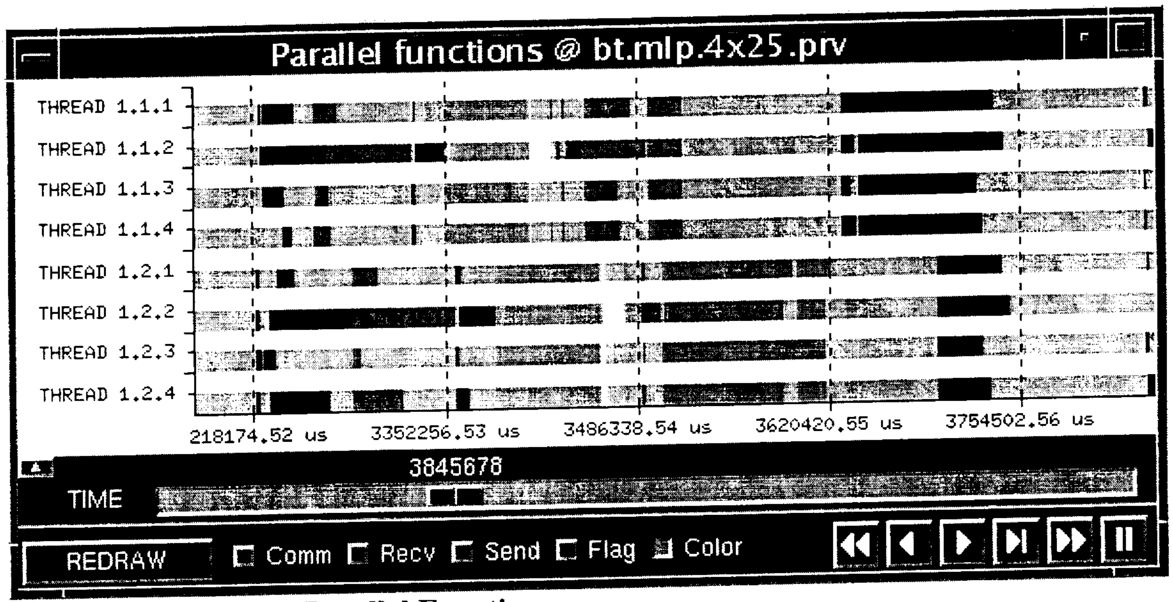
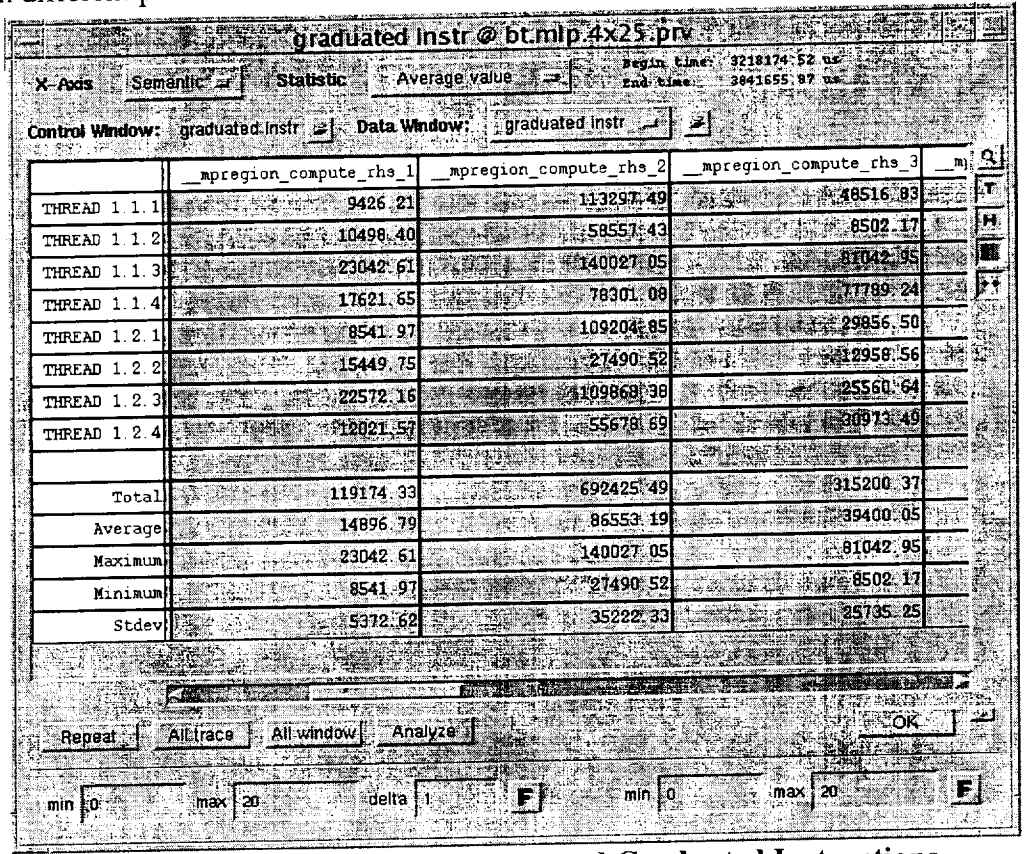


**Figure 1: Timeline of Parallel Functions**

Figure 1 shows the parallel functions view of an MLP. The different colors indicate the parallel regions with light blue representing time that is spent outside of parallel regions.

Another interesting view shows whether a thread is doing useful computation or not. To obtain such a view we mask time in process synchronization primitives, communication routines, time in OpenMP idle loops, barriers between successive OpenMP work sharing constructs and finally the synchronization waits implemented by the programmer through busy wait loops and the FLUSH directive. Once identified time intervals where really useful computation is done, views showing MIPS or duration of those regions can be derived.

The main quantitative analysis module in Paraver builds a table where for each thread profile data or histograms are shown. Typical statistics such as average value, maximum or standard deviation can be applied to a user-selected region of any of the timeline views previously defined. The result is a very powerful profiling capability. Example statistics that can be reported are time or miss ratio for each thread within each user function or parallel routine but also histogram of duration of a given MPI call or even correlation between cache misses and duration of a given MPI call. In Figure 2 we show the mapping of the number of graduated instructions for each thread within different parallel functions.



| | __mpregion_compute_rhs_1 | __mpregion_compute_rhs_2 | __mpregion_compute_rhs_3 | __m |
|---|---|---|---|---|
| THREAD 1.1.1 | 9426.21 | 113297.49 | 48516.83 | |
| THREAD 1.1.2 | 10498.40 | 58557.43 | 8502.17 | |
| THREAD 1.1.3 | 23042.61 | 140027.05 | 81042.95 | |
| THREAD 1.1.4 | 17621.65 | 78301.08 | 77789.24 | |
| THREAD 1.2.1 | 8541.97 | 109204.85 | 29856.50 | |
| THREAD 1.2.2 | 15449.75 | 27490.52 | 12958.56 | |
| THREAD 1.2.3 | 22572.16 | 109868.38 | 25560.64 | |
| THREAD 1.2.4 | 12021.57 | 55678.69 | 30913.49 | |
| | | | | |
| Total | 119174.33 | 692425.49 | 315200.37 | |
| Average | 14896.79 | 86553.19 | 39400.05 | |
| Maximum | 23042.61 | 140027.05 | 81042.95 | |
| Minimum | 8541.97 | 27490.52 | 8502.17 | |
| Stdev | 5372.62 | 35222.33 | 25735.25 | |

**Figure 2: Analysis of Parallel Functions and Graduated Instructions**

## 4 A Comparative Case Study

We used the BT benchmark from the NAS Parallel Benchmarks (NPB) [1] for our comparative study. The BT benchmark solves three systems of equations resulting from an approximate factorization that decouples the x, y and z dimensions of the 3-dimensional Navier-Stokes equations. These systems are block tridiagonal consisting of 5×5 blocks. Each spatial dimension is alternatively swept as depicted in Figure 3.

We evaluated three parallelization approaches employing a total five programming models. We used two pure process level parallel, two hybrid parallel, and one nested OpenMP implementation of the same benchmark.

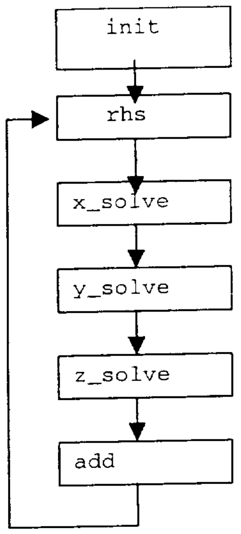


**Figure 3: Structure of the BT benchmark**

### 4.1 Description of the Different Benchmark Implementations

The MPI implementation of the BT employs a multi-partition scheme [2] in 3-D to achieve good load balance and coarse-grained communication. In this scheme, processors are mapped onto sub-blocks of points of the grid in a special way such that the sub-blocks are evenly distributed along any direction of solution. The blocks are distributed such that for each sweep direction the processes can start working in parallel. Throughout the sweep in one direction, each processor starts working on its sub-block and sends partial solutions to the next processor before going into the next stage. An example for one sweep direction of a 2-D case is illustrated in Figure 4. Communications occur at the *sync points* as indicated by gray lines in Figure 4. We indicate the number of the process who owns the data within each square. In the actual implementation separate routines are used to form the left-hand side of the block tridiagonal systems before these systems are solved. A number of five- and six-dimensional work arrays are used to hold intermediate results.

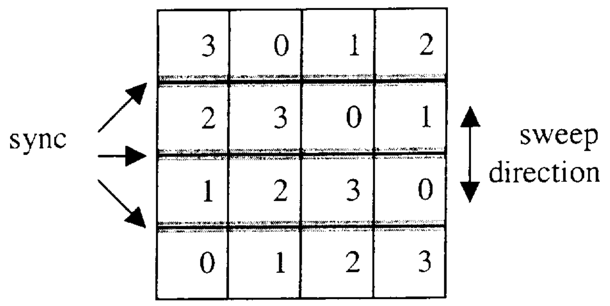


**Figure 4: The multi-partition scheme in 2-D.**

In the SMA implementation we employ the same algorithm, except that communications are handled by data exchange through the shared memory buffers and proper synchronization primitives.

The OpenMP implementation is based on the version described in [5]. The implementation is similar to that of the message-passing version, but the large working arrays are significantly reduced in size and a couple of the computational intensive loops are fused. The code that we used in our study contains nested OpenMP directives, which were automatically generated using the CAPO [6] parallelization tool. It also contains OpenMP extensions for nested parallelism supported by the NanosCompiler [3] In our study we used the NanosCompiler for the nested OpenMP since nested OpenMP parallelism was not supported by the commercial SGI compiler. The automatic generation of nested OpenMP directives using CAPO is described in [7].

The hybrid MPI/OpenMP and MLP implementations are also based on the versions described in [5], however, the memory requirements are somewhat higher than in the pure OpenMP code, since we have to provide the workspace for a one dimensional data distribution. The coarse-grained parallelization was achieved by using the CAPTools [4] parallelization tool to generate a message-passing version with a one-dimensional domain decomposition in the z-dimension. For the MLP implementation communications are handled by usage of the shared memory buffer. The OpenMP directives we inserted by using CAPO on the y-dimension. Since the data is distributed in the z-dimension, the call to `z_solve` requires communication within a parallel region as depicted in Figure 5. The routine `y_solve` contains data dependences on the y-dimension, but can be parallelized by pipelined thread execution.

In summary, we use three different parallel algorithms for the BT benchmark. SMA and MPI use the same algorithm, but different means of data transfer. MLP and MPI/OMP use the same algorithm, but different means of data transfer. The nested OpenMP implementation uses a third algorithm.

```
!$omp parallel
!$omp do
   do j=1,ny
     call receive
     do k=k_low,k_high

       do i=1,nx
       rhs(i,j,k)=
          ... rhs(i,j,k-1)
    enddo
   enddo
   call send
  enddo
```

**Figure 5: Code fragment of hybrid routine z-solve**

### 4.2 Performance Indices

We will use the following abbreviations for the various programming models under consideration:

- **MPI:** Process level parallelism employing message passing based on MPI [10].
- **SMA**: Process level parallelism employing shared memory buffers for data transfer.
- **OpenMP:** Thread level parallelism based on OpenMP [13]
- **MLP:** Hybrid process and thread level parallelism employing SMA and OpenMP.
- **MPI/OMP**: Hybrid process and thread level parallelism employing MPI and OpenMP.

In order to compare and quantify the performance differences, we consider the following metrics, not all of which are applicable to all programming models. By useful instructions we mean instructions not spent on thread management, process synchronization, and communication.

- **MEM:** Number of L2 cache misses while executing useful instructions reported in millions.
- **INST:** Number of useful instructions reported in billions.
- **COMM**: Percentage of time spent in MPI routines for point-to-point or global communication
- **SYNC:** Percentage of time spent in process synchronization. This includes the time in MPI barriers and wait operations and time spent in MLP synchronization routines.
- **THREADM:** Percentage of time spend in thread management, i.e. thread fork and join, OpenMP thread synchronization, user introduced synchronization for pipelined thread execution, thread idle time.

These simple metrics will not give a detailed explanation of a particular performance problem. They will, however, give hints on where further analysis is required. Quantifying and comparing these characteristics require detailed measurements of system influence and hardware counters.

## 4.3 Comparing the Performance of Different Paradigms

We ran the various implementations of BT on an SGI Origin 3000. We used the SGI compiler for all implementations. For the nested OpenMP code we used the NanosCompiler [3], which is based on the SGI compiler, but supports nested OpenMP parallelism. All our timings where run in batch mode with the appropriate number of CPUs. In Figure 6 we show the execution times for various combinations of processes and threads for our hybrid implementations and for different choices of thread groupings in the nested OpenMP implementation. For the hybrid codes we indicate the nesting level as NPxNT, where NP is the number of processes and NT is the number of threads. For the nested OpenMP code we indicate the nesting level as NGxNT, where NG is the number of groups of threads and NT is the number of threads within one group.

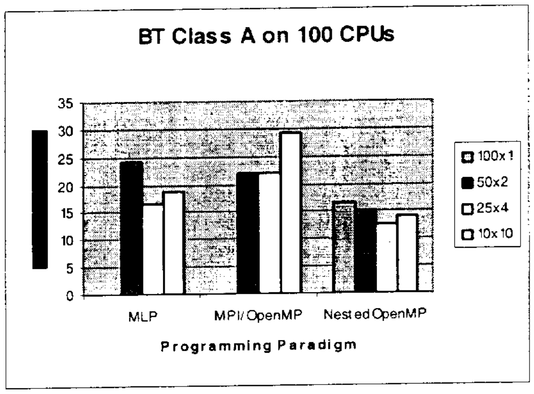


**Figure 6: Timings for different nesting levels.**

The speedup from 1 to 144 CPUs for all implementations is shown in Figure 7. For hybrid codes and the nested OpenMP code we report the best times over various nesting levels. We calculate the speedup by the ratio of the execution time on N CPUs and the fastest time achieved on 1 CPU. The fastest single CPU run was achieved with the OpenMP code. Therefore we report a speedup of the MPI implementation on 1 CPU as 0.5, indicating that the MPI code takes twice as long as the OpenMP code on 1 CPU. For the multilevel parallel implementations we tested various nesting levels and report the speedup for the best configuration. As can be seen, the scalability of the different implementations varies widely. The process level parallel implementations outperform the hybrid and OpenMP implementations. To understand why this is the case, we will conduct a detailed performance analysis using Paraver and OMPItrace.

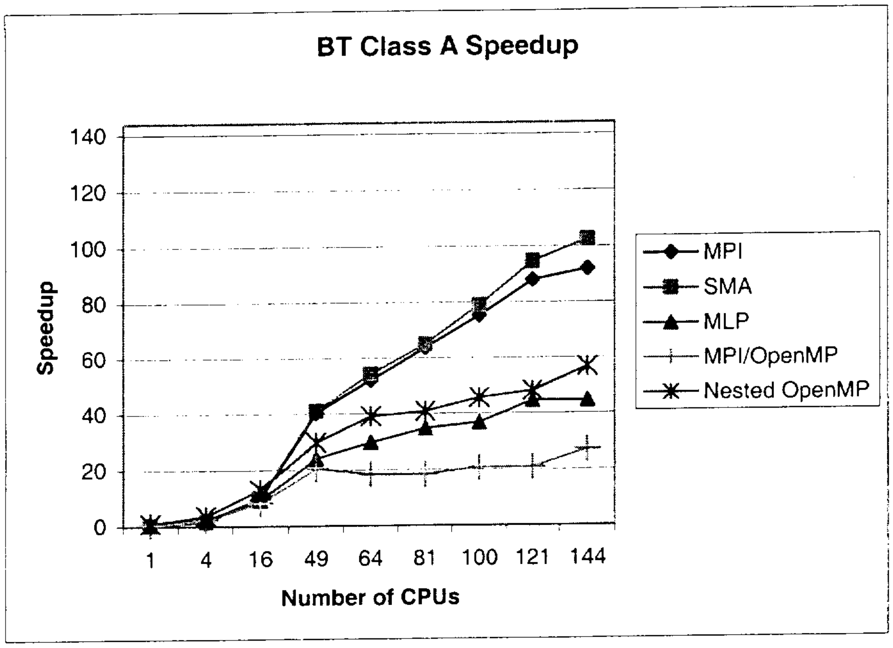


**Figure 7: Speedup for different paradigms**

For our analysis we obtained traces for all implementations. The SGI Origin allows tracing two hardware counters at a time. We traced graduated instructions and L2 cache misses. We collected the traces without and with hardware counters Tracing without hardware counters has minimal probe perturbation and should report detailed patterns of the temporal behavior of the application. Tracing with hardware counters introduces a bit more of overhead but provides extra detailed information. Based on the traces we can display relevant performance views and based on them we can compute some statistics. Statistics that do not depend on the hardware counters are computed on the trace without counters to reduce possible instrumentation overheads. We used the analysis module to calculate and average useful instruction and L2 miss rate.

We chose a run with 100 CPUs and the class A problem size for our investigations In Table 1 we summarize the statistics for the different benchmark implementations. The values were obtained by applying the analysis module. We applied the Paraver analysis module to the state view in order to determine the statistics COMM, SYNC, and THREADM. The main performance impact comes from time spent outside of useful calculations.

| | MPI | SMA | MLP | MPI/OMP | OMP |
|---|---|---|---|---|---|
| COMM | 14% | N/A | N/A | 7% | N/A |
| MEM | .45M | 0.56M | 3.78M | 5.12M | 1.01 M |
| INST | 1.2B | 1.39B | 1.55B | 1.38 B | 1.27 B |
| SYNC | 25% | 12% | 5.5% | 70% | N/A |
| THREADM | N/A | N/A | 65% | 21% | 56% |

**Table 1: Performance metrics for a 100 CPU run**

As a first step, we want to determine why the SMA code scales better than the MPI code, in spite of the fact that the same algorithm is being used. As can be seen from Table 1 SMA spends less amount of time in communication and synchronization compared to MPI. In Table 2 we report the average duration per call for the four main time consuming routines. The timings correspond to the combination with the best performance, employing 25 processes and 4 threads each.

| | x_solve | y_solve | z_solve | rhs |
|---|---|---|---|---|
| MPI | 13.844 | 13.189 | 13.877 | 19.240 |
| SMA | 10.315 | 10.663 | 11.086 | 7.705 |

**Table 2: User function timings (in ms) inclusive MPI and MLP calls**

A view of the trace that exposes relevant information regarding the actual computations in the application shows the user function being executed but taking out all the time inside MPI calls. These timings are reported in Table 3.

| | x_solve | y_solve | z_solve | rhs |
|---|---|---|---|---|
| MPI | 9.715 | 10.098 | 10.723 | 4.343 |
| SMA | 9.808 | 10.123 | 11.003 | 7.024 |

**Table 3: User function timings (in ms) exclusive MPI and MLP calls**

Routine `rhs` is communication bound as can be inferred from the important difference in duration of user mode duration. The routine performs the exchange of boundary data before each sweep. Each process performs a series of `MPI_Isend` and `MPI_Irecv` followed by an `MPI_Waitall`. In Figure Figure we show the timelines of MPI calls for a 100 process run. A striking effect is that the long time in MPI is not just in the `MPI_waitall`. The view of the MPI routines exposes some very long calls to `MPI_Isend` and `MPI_Irecv`. They correspond to marked dense areas in Figure 8. This shows that even if one may think that the use of non-blocking MPI routines decouples communication and synchronization it does not seem to be totally true. The duration of a given call seems to depend on whether the matching call has been posted. In case of many communication requests there is the possibility of contentions in the internals of the MPI library. The result is that a process posting several consecutive calls may be delayed in one of them, thus delaying the posting of successive calls. The effect propagates through the process dependence chain of the application in very intricate ways.

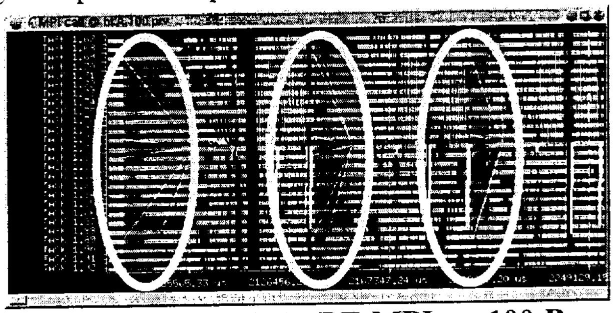


**Figure 8: MPI calls in BT MPI on 100 Processes**

The SMA implementation does not suffer from this effect. Data transfer takes place within the use code itself, by reading from and writing to the shared memory buffer. This increases the computation time somewhat, but yields in general better scalability.

Now we can go on to address the question of what limits the scalability of the multilevel parallel codes. In order to answer this question, we apply the view of useful computations described in Section 3. Based on this view it is possible to obtain the percentage of time spent in useful instruc-

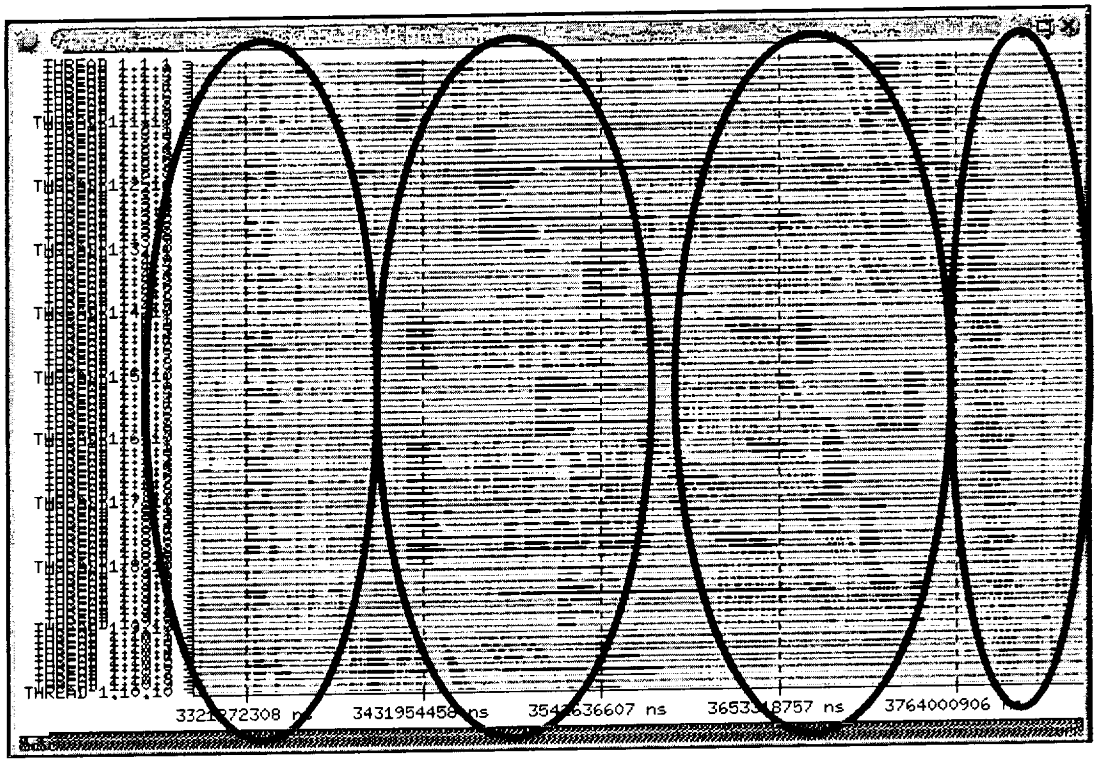


**Figure 9: Flow of Useful Computations in BT MLP.**

tions within each user routine. As an example we will demonstrate a detailed analysis of the MLP code and then summarize our findings for the other multilevel implementations.

For BT MLP we computed that only 12% of the time within routine `rhs` is spent in useful computations. For routine `x_solve` it is 46%, for y_solve it is 31% and for `z_solve` it is 14%. In Figure 9 we show the view of useful computations for an MLP run on 10 processes with 10 threads each. This is not the combination that scales best, but it is best suited to point out the performance issues. Dark blue represents time spent in useful instructions and light blue indicates none useful time.

The initial part in Figure 9 represents parallel regions inside routine `rhs`. A detailed analysis of the time behavior shows that parallel computations tend to get blocked for periods of time that happen to be in the order of 10ms or multiples of that. A close look at the view of instructions executed by the threads in those intervals exposes the fact that a very low number of instructions has been executed by at least one thread. We interpret this as the thread having been preempted. Until it regains a CPU, other threads get blocked at the barriers between OpenMP work sharing constructs. These barriers are required by the OpenMP standard and not necessary for the correctness of the program's execution.

The denser regions of useful computations correspond to the single parallel region in `x_solve`. This is the second marked area in Figure 9. Its behavior in terms of duration is fairly imbalanced and can be attributed to effects like data placement and memory access time.

Routine `y_solve` has two major parallel regions; the second one containing pipelined thread execution within one MLP process. The thread pipeline within each process contributes to the overall poor percentage of useful computation in this routine. Routine y_solve corresponds to the third marked area in Figure 9.

The last marked area corresponds to the useful computations of routine z_solve. It contains a parallel region where every OpenMP thread uses MLP signal and wait routines to synchronize with other threads, possibly in other processes. This creates a pipeline from the first to the last process. A very detailed time analysis shows a second pipeline with opposite direction, resulting in a V-shaped pattern of useful computation. All of these dependencies between threads contribute to the very low percentage of useful computation in this routine.

Even though the MPI/OMP implementation employs the same algorithm it does not show the problem of CPU preemption in routine rhs that we identified in its MLP counterpart. The interaction of OpenMP and MPI causes less OS side effects than MLP. However, in this case it is the MPI library that causes major problems. The MPI calls within the parallel region of routine z_solve have an extremely low MPIS value. This low value can be explained by the process being preempted. We have also observed massive migration of threads during the whole run of the MPI/OMP code. This results in the poor scalability of the MPI/OMP code. For the MLP code we observed thread migration only during the initial phase of program execution.

We conclude that the fine grain parallelism introduced in the hybrid parallelization leads to interactions between OS activities, such as system interrupts, and the program synchronization operations. The impact of these interactions on performance increases rapidly with the number of processors. The blocking of processes maybe be caused by OS interrupts or context switches, but may also be done by the OpenMP or MPI library in order to support single and multi-user mode on the system. The behavior is very much dependent on the implementation of the run time libraries for parallel execution.

The OpenMP implementation has the best performance for a single CPU run and has the lowest memory requirements. The implementation does not use pipelined thread execution but rather places the directives on the first dimension in some of the loops. The outer loop of the Class A BT benchmark has 62 iterations. Clearly, single OpenMP parallelization will not scale beyond 62 threads. Nesting of OpenMP directives allows distributing the work in two dimensions thereby providing work for more threads. A view of the useful computations of the nested OpenMP code for 100 CPUs is shown in Figure 10. The view corresponds to best performing nesting level employing 25 groups of 4 threads. For this combination we calculated an average of 44% of time spent in useful computations per iteration. All other combinations yielded a lower percentage of useful time. The non-useful time is mostly spent in idle time, not so much in synchronization and fork/join operations. The NanosCompiler provides its own thread library, which handles thread synchronization and scheduling very efficiently. The thread idle time can probably be reduced by optimizing the distribution of the loop iterations onto the threads. The NanosCompiler provides clauses to allow detailed mappings of the iteration space onto threads, but in our study we have only used a simple block distribution.

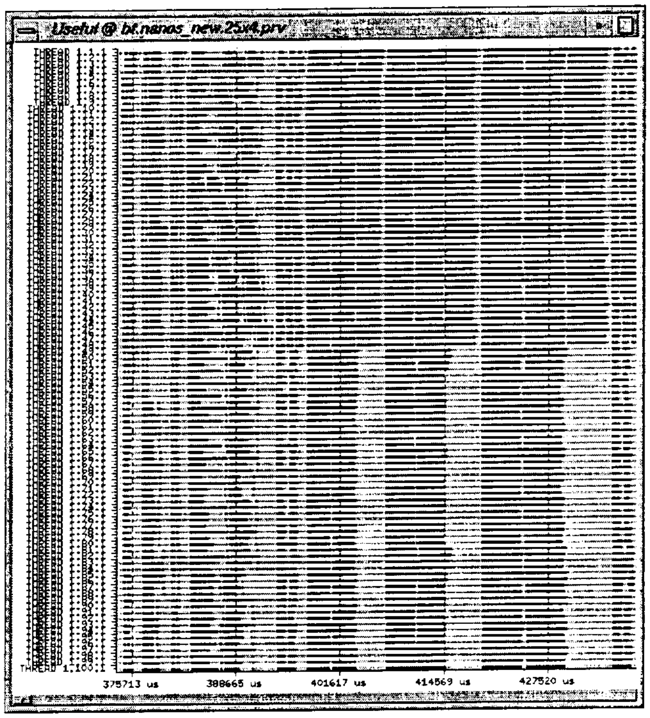


**Figure 10: Useful computations for BT OpenMP.**

## 5 Related work

There are many published reports on the comparison of different programming paradigms. We can only name a few of the. In [19] Taft discusses the performance of a large CFD application. He compares the scalability of message passing versus MLP. A comparison of message passing versus shared memory access is given in[15] and [17]. The study uses the SGI SHMEM library for RMA programming. A discussion of nested OpenMP parallelization is given in [7]. Our current work differs from these reports in that we are exploiting powerful performance analysis tools to obtain detailed information about program behavior.
There are a number of commercial and research performance analysis tools that have been developed over the years. An example for a commercial product is Vampir [21], which allows tracing and analysis of OpenMP, MPI, and hybrid MPI/OpenMP applications. TAU (Tuning and Analysis Utilities) [20] was developed at the University of Oregon. It is a freely available set of tools for analyzing the performance of the C,C++, Fortran and Java programs. To our knowledge neither of these tools supports the MLP programming paradigm.

## 6 Conclusions

We have used the Paraver/OMPItrace performance analysis system to conduct a detailed performance analysis of five different implementations of the BT NAS Parallel benchmark employing different programming. We have demonstrated how the performance analysis tool was used to calculate performance statistics expose patterns in the traces, which help to identify performance problems.

A first conclusion of our study is that implementation issues in the run time library and their interaction with the OS are very relevant to the total application performance. The hybrid implementations of our benchmark where particularly affected by the interaction of the run time libraries for process and thread parallelization.

We also observed several issues that are related to the programming model itself rather than the underlying run time library. The strength of the purely process based model lies in the fact that the user has complete control over process synchronization and data distribution. When employing OpenMP this it not the case anymore.

The scalability of our hybrid codes suffered from this fact that process level communication was delayed until the barrier synchronization points. Nested OpenMP parallelism requires the nesting of parallel regions, which introduces many barrier synchronization points into the outer region. This raises the question whether the OpenMP standard should be extended to allow nested parallel loops without extra parallel regions. Such an extension would also make nested parallelism more efficient.

The second issue concerns workload distribution. The well-balanced distribution of work, which yields the good scalability for the SMA implementation, is not supported by OpenMP directives. It is possible to mimic the process level parallelization, by assigning specific loop bounds to each thread manually. However, this would defeat the advantage of OpenMP which is the ease of the programming paradigm. We are currently working with the NanosCompiler group to identify suitable OpenMP extensions.

**Acknowledgments**

This work was supported by NASA contracts NAS 2-14303 and 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] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995. http://www.nas.nasa.gov/Software/NPB.

[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] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. http://captools.gre.ac.uk/

[5] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance", *NAS Technical Report NAS-99-011,* 1999.

[6] H. Jin, M. Frumkin and J. Yan. "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," in *Proceedings of Third International Symposium on High Performance Computing* (ISHPC2000), Tokyo, Japan, October 16-18, 2000.

[7] H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, X. Martorell, "Automatic Multilevel Parallelization Using OpenMP", 3rd European Workshop on OpenMP, Barcelona, Spain, Sep. 2001.

[8] R.D. Loft, S. J. Thomas, J.M. Dennis, "Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models", *Proceeding of Supercomputing,* Denver, Nov. 2001.

[9] D. J. Marvripilis, "Parallel Performance Investigations of an Unstructured Mesh Navier-Stokes Solver", *Technical Report NASA/CR-2000-210088, ICASE No.2000-13,* ICASE, Hampton, Virginia, 2000.

[10] MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich.

[11] MPI–2: Extensions to the MPI Interface, http://www-unix.mcs.anl.gov/mpi/mpich.

[12] OMPItrace User's Guide, https://www.cepba.es.paraver/manuals.htm

[13] OpenMP Fortran Application Program Interface, http://www.openmp.org/.

[14] Paraver, http://www.cepba.upc.es/tools.paraver/.

[15] H. Shan, J. Pal Singh, "A comparison of MPI, SHMEM, and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessor", *International Journal of Parallel Programming,* Vol. 29, No. 3, 2001.

[16] A. Serra, N. Navarro, and T. Corts, "DITools: Application-level support for Dynamic Extension and Flexible Composition", Proceedings of the USENIX Annual Technical Conference, June 2000.

[17] H. Shan, J. Pal Singh, "Comparison of Three Programming Models for Adaptive Applications on the Origin 2000", *Journal of Parallel and Distributed Computing 62, 241-266, 2002.*

[18] J. Taft, "Achieving 60 GFLOP/s on the production CFD code OVERFLOW-MLP," *Parallel Computing*, 27 (2001) 521.

[19] J. Taft, "Performance of the OVERFLOW-MLP Code on the NASA Ames 512 CPU Origin System", *NASA HPCCP/CAS Workshop*, NASA Ames Research Center, February 2000.

[20] TAU: Tuning and Analysis Utilities, http://www.cs.uoregon.edu/research/paracomp/tau/.

[21] VAMPIR User's Guide, Pallas GmbH, http://www.pallas.de