Analysis and Optimization of a Hybrid Linear Equation Solver using Task-Based Parallel Programming Models

Claudia Rosas, Vladimir Subotic, José Carlos Sancho, Jesús Labarta

BSC, Barcelona Supercomputing Center, Jordi Girona, 29, Barcelona, 08034, Spain

Abstract

This paper describes a methodology and tools to analyze and optimize the performance of task-based parallel applications. For illustrative purposes, a cutting-edge implementation of the Jacobi method aimed to address software challenges at exascale computers is evaluated. Specifically, the analysis was carried out on synchronous and asynchronous task-based implementations of the Jacobi method. The methodology consists of three basic steps: (i) performance analysis; (ii) prediction; and (iii) implementation. First, by instrumenting and tracing an application a general overview of its behavior can be obtained. The Paraver visualization tool enables the identification of performance bottlenecks or scalability problems. Secondly, with the help of prediction tools, such as Tareador and Dimemas, the inherent parallelism of the application is evaluated. Finally, the code is refactored to solve potential inefficiencies that prevent it to achieve higher performance. This final step is accomplished by using the OmpSs task-based parallel programming language. Results reported from using the methodology highlighted performance issues regarding to memory access, synchronization among the threads, and processors with long waiting periods. Additionally, the OmpSs implementation enabled the parallel execution of core functions of the application inside each thread, therefore obtaining a greater utilization of the computational resources.

1. Introduction

In the last years, we have been witnesses of a new paradigm shift in computer systems driven by the emergence of multicore processors. Currently, higher processor performance can only be achieved by an increase of core count. As an example, current Knights Corner Intel processors [1] provide more than 50 physical cores reaching one teraflop/s of peak performance. This hardware paradigm shift is disrupting any traditional software development tools, because they were conceived to deal with single-core processor engines rather than multiple cores.

There is an urgent need to adapt software design to new multicore processors. The adaptation process requires an entirely new thinking and innovation to use this kind of processing power. Applications must efficiently exploit the availability of parallelism that large amount of cores are providing. Therefore, new parallel programming languages, libraries, tools and compilers become necessary to design and implement applications able to take advantage of the potentiality of parallelism.

Along with these challenges, dataflow programming models have been developed to harness the potential of multicore architectures. Examples of these are the task-based parallel programming models, such as MPI [2]/OpenMP [3] and MPI/OmpSs [4]. OmpSs is an effort to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity. Compared to fork-join parallelism exposed by OpenMP, the parallelism of OmpSs can be much more irregular and distant.
Task-based parallel programming languages combine the programming model of MPI, dedicated to the distributed address space on different nodes in the system, and the programming model of OpenMP and OmpSs, focused on shared address space within the node. However, optimizing hybrid codes, which combine more than one programming model, implies a difficult and complex process because of limited development in analysis and improvement tools for hybrid-programming models.

Currently, the programmer is fully involved in the process of exploiting the potential parallelism of the application. In general, code is manually partitioned into smaller and different tasks (this process is referred to here as taskification). However, in many cases, a programmer does not always achieve the optimal taskification. It can result in even more difficult large codes, where the amount of tasks might become overwhelming; making extremely difficult to manually achieve an appropriate decomposition into tasks. In addition, it can be almost impossible to anticipate the performance of an application using a specific taskification in a specific multicore machine. The complexity and uncertainty behind porting existent applications to a more efficient hybrid parallel programming model, prevents programmers of doing so, due to uncertainty about whether porting the application is worth the programming effort.

Additionally, new development tools would be crucial to educate tomorrow's software developers to design, develop, and debug the next generation of applications for modern, multi-core platforms. Learning of multi-core and multi-threading concepts, and to harness the available parallelism are key to develop new forthcoming applications.

Tools and methodologies are necessary to efficiently optimize applications that follow a task-based parallel programming model. Tools will be used to both investigate techniques to hide stalls in a task-based execution, or to explore methods to identify parallelization bottlenecks and pinpoint critical code sections. Moreover, strategies to deploy and increment task-based parallelism in MPI applications might be needed.

In this paper, it is shown the usability of performance analysis and visualization tools developed at Barcelona Supercomputing Center [5]. Tools enable programmers to optimize existent task-based applications. To this aim an easy methodology has been designed. The methodology guides the user along three steps to identify potential performance bottlenecks and scalability issues when an application is using a large number of threads. The methodology comprises a first performance analysis, followed by a prediction stage where potential performance is evaluated before finally doing new implementations.

To illustrate the use of the methodology, every step was applied to a hybrid implementation of a linear equation solver based on the Jacobi’s method [6]. This algorithm was chosen because it is widely used in the core of scientific applications. Moreover, this code represents new implementations of algorithms particularly developed for the forthcoming exascale machines. In large-scale machines, it is desired to avoid tightly coupled synchronization between participating cores.

The rest of this paper is organized as follows. Section 2 provides an overview of the target application where the methodology is applied, and a brief description of the tools used along the analysis and improvement process. Then, Section 3 describes the method of performance analysis and presents initial insights of the behavior of the application. In Section 4, initial performance issues are overcome by using OmpSs task based parallel programming model. Finally, a summary and conclusions are presented in Section 5.

2. Understanding and improving a hybrid parallel application

In this paper, an easy methodology to analyze the performance and improve current development of parallel hybrid applications is proposed. This methodology, summarized in Figure 1, considers an already implemented application, which goes through the proposed loop to reach a more efficient implementation.

The methodology consists of three major steps to evaluate, propose modifications, and improve a hybrid code. First, a previous performance analysis of the application must be done in order to: (i) identify possible performance issues, (ii) obtain initial insights about the implementation, and (iii) visualize its overall
performance. In this step, most of the performance factors (i.e. total execution time, memory accesses, application structure, among other) are visualized inside a graphical environment, enabling a detailed view of the execution of the application. Visualization tools ease the analysis process, and therefore, reduce the time and cost to identify performance bottlenecks (in comparison to read the code, when available). This process can be easily performed by using tracing and visualization tools, such as Extrae [7] and Paraver [8], both developed at BSC.

Second, a prediction analysis is performed after main bottlenecks are identified and reported. Main proposals of performance improvements are evaluated before modifying the source code. The prediction is based on coupling the understanding of scalability problems from the estimation of the expected performance of the application for a particular taskification. To this aim Tareador [9] and Dimemas [10] tools are used. Along this step, main insights of potential modifications to the original source code are obtained. Additionally, it facilitates the understanding of the benefits of considering a different paradigm to develop hybrid applications, such as using OmpSs as task-based parallel programming model. The relevance of this step lies in avoiding source code modification before knowing the maximum benefits of it. Thus, the user/programmer obtains an overall view of the expected performance before actually changing a line of the source code.

Finally, once the main issues are reported and potential optimizations are identified, e.g. defining an optimal taskification, the programmer may proceed to the evaluation of the solutions on a real production machine. Real modifications are introduced into the source code and are executed. In the implementation step, the methodology proposes the use of runtimes, compilers and task-based programming languages such as Nanox [11], Mercurium [12], and OmpSs [4]. Nevertheless, the migration process should not entail a significant effort from the programmers’ point of view. It could be easily understood and carried out by programmers who have some knowledge about distributed or shared-memory programming models, such as MPI [2] or OpenMP [3].

The proposed methodology may iterate once and again along the three steps to refine any flaws that have been left behind.

2.1 Jacobi’s method implementations

Since the deployment of efficient hybrid parallel applications is a shared goal within research groups inside the PRACE project, the proposed methodology took as reference a hybrid code written by our partners at EPCC.

The Jacobi’s method implemented by EPCC [6] provides the solution of sets of linear equations $Ax = b$, where $A$ is a large, sparse $n \times n$ matrix, and $x$ and $b$ are vectors. The algorithm is based on an iterative method, where a number of computational iterations are required to converge to the solution. Traditional iterative techniques for solving such a system in parallel are typically synchronous. In synchronous implementations, all processors must exchange updated vector information at the end of the iteration, and the algorithm may require scalar reductions. Because of the frequent synchronizations, these implementations might not scale as expected in large-scale parallel computers, such as the coming exascale machines.

Additionally, there is a more complex variant of the algorithm, based on asynchronous communication and computation to reduce undesirable synchronization. In this case, processes go through the iterative algorithm without synchronization after finishing the iteration. Therefore, processes may iterate at different rates. As communication
may occur at different rate inside the iterations it may be overlapped with computation. In addition, the receiving process will continue iterating with the data it has, incorporating new data as it is received.

The code follows a weak-scaling method where the problem size increases with the number of processes. Each process defines a cube of $n \times n \times n$ elements, which is replicated in every process. Figure 2 illustrates the partitioning of a problem size equal to $n = 4$ and 64 processes. Processes are arranged in a cube shape, e.g. in a cube of $4 \times 4 \times 4$ processes. Therefore, the final matrix to be solved will be $64 \times 64 = 4096$ data elements.

Processes communicate only with adjacent nodes to exchange their halo data. Non-blocking MPI_Issend communication operations are used for this purpose. A global reduction is performed every several iterations to check for convergence. In the asynchronous strategy, global reductions are non-blocking and being performed during computation. The current implementation uses a binary tree to perform the non-blocking reduction using a series of point-to-point messages using non-blocking MPI_Isend operations.

Both synchronous and asynchronous MPI Jacobi implementations were provided by EPCC. In addition to these, EPCC also provided specific implementations for MPI/OpenMP. These latter implementations have been recently successfully tested in the large parallel computer HECToR [13] to solve 4.1 billion unknowns on 32,768 cores. Depending on whether OpenMP threads are computing synchronously or asynchronously, there will be a different implementation of the code. In summary, there are three different MPI/OpenMP implementations, synchronous-synchronous (syncsync), asynchronous-synchronous (asyncsync), and asynchronous-asynchronous (asyncasync) referring to whether MPI and OpenMP are synchronous or asynchronous. In this paper, we are dealing with only syncsync and asyncsync implementations. The selected problem size used in our experimental evaluations is $50 \times 50 \times 50$ variables per MPI process.

2.2 Tools to analyze task-based parallel programming languages

2.2.1 Extrae

Mpitrace [14] is the tracing library we used for instrumenting parallel execution. Mpitrace intercepts calls to certain functions and records to the trace the events that mark these occurrences. By default, it intercepts MPI functions, timestamps the occurrence of the calls and reports that information to the trace. In addition, mpitrace can record information collected from various hardware counters. The output of mpitrace is a trace that can be fed to Paraver for visualization or to Dimemas for further simulation of parallel execution.

Mpitrace is implemented as a lightweight, easy to use, dynamic library. When the application preloads mpitrace, the tracing library intercepts functions and events of interest and generates the output trace. This way, mpitrace can instrument the application without the need of restructuring the code. The intercepting mechanism is very lightweight, providing the overhead of around 200ns per intercepted call, and around 6µs for reading hardware counters. Furthermore, mpitrace uses conventional optimization techniques, such as:
sampling (switching on/off instrumentation), filtering (configuring which subset of the events should be tracked), and buffering (collecting events locally and flushing them to the disc in bulk).

2.2.2 Paraver

Paraver [8] is a parallel program visualization and analysis tool. Paraver provides a qualitative perception of the application's time-behavior by visual inspection. Moreover, it provides a quantitative analysis of the run. Paraver reveals bottlenecks in parallel execution and discovers areas of suboptimal performance. This functionality is essential in the process of optimizing parallel applications. Paraver is not tied to any programming model. It provides a flexible three-level model of parallelism on which the parallel execution should be mapped. Currently, Paraver supports MPI, OpenMP, OmpSs, pthreads and CUDA, as well as the hybrid programming models such as MPI/OpenMP and MPI/OmpSs.

Paraver is a flexible data browser that provides a huge analysis power. Paraver's flexibility comes from the fact that the trace format is independent of the programming model. Thus, the visualization requires no changes in order to support some new programming model. In order to visualize time-behavior of a programming model, the programming model needs to express its performance data in the Paraver independent trace format. On the other hand, Paraver's expressing power comes from the possibility to easily program different metrics. The tool provides filter and arithmetic operations on the essential metrics in order to generate new derived and more complicated ones. This way, from one trace, the tool can provide different views with different metrics. To capture the expert's knowledge, any view can be saved as a Paraver configuration file. Re-computing the view on new data is as simple as loading a file.

2.2.3 Dimemas

Dimemas [10] is an open-source trace file based simulator for analysis of message-passing applications on a configurable parallel platform. The Dimemas simulator reconstructs the time behavior of a parallel application on a machine modeled by a set of performance parameters. Dimemas also allows to model aspects of the MPI library and the operating system. As input, Dimemas takes the trace produced by mpitrace and the configuration file of the modeled parallel machines. As output, Dimemas generates trace files that can be processed by two performance analysis tools: Paraver and Vampir [15]. These visualization tools enable the user to qualitatively inspect the simulated parallel time-behavior.

The initial Dimemas architecture model considered networks of Shared Memory Processors (SMP). Dimemas configuration file defines the modeled target machine, specifying the number of SMP nodes, the number of cores per node, the relative CPU speed, the memory bandwidth, the memory latency, etc. The simulation also allows different task to node mappings. The communication model consists of a linear model and nonlinear effects, such as network congestion. The configuration file parameterizes the network specifying the bandwidth, the latency, and the number of global buses (denoting how many messages can concurrently travel throughout the network). In addition, each processor is characterized by the number of input/output ports that determine its injection rate to the network.

Dimemas simulates the trace by replaying computation records and re-evaluating communication records. Dimemas trace consists basically on two types of records: computation records specifying duration of the computation burst; and communication records specifying transfer parameters such as the sender, the receiver, the message size, the tag, whether the operation was blocking or non-blocking, etc. During the simulation, the computation records are simply replayed — each computation burst lasts as it is specified in the trace. Conversely, each communication record is reevaluated — each transfer request is evaluated considering the specified configuration of the target machine and the newly calculated transfer time is incorporated in the simulated run.

2.2.4 Tareador

Dataflow can extract very distant and irregular parallelism that a programmer himself can hardly identify.
Tareador [9] allows the programmer to start from a sequential application, and using a set of simple constructs it proposes some decomposition of the sequential code into tasks. Then, Tareador dynamically instruments the annotated code and at run-time detects actual data-dependencies among the proposed tasks. Furthermore, Tareador automatically estimates the potential parallelism of the proposed decomposition, providing to the programmer the data-dependency graph of the tasks as well as the potential parallel time-behavior. Similarly, the programmer can start from an MPI application, propose some decomposition of MPI processes into OmpSs tasks and observe the parallelism of the potential MPI/OmpSs execution.

The idea of the framework is to: (i) run an MPI code with annotations that mark task decomposition; (ii) dynamically detect memory usage of each annotated task; (iii) identify dependencies among all task instances; and (iv) simulate the parallel execution of the annotated tasks. First, Tareador instruments all annotated tasks in the order of their instantiation. That way, the instrumentation can keep the shadow data of all memory objects and thus identify data dependencies among tasks. Considering the detected dependencies, Tareador creates the dependency graph of all tasks, and finally, simulates the OmpSs execution. Moreover, Tareador can visualize the simulated time-behavior and offer deeper insight into the OmpSs execution. Note that instrumenting an MPI code with annotated tasks; Tareador can reconstruct the potential parallel MPI/OmpSs execution.

The Tareador framework described in Figure 3, takes the input code and passes it through the tool chain that consists of the Mercurium [12] based code translator, the Valgrind [16] based tracer, the Dimemas replay simulator and Paraver visualization tool.

Input code is a complete MPI/OmpSs code or an MPI code with only light annotations specifying the proposed taskification. A Mercurium based tool translates the input code in the pure MPI code with inserted functions annotating entry and exit from each task. The obtained code is compiled and executed in pure MPI fashion. Each MPI process runs on top of one instance of the Valgrind virtual machine that implements a designed tracer. The tracer makes the authentic trace of the (actually executed) MPI execution, while at the same time it reconstructs what would be the artificial trace of the (potential) MPI/OmpSs execution. Processing the obtained traces, the Dimemas simulator reconstructs parallel time-behavior on a reconfigurable platform. Finally, Paraver can visualize the simulated time-behaviors and allow studying the differences between the (instrumented) MPI and the (corresponding simulated) MPI/OmpSs execution.

Based on Tareador, we design the top-to-bottom trial-and-error approach that can be used to port sequential applications to OmpSs. The approach requires no knowledge or understanding of the target code. The programmer starts by proposing a very coarse-grain task decomposition of the studied sequential code. Then,
Tareador estimates the potential parallelism of the proposed decomposition and plots the potential parallel time-behavior. Based on this output, the programmer decides how to refine the decomposition to achieve higher parallelism. The programmer repeats these steps of proposing decomposition until finding a decomposition that exposes satisfactory parallelism.

Given the final decomposition, the programmer can use Tareador to get hints to complete the process of exposing OmpSs parallelism. Tareador detects for each parameter of the task whether it is used as input, output or inout. Moreover, Tareador warns the programmer about the objects that are accessed in tasks but not passed through the parameters list. Finally, Tareador can be used as a debugging tool – the programmer can run it on the already existing MPI/OmpSs code to automatically detect all the miss-uses of the memory. By doing all these checks with Tareador, the programmer can assure that an MPI/OmpSs code, is correct from the point of view of OmpSs parallelization.

2.3 The OmpSs programming language

OmpSs is based on the OpenMP programming model with some modifications to its execution and memory model in order to support asynchronous parallelism and heterogeneous devices such as GPUs. It is open source and available for download at: https://pm.bsc.es/ompss.

In particular, the OmpSs execution model is a thread-pool model instead of the traditional OpenMP fork-join model. The master thread starts the execution and other threads cooperate executing the work it creates (whether it is from work-sharing or task constructs). Therefore, there is no need for a parallel region. Nesting of constructs allows other threads to become work generators as well. On the other hand, the OmpSs memory model assumes a non-homogeneous disjoint memory address space. As such, shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data, while for shared data it must specify how this is going to be used. This assumption is true even for shared memory machines as the implementation may reallocate shared data taking into account memory effects (e.g., NUMA).

In order to support OmpSs, programs must be modified to reflect this runtime model. This process is referred to here as “taskification”. Furthermore, OmpSs allows annotating task constructs with three additional clauses: Input, Output, and Inout. Input specifies that the construct depends on some input data, and therefore, is not eligible for execution until any previous construct with an output clause over the same data is completed. Output specifies that the construct will generate some output data, and therefore, is not eligible for execution until any previous construct with an input or output clause over the same data is completed. Inout specifies a combination of input and output over the same data.

Finally, to support heterogeneity and data motion between address spaces a new construct is introduced: the target construct. It allows one to specify on which devices the construct should be targeting (e.g. Cell, GPU, SMP, etc.) and also specifies that a set of shared data may need to be transferred to the device before the associated code will be executed. In addition, there is a specific construct called implements, which specifies that the code is an alternate implementation of the target devices of the function name in this clause. This alternate clause can be used instead of the original one if the implementation considers it appropriate.

3. Performance Analysis

3.1 Test bed description

The test bed where the Jacobi implementations were executed is called MinoTauro and it is being hosted by BSC. It consists of 126 compute nodes and 2 login nodes. Every node has two processors Intel Xeon E5649 6-Core at 2.53 GHz running Linux operating system with 24 GB of RAM memory, 12MB of cache memory and 250 GB local disk storage. The machine is interconnected using an InfiniBand QDR that provides 40Gbits bandwidth. Network topology is a non-blocking 2-level fat-tree.
3.2 Performance Analysis

This section describes the analysis performed on both synchronous-synchronous (syncsync) and asynchronous-synchronous (asyncsync) versions of the hybrid Jacobi code. The applications were implemented using a combination of MPI and OpenMP parallel programming models. The main difference between the applications is related to the collective MPI communication process among the computational nodes. In syncsync, a synchronization step is called between iterations along the execution. This step precedes data exchanging between threads and MPI. While in asyncsync, an asynchronous tree-based reduction method has been implemented to overlap both computation and communication from different iterations along the execution.

In this work, performance analysis is based on instrumenting applications and generating traces, which can be visualized and analyzed by using tools such as Paraver. However, due to the size of final traces the number of MPI processes (and therefore cores) has been kept low. Results shown in this section have been generated using 16 processing nodes (one MPI process per compute node). Inside each node, the number of cores (threads) has been increased from 1 to 12, obtaining executions with 16, 32, 64, 128, and 192 cores, respectively.

3.2.1 Parallel Performance

To obtain an overall view of the performance of the applications, syncsync and asyncsync implementations have been executed using 16, 32, 64, 128 and 192 cores. From these executions, the total execution time has been collected as an initial performance metric to observe the influence of synchronization steps within the execution. As a result, a small reduction in total execution time is observed when computation and communication are overlapped by asynchronous communications between the MPI processes (as shown in Figure 4). Moreover, there is a continuous climb of total execution time when adding more computational resources. This raise is given by the increase of the total problem size in every dimension to maintain weak scaling.

To identify possible performance bottlenecks the total execution time of the applications will not be used as a main performance metric. Instead of total execution time, we choose as performance metrics: the ratio of useful computation, the effective number of Instructions per Cycle (IPC), and a ratio of general computation unbalance. Metrics that can be seen in the traces generated after instrumenting the applications’ source code.

Useful computation represents the duration of a computation burst between an exit from MPI and the next entry, i.e. summarizes the major computation phases. Values close to 100% indicate that for example, applications have been doing computations all the time, while lower values indicate that applications may have been losing performance because of a large number of communications. Useful IPC shows the ratio between the total clock cycles spent along the execution and the total number of instructions that have been executed. Finally, general unbalance reports the fraction of total execution time where there has been lack of parallelism; i.e. there are idle processes while the application is running.

Results from executions of syncsync and asyncsync applications are summarized in Figure 5. First, in Figure 5(a) the ratio of total execution time that applications are doing computation can be observed. In this case, when increasing the number of cores (and therefore the problem size) there is a reduction in the total fraction of time where processors are doing effective computation. In comparison with the first column (1 thread per MPI process), adding more cores (threads) degrades up to a
50% the total useful computation in syncsync and up to a 62% in asyncsync.

For syncsync, obtained results might be related to long waiting times in the synchronization steps between iterations (due to slowest processes). While in asyncsync, although the removal of synchronization steps has reported lower execution times; the overall degradation when adding more computational resources might be influenced by an unexpected increase in communications and/or memory accesses.

When analyzing the rate of useful instructions per clock (shown in Figure 5(b)) a strong degradation when increasing the number of cores in asyncsync executions can be seen. This suggests that multi-threading and perhaps an inherent synchronization inside parallel sections might be reducing the amount of effective computation inside the executions. A reduction in useful computation could mean that some processes might be idle along the execution.

Hypothesis about having idle processes can be confirmed by analyzing the global unbalance of the execution of the applications. Figure 5(c) reports a constant increasing in the fraction of total execution time where there has been lack of parallelism. In the hybrid implementations of Jacobi’s method that have been analyzed, this raise implies an inefficient distribution of computational load, causing some processes to do more computation than other.

3.2.2 Graphical identification of performance bottlenecks

The traces generated after instrumenting and executing the applications can be viewed in Paraver. By doing this, it is possible to analyze the influence of main implementation details, such as: collective communication patterns and memory accesses. To provide a simpler and clearer view, all traces presented in this section were generated using 16 MPI processes (physical computing nodes) and 1 computational thread each.

Cuts from the useful duration traces generated after executing syncsync and asyncsync applications, are shown in Figure 6(a) and Figure 6(b), respectively.

In Figure 6(a) the barriers between iterations can be clearly seen, as well as the cost of computation in the different sections of the application. Based on a gray scale representation, darker sections are executing a larger number of useful instructions. Additionally, white spaces represent portions of the execution where processes are communicating. In these areas, two types of communications are carried out: (i) point-to-point communications, where one MPI process exchanges data with its neighbors (white spaces between blocks of dark areas); and (ii) collective synchronous communications, where all MPI processes exchange local data to calculate global values. In the latter, faster processes will not be able to do more computation, and therefore will be idle until all MPI processes have reached that point. In contrast, in the trace of asyncsync shown in Figure 6(b), it can be noticed how computation and communication (from different iterations along the

![Figure 5. Initial performance analysis when increasing the number of cores for asynchronous and synchronous versions of hybrid Jacobi implemented using MPI and OpenMP: (a) Percentage of total execution doing effective computation; (b) Useful IPC achieved by core; and (c) Percentage of total execution time lost because of unbalance between processes.](image-url)
execution) can be overlapped. Orange portions in the trace represent semantic values that cannot be displayed within the given grayscale (they are greater than the maximum chosen for both traces).

When computation and communication are overlapped, point-to-point communications (represented as blue lines in Figure 7) instead of being carried out in clearly defined regions along the execution (as shown in Figure 7(a)), are held at any moment while the application is running (Figure 7(b)).

Finally, as messages are being continuously sent and received during applications execution, it can be expected that the number of memory accesses may be affected. In this sense, reported ratios of L2 data cache misses for

---

**Figure 6.** Useful duration cut from trace generated for syncsync using 16 computational nodes (MPI processes) and 1 thread by process: (a) syncsync; (b) asyncsync. Low computation areas are shown in light gray, while darker areas represent high computations. White spaces represent intervals where the application is communicating.

**Figure 7.** MPI calls and communications performed in an interval of 35,000 µs. Traces obtained from the execution of hybrid implementations of Jacobi method in 16 MPI processes using 1 thread each: (a) syncsync; (b) asyncsync.
every 1000 instructions are shown in Figure 8. Again, values are based on a gray scale where darker spots represent a higher number of data cache misses incurred by a specific computational thread during the execution. Figure 8(a) and Figure 8(b) are trace cuts of 35,000 µs taken from the execution of syncsync and asyncsync applications.

As expected, there is a significant increment in L2 data cache misses incurred when synchronization steps are removed. Initially, expensive memory reads were condensed in some portions of the total execution in syncsync implementation of hybrid Jacobi (Figure 8(a)). Nevertheless, with MPI processes communicating with their neighbors or to the rest of the MPI processes at any time, the amount of data being read from memory, and the continuous load of new data when needed by a different process increased the ratio of L2 data cache misses (Figure 8(b)).

As a consequence from the results shown in this section, main performance degradations observed in both hybrid implementations of Jacobi’s method are closely related to the communication stages within the execution. Initially, using synchronous collective communications generate long waiting times that delayed fastest processes.

However, the implementation of an asynchronous collective communication protocol using a tree-based reduction generates a large number of communications that has reduced parallelism between the processors. The performance degradation is caused by constant memory accesses, and unexpected idleness or synchronization inside the parallel sections.

3.3 Thread-Scaling Optimization

In this section the scalability problem when multiple threads are used in the MPI/OpenMP version of the Asynchronous Jacobi is identified. This problem is related to quantifying the amount of communicating data that is exchanged during the execution. Furthermore, a potential solution is suggested to overcome this problem, which could substantially increase the performance of this code when using a large number of threads.

3.3.1 Default Thread-scaling Strategy

In MPI/OpenMP Jacobi the problem size per node is specified by an input parameter n and the number of
threads used in the node. Each thread is assigned a regular cube of dimension, \(T_x \times T_y \times T_z\), where \(T_x = T_y = T_z = n\). Multiple threads in the node are growing the problem along the \(z\)-dimension as depicted in Figure 9. For illustration purposes, the corresponding cubes for each different thread in a node are shown in blue. Orange cubes are shown to illustrate the corresponding cubes for threads placed in some adjacent neighboring nodes. Therefore, the problem size per node can be defined as a volume of \(N_x \times N_y \times N_z\) where

\[
N_x = n \\
N_y = n \\
N_z = n \times T
\]

In this case, \(T\) is the number of OpenMP threads used within the node, resulting in a total volume of \(n^3 \times T\).

As known, Jacobi nodes communicate to exchange their \textit{Halo} with their adjacent nodes. In Figure 9, the nodes placed in the \(+Y\), \(-Z\), and \(+X\) positions are shown. The amount of data exchanged every iteration corresponds to the surface of the cubes. For example, the amount of data that has to be exchanged with Node \(-Z\) is \(X \times Y = n^2\), that corresponds to the surface of the area of one face of the only cube that is close to node \(-Z\). A similar amount of going home data will be also exchanged in the other direction \(+Z\).

On the other hand, for communication in the other dimensions, \(X\) and \(Y\), there is more than one cube to exchange halo data with. This is because the number of cubes along the dimension \(Z\) is proportional with the number of threads.

In summary, the resulting amount of data exchanged associated with the area of the faces of the cubes in each dimension is given by:

\[
\text{Dimension } X = T \times n^2 \\
\text{Dimension } Y = T \times n^2 \\
\text{Dimension } Z = n^2
\]

As it can be seen there is some imbalance in the amount of data exchanged in the different dimensions. The amount of data communicated along the dimension \(Z\) is fixed to the problem size \(n\) whereas in the other dimensions the amount of data grows with to the number of OpenMP threads. This imbalance could impact negatively the performance of the application, as some communications are taking longer than others because they have more data to transmit. This problem is exacerbated in parallel computers that have a 3D topology such as 3D torus, where there is an independent communication channel per each dimension. In that case, some channels might be overloaded while others might be transmitting very low data. Furthermore, the amount of data communicated grows proportional with the number of threads, which could be posed a scalability problem as well.

3.3.2 Using Performance Tools to Analyze Communications

The Paraver visualization tool is used to analyze per thread level the communications operations during execution. Figure 10 shows the communication operations when running Jacobi with a problem size equal to 10, and eight processes with two OpenMP
threads. Brown boxes correspond to the communications occurring during the Halo swap, and red and pink boxes correspond to communicating the residual value. In particular, MPI_Isend communication operations are used for this purpose. The amount of data exchanged during communication is equal to 1,600 bytes for each one. It remains similar for all threads when they communicate to neighbor nodes located in any dimension. It is the result of doubling the area of one side of a cube for this particular problem size. The area of one side of the cube is 10 * 10 data elements. Also, it is doubling because there are two cubes, one per each thread.

As it can be seen, the amount of data exchanged in each dimension, X, Y, and Z, is the same, corresponding to two cube side areas. However, as described earlier, the amount of data communicated along the dimension Z should be only one cube side area. This issue is causing Jacobi to communicate more data than expected. Moreover, this example illustrates the potential of the Paraver tool to identify issues on the communication operations in hybrid MPI and OpenMP codes.

Based on this analysis, we suggest code developers to analyze this functionality to reduce the amount of communicated data during the Halo swap. The potential improvements could be significant. It will be decreased from $3T * n^2$ to $(2T + 1) * n^2$.

### 3.3.3 Proposed strategy to improve thread-scaling

The proposed solution to balance communication in all dimensions is based on distributing the cubes of the threads evenly among the three dimensions. The idea is to build a bigger cube based on the thread cubes. This new scaling is illustrated in Figure 11. In this figure, the corresponding volume of the problem size by node is shown in blue. Similarly to Figure 9, each thread receives a regular cube of dimension $T_x * T_y * T_z$ where $T_x = T_y = T_z = n$. The amount of cubes placed in each dimension is equal to $\sqrt[3]{T}$, where $T$ is the number of threads. Note that when the number of threads is not a cubic number then the number of cubes would be evenly balanced among the different dimensions.

Therefore, the size per node can be defined as a volume of $N_x * N_y * N_z$ where,

$$N_x = n \times \sqrt[3]{T}$$
$$N_y = n \times \sqrt[3]{T}$$
$$N_z = n \times \sqrt[3]{T}$$

The resulting volume of the cube in the node will be $n^3 * T$ which is similar to the value obtained using the default scaling described previously.

The advantage of using this new scaling is twofold. First, the amount of data communicated is evenly distributed among the three dimensions. There is the same number of cubes in each dimension, i.e. $\sqrt[3]{T} \times \sqrt[3]{T}$. The amount of data communicated in each dimension is described as following,

$$Dimension \ X = \sqrt[3]{T} \times \sqrt[3]{T} \times n^2$$
As a result, Figure 12 shows the total communicated data per node (across all three dimensions) when increasing the number of threads for the default and proposed scaling. As it can be seen, in comparison to the default scaling, the proposed scaling reports a lower amount of communicated data per node. Furthermore, the reduction in communicated data is greater as the number of threads increases. In particular, for 256 threads the improvement between the default and the proposed scaling is 4.5X.

4. Optimizing the code with OmpSs

4.1 Task analysis

This section covers the analysis of the potential inherent parallelism in the current implementation of the Jacobi’s method. First, the potential optimal taskification to achieve a higher performance on multi-core processor architectures is shown. Then, by using simulation, the expected number of cores required in real executions to achieve a higher performance is estimated. For the analysis, the *syncsync* Jacobi implementation is used. To simplify the visualization of resulting graphs, a small problem size of 4 x 4 x 4 variables is chosen. The variable *Iter1* is the variable that controls the point where *halo swap* among neighbor nodes is performed. *Iter2* is the variable that represents where the calculation and global evaluation of the residue is done.

The Tareador tool is used to analyze the data dependencies in the source code. Tasks are manually specified using simple clauses in the code. It only requires setting up the start and end of the section evaluated by Tareador, as well as the sections of the code that will represent a task. Then, Tareador will analyze the data dependencies to see if the proposed task can be executed in parallel in an ideal multi-core machine with an infinite number of cores.

To illustrate this process, Figure 13 shows the pseudo-code of the Jacobi method on the left, and the different *taskifications* (decompositions into tasks) proposed on the right. There are ten different taskifications proposed, D1, D2, D3, D4, D5, D6, D7, D8, D9, and D10. D1 is the taskification where all the code is one task. This one follows the behavior of a traditional sequential program. From D2 up to D10 different parallelization strategies
of the code are explored, going from a coarse-grain to a finer-grain. The latter, \(D10\), taskifies down to the computation performed per data element in the grid. Notice that new tasks created from the previous taskification are shown in dark blue color boxes to help illustrate the changes made from one taskification to another.

The output of the Tareador tool corresponding to decomposition \(D6\), is depicted Figure 14. Lines connecting the different tasks show the data dependencies among them. Tasks aligned horizontally can be executed in parallel, and thus, they are a source of parallelism in the code.

In this particular taskification, it is worth noting that the communication for halo swap operations in both dimensions, \(X\) and \(Y\), are taskified, but not the \(Z\) dimension. This yields a total of \(4 \times 4 = 16\) halo swap tasks shown in Figure 14. However, the number of computation tasks created for this taskification is lower, only four. This is the reason why the computation is dependent of four halo swaps. To balance the execution it might be necessary to add more computation tasks, and then to observe if they can be run in parallel as well.

In decomposition \(D10\) (shown in Figure 15), the finest grain for taskification is used and it can be seen a higher level of parallelism. For clarity purposes, this figure shows only part of its task data dependency. For this decomposition, at every halo swap task there are new tasks doing computations independently from each other. It is important to note that there are no data dependencies among these independent tasks, so they can be freely executed in parallel. This is an interesting conclusion that can be easily obtained using the Tareador tool. Decomposition \(D10\) exhibits the highest parallelism, thus resulting in an optimal taskification.

As the granularity of tasks is finer the amount of tasks created increases. This is illustrated in Figure 16 where it is shown the total amount of tasks created for the different taskifications. As it can be noticed, the amount of tasks created grows exponentially as the granularity of tasks is setting up smaller. This occurs because in the code there are “for” loops to go through the grid, and then when the code inside a loop is taskified; it creates a task by iteration of the loop. The large amount of tasks created in \(D10\) could be an issue in a real multi-core

![Figure 13. Proposed decompositions into tasks to parallelize main functions of the Jacobi method.](image)
processor due to the overhead of creating tasks in current processors, and therefore it has to be tested in a real machine.

![Diagram](image.png)

Figure 14. Task dependency visualization for decomposition D6

Furthermore, the analysis to identify the number of cores needed to achieve a highest performance for the finer decomposition $D10$ was carried out. Reported results are shown in Figure 18. By looking at the curve, there is an inflection point close to 32 cores. This inflection indicated that only 32 cores are needed to achieve the highest performance improvement.

![Diagram](image.png)

Figure 15. Task dependency visualization for decomposition D10

For illustration purposes, a screenshot of a visualization obtained from Dimemas simulations is included. Specifically, Figure 19 presents a resulting simulation for the execution of $D10$ on 32 cores. In this figure it is clearly seen how almost all the cores are doing both computation and communication along all the execution (except by the core/thread 15 that seems to be executing only the main task). Finally, it is worth pointing out
that tasks allocation is controlled by the Nanox runtime, and therefore tasks are executed dynamically on whatever core is becoming available.

To sum up, we demonstrated that Jacobi’s method is highly parallelizable and a task-based parallel programming language could exploit this feature. An optimal taskification is suggested to be as fine as possible, always depending on the additional overhead of creating and destroying a task in a processor. The decomposition shown in this section now has to be implemented in the original source code and executed on a real machine. Additionally, from results obtained from simulation, we expect that a low number of cores (only 32 cores) will be necessary to achieve a potential performance improvement of 50X.

4.2 The OmpSs implementation

The OmpSs programming model extends OpenMP functionalities to support asynchrony, heterogeneity and data movement for tasks parallelism. OmpSs is based on augmenting an existing serial version of an application with compiler directives that are translated into calls to a runtime system that manages the parallelism extraction, and data coherence and movement. The main idea behind a task-based programming model is to have a single programming model, OmpSs, covering current and future homogeneous and heterogeneous architectures.
Having the main concepts of OmpSs in mind and after the performance analysis and simulations, a first approach to achieve the benefits of using tasks-based programming languages is shown next. Functions identified with Tareador, which can be decomposed in parallel tasks, are now annotated with main directives of the OmpSs programming model. These annotations identify the functions that the runtime/compiler can distribute in parallel among available cores, i.e. any call to the annotated function creates a new task that will execute the function body.

Main functions of the Jacobi initial implementation were annotated. Annotations include the corresponding clauses to indicate the dependence synchronization of each generated task. In these cases, work-sharing constructs that depend of some input data are identified with the clause input; those that generate output data are labeled with an output clause; and finally, those which combine some input and output functions receive an inout clause.

From the analysis done with Tareador for the implementation of the Jacobi’s method, it has been seen that updateValue() and CopyDataBack() functions, can benefit from a decomposition into tasks. Different portions of data can be read and written in parallel without affecting the final result. This can be done because local portions of data are then updated in the exchange process. Moreover, exchanging process (named HaloSwap) can be decomposed into 6 different functions, one for each neighbor in the grid, and therefore, can be parallelized as well.

An overview of the resulting source code, after inserting the corresponding annotations, is shown in Figure 20. Here, OmpSs directives are inserted by adding a pragma omp task before the name of the function. By doing this, each time these functions are called during the execution, a task is instantiated and the runtime will decide where to put it in available cores.

In addition, each annotation indicates which element can be shared between the tasks. In halo swap functions, such as get_put_right(), get_put_left(), etc. (there are 6: left, right, up, down, front and back), there is one element from where data is read by the task (grid_orig) and another in which data is written (grid_dest). According to the description of OmpSs clauses to indicate dependence synchronization, in the pragma directive, grid_orig is annotated as input, and grid_dest is annotated as output. The same reasoning is used to indicate the dependences for updateValue and copyDataBack functions.

Once all the functions to be parallelized are annotated, the source code inside the main function can be written as shown in Figure 21. Each function can be called inside of for loops, where tasks are generated and distributed. Nevertheless, once the taskified section is complete, and before doing computation that may involve all data, it is necessary to call a synchronization point or barrier using the #pragma omp taskwait. This guarantees that all the threads have reached that point before doing sensitive computation, e.g. calculating the average value of the elements in the grid.

To observe the advantage of using OmpSs in the implementation of the Jacobi method, the total execution time of a serial execution (1 thread) has been compared with the resulting execution time when increasing the number of cores from 1 to 12. The application has been compiled with the Mercurium version 1.3.5.8 and Nanox 0.7a. Additionally, the instrumentation flag has been enabled to call Extrae and generate the execution traces for Paraver. Executions have been performed using a fixed problem size (size of the grid inside each process) equal to 50 and a fixed number of iterations equal to 60. The results are reported in Figure 22.

There is a strong reduction in total execution time when increasing the number of threads used by OmpSs. In this case, although the creation of tasks in OmpSs could take some time, the distribution of these tasks has enabled an improvement in total execution time of up to an 80% when using 12 threads in comparison with the serial version. Nevertheless, it is worth mentioning that there is a limit in the performance improvement. In this example, this limit is reached when 8 and 10 threads are used, indicating that beyond that point, potential gains in total execution time might not justify increasing the number of active threads.

The traces of user defined functions reported from executions using 1, 2, 4, and 8 active threads are shown in Figure 23. By augmenting the number of threads a reduction in total execution time is achieved. This is due to
a better distribution of generated tasks among available cores. Computation areas, represented as violet and pink, correspond to the updateValue and copyDataBack functions, respectively. And communication stages, represented by the exchange halo section that has been divided into 6 functions (one for each frontier in the grid), are executed at the end of each block and are represented by the colorful lines. At the same time, for these executions, along the 60 iterations, there are 3 printing stages (one every 20 iterations) that correspond with the 3 white spaces between blocks doing computation and communication.

Finally, initial trace from executing the MPI/OmpSs implementation is shown in Figure 24. Here, the execution is based on synchronous communications between MPI processes, and asynchronous communication between OmpSs threads. As a first insight, it can be observed the similarity of the obtained trace with the result from Tareador’s simulations (please refer to Figure 19). Both traces represent an execution with 32 cores. In the case of the implementation based on MPI and OmpSs, 4 MPI processes were used and every process had 8 threads active. Halo swaps between tasks were performed every 10 iterations (represented as...
small colorful tasks at the beginning and end of the cut). In between, there are blocks of tasks generated for `updateValue` and `copyDataBack` functions (pink and yellow blocks respectively). By using OmpSs as task-based parallel programming, the use of available cores is increased, and therefore greater parallelism is reported. Nevertheless, additional extensive experimentation using MPI/OmpSs, and final comparisons with original MPI/OpenMP implementations, are still under development and will be reported soon.

Figure 23. Traces for user functions in the implementation of the Jacobi method using OmpSs directives, problem size equal to 50, and number of tasks inside each thread equal to 4x4x4. (a) 1 thread; (b) 2 threads; (c) 4 threads; and (d) 8 threads.
5. Conclusions

This paper shows the usability of performance analysis and development tools designed and implemented by the Barcelona Supercomputing Center. Tools help programmers and analyzers to understand and optimize existing task-based applications. To this aim, a methodology that takes the analyzer along the process of understanding and optimizing an unknown hybrid application was proposed. In this work, a linear equations solver based in the Jacobi’s method implemented using MPI and OpenMP programming models was used to illustrate the analysis and improvement process. First, through instrumentation and tracing using Extrae, and trace visualization using Paraver, it was possible to identify potential performance bottlenecks and scalability issues. Second, in the prediction step, tools such as Tareador and Dimemas were useful to identify and evaluate parallel decompositions for the application. Additionally, potential solutions for the synchronization issues that were reducing the efficiency of the application were evaluated. Finally, based on results obtained in previous steps, an initial implementation using OmpSs as task-based programming model was designed for Jacobi’s method. The results show that resources utilization can be easily improved when the application is decomposed into small tasks that can be executed in parallel. This concept, along with the functionalities obtained from the N anos runtime and the Mercurium compiler enabled a simpler and efficient way to implement hybrid applications.

6. Acknowledgements

This work was financially supported by from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°RI-283493. We would also like to thank Mark Bull, Nick Brown and Iain Bethune from EPCC for providing the Asynchronous Jacobi code and helping us to understand the configuration options and the implementation characteristics of the code.

7. References


