Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications

Vladimir Subotic, Roger Ferrer, Jose Carlos Sancho,
Jesús Labarta, and Mateo Valero

Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
{vladimir.subotic,roger.ferrer,jsancho,jesus.labarta,mateo.valero}@bsc.es

Abstract. Task-based parallel programming languages require the programmer to partition the traditional sequential code into smaller tasks in order to take advantage of the existing dataflow parallelism inherent in the applications. However, obtaining the partitioning that achieves optimal parallelism is not trivial because it depends on many parameters such as the underlying data dependencies and global problem partitioning. In order to help the process of finding a partitioning that achieves high parallelism, this paper introduces a framework that a programmer can use to: 1) estimate how much his application could benefit from dataflow parallelism; and 2) find the best strategy to expose dataflow parallelism in his application. Our framework automatically detects data dependencies among tasks in order to estimate the potential parallelism in the application. Furthermore, based on the framework, we develop an interactive approach to find the optimal partitioning of code. To illustrate this approach, we present a case study of porting High Performance Linpack from MPI to MPI/SMPSs. The presented approach requires only superficial knowledge of the studied code and iteratively leads to the optimal partitioning strategy. Finally, the environment provides visualization of the simulated MPI/SMPSs execution, thus allowing the developer to qualitatively inspect potential parallelization bottlenecks.

1 Introduction

New proposals for large-scale programming models are persistently spawned, but most of these initiatives fail because they attract little interest of the community. It takes a giant leap of faith for a programmer to take his already working parallel application and to port it to a novel programming model. This is especially problematic because the programmer cannot anticipate how would his application perform if it was ported to the new programming model, so he may doubt whether the porting is worth the effort. Moreover, the programmer usually lacks developing tools that would make the process of porting easier.

MPI/SMPSs is a new hybrid dataflow programming model that showed to be efficient for numerous applications. In a manner similar to MPI/OpenMP, MPI/SMPSs parallelizes computation of the distributed-memory nodes using

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 39–51, 2011.

MPI [13], while it parallelizes computation of the shared-memory cores using SMPSs [10], a task-based dataflow programming model. This integration of message-passing paradigm and dataflow execution potentially extracts distant parallelism (parallelism of code sections that are mutually "far" from each other). Finally, MPI/SMPSs outperforms MPI in numerous codes [8], among which is the High Performance Linpack (HPL), the application that is used to rank the parallel machines on the top 500 supercomputers lists [1].

To continue its progress, MPI/SMPSs must get wider community involved by encouraging MPI programmers to port their applications to MPI/SMPSs. This encouragement is strictly related to assuring the programmer that he can benefit from this porting and that the porting would be easy. Therefore, our goal in this study is to develop a framework that provides support to:

- help an MPI programmer estimate how much parallelism MPI/SMPSs can achieve in his MPI application, so he can decide whether the porting is worth the effort.
- help an MPI programmer find the optimal strategy to port his MPI application to MPI/SMPSs.

2 SMPSs Programming Model

SMPSs [10] is a new shared-memory task-based parallel programming model that uses dataflow to exploit parallelism. SMPSs slightly extends C, C++ and Fortran, offering semantics to declare some part of a code as a task, and to specify memory regions on which that task operates. In porting a sequential code to SMPSs, the programmer has to specify the following: *taskification* – to mark with pragma statements the functions that should be executed as tasks; and *directionality of parameters* – to mark inside pragmas how are the passed arguments used within these function. The specified directionality can be: *input*, *output* and *inout*. Figure 1 illustrates the annotations needed to port a sequential C code to SMPSs.

Given the annotations, the runtime is free to schedule all tasks out-of-order, as long as the data dependencies are satisfied. The main thread starts and when it reaches a taskified function, it instantiates it as a task and proceeds with the execution. Based on the parameters' directionality, the runtime places the task

```
#pragma css task   input(A[SizeA]) output (B[SizeB])
void compute(float *A, float *B) {
    …
}
```

```
int main () {
    …
    compute(a,b);
    …
}
```

Note: The code in black presents the unchanged code of the legacy C application. Conversely, the code in dark gray presents the annotations needed to mark the *taskification choice*, while the code in light gray presents the annotations needed to declare the *directionality of parameters*.

Fig. 1. Annotations needed to port a code from sequential C to SMPSs

instance in the dependency graph of all tasks. Then, considering the dependency graph, the runtime is free to dynamically schedule the execution of tasks to achieve high parallelism. To further increase dataflow parallelism, the runtime automatically renames data objects to avoid all false dependencies (dependencies caused by buffer reuse).

Integrated with MPI, SMPSs allows to taskify functions with MPI transfers and thus potentially extract very distant parallelism. The idea is to encapsulate functions with MPI transfers inside tasks, and thus relate the messaging events to dataflow dependencies. For example, a task with *MPI_Send* of some buffer locally reads (*input* directionality) that buffer from the memory and passes it to the network, while a task with *MPI_Recv* of some buffer gets that buffer from the network and locally stores (*output* directionality) it to the memory. Taskification of transfers overcomes strong synchronization points of pure MPI execution and potentially exploits distant parallelisms, providing much better messaging behavior than fork-join based MPI/OpenMP. Marjanovic *at. el.* [8] showed that apart from better peak GFlops/s performance, compared to MPI, MPI/SMPSs delivers better tolerance to bandwidth reduction and external perturbations (such as OS noise).

3 Motivation

Finding the best taskification strategy is far from trivial. Figure 2 shows a simple sequential application composed of four computational parts (A, B, C and D), the data dependencies among those parts, and some of the possible taskification strategies. Although the application is very simple, it allows many possible taskifications that expose different amount of parallelism. $T0$ puts all code in one task and, in fact, presents non-SMPSs code. $T1$ and $T2$ both break the application into two tasks but fail to expose any parallelism. On the other hand, $T3$ and $T4$ both break the application into 3 tasks, but while $T3$ achieves no parallelism, $T4$ exposes parallelism between C and D. Finally, $T5$ breaks the application into 4 tasks but achieves the same amount of parallelism as $T4$. Considering that increasing the number of tasks increases the runtime overhead of instantiating and scheduling tasks, one can conclude that the optimal taskification is $T4$, because it gives the highest speedup with the lowest cost of the increased number of tasks. On the other hand, for a complex MPI application, the number of possible taskifications could be huge, so finding the optimal taskification can

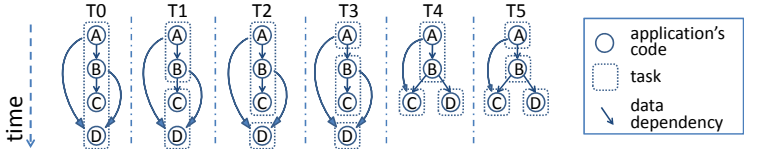


Fig. 2. Execution of different possible taskifications for a code composed of four parts

be both hard and time consuming. As a result, most likely, a programmer ends up with a sub-optimal taskification of his code.

We believe that it would be very useful to have an environment that quickly anticipates the potential parallelism of a particular taskification. We design such environment and we show how it should be used to find the optimal taskification. In this paper, as a case study we present a black-box approach to port the High Performance Linpack (HPL) from MPI to MPI/SMPSs. First, the environment instruments the studied application and generates quantitative profile of the execution. Then, considering the obtained profile the interactive trial-and-error process can start following this method: 1) the programmer proposes a coarse-grained taskification for the code; 2) given the taskification, the environment estimates potential parallelism and offers the visualization of the resulting MPI/SMPSs execution. 3) based on the output, the programmer proposes a finer-grained taskification and returns to step 2. This interactive algorithm converges into the optimal taskification.

4 Framework

The idea of the framework is to: 1) run an MPI/SMPSs code by executing tasks in the order of their instantiation; 2) dynamically detect memory usage of all tasks; 3) identify dependencies among all task instances; and 4) simulate the execution of the tasks in parallel. First, the framework forces sequential execution of all tasks, in other words it executes tasks in the order of their instantiation. That way, the instrumentation can keep the shadow data of all memory references and thus identify data dependencies among tasks. Considering the detected dependencies, the framework creates the dependency graph of all task, and finally, simulates the MPI/SMPSs execution. Moreover, the framework can visualize the simulated time-behavior and offer deeper insight into the MPI/SMPSs execution.

The framework (Figure 3) takes the **input code** and passes it through the tool chain that consists of Mercurium based **code translator**, Valgrind based

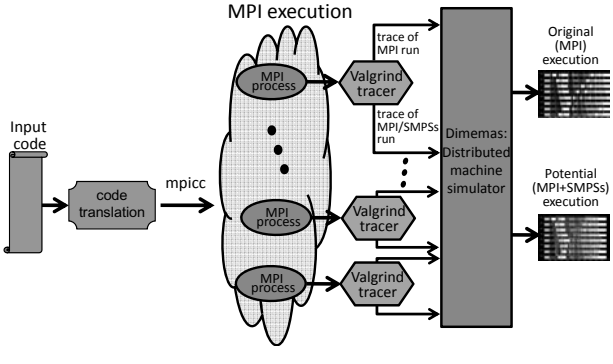


Fig. 3. The framework integrates Mercurium code translator, Valgrind tracer, Dimemas simulator and Paraver visualization tool

tracer, Dimemas **replay simulator** and Paraver **visualization tool**. Input code is a complete MPI/SMPSs 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 entries and exits from tasks. Then the obtained code is compiled and executed in pure MPI fashion. Each MPI process runs on top of one instance of Valgrind virtual machine that implements a designed tracer. The tracer makes the trace of the (actually executed) MPI execution, while at the same time, it reconstructs what would be the traces of the (potential) MPI/SMPSs execution. Dimemas simulator merges the obtained traces and reconstructs time-behavior of these traces on a parallel platform. Finally, Paraver can visualize the simulated time-behaviors and allow to profoundly study the differences between the (instrumented) MPI and the (corresponding simulated) MPI/SMPSs execution. In our prior work [14], we used a similar idea to estimate the potential benefits of overlapping communication and computation in pure MPI applications.

4.1 Input Code

The input code can be MPI/SMPSs code or an MPI code with light annotations. The input code has to specify which functions (parts of code) should be executed as tasks, but not the directionality of the function parameters. Thus, the input code can be an MPI code, only with annotations specifying which functions should be executed as tasks. Figure 4 on the left shows an example of an MPI code with annotated *taskification* choice.

4.2 Code Translator

Our Mercurium based tool translates the input code into the code with forced serialization of tasks. The obtained code is a pure MPI code with empty functions (*hooks*) annotating when the execution enters and exits from a task (Figure 4). The translated code is then compiled with *mpicc*, and the binary of the MPI execution is passed for further instrumentation. It is important to note that the

Input code	Translated code

`#pragma css task`
`void compute(float *A, float *B) {`
`…`
`}`
	`void compute(float *A, float *B) {`
	`…`
	`}`

`int main () {`
`  …`
`  compute(a,b);`
`  …`
`}`
	`int main () {`
	`  …`
	`  start_task_valgrind("compute");`
	`  compute(a,b);`
	`  end_task_valgrind("compute");`
	`  …`
	`}`

Note: The input code does not have to be a complete MPI/SMPSs code, because the instrumented code only needs to mark all entries/exits from each task. Thus, as shown, the input code can be an MPI application only with a specified proposed taskification.

Fig. 4. Translation of the input code required by the framework

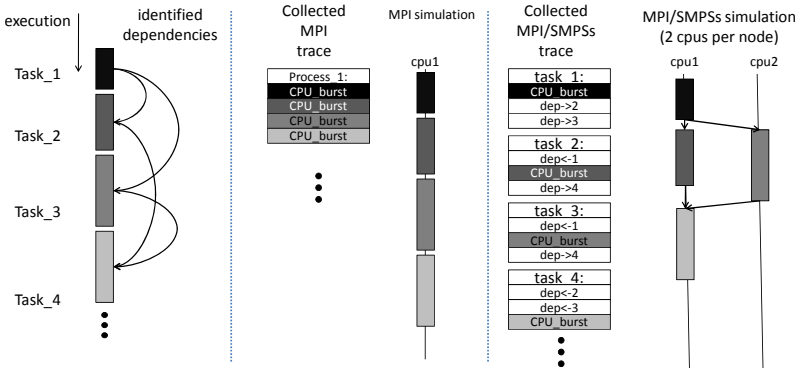
hooks can be inserted directly in the input code, allowing to declare as task any part of the application's code. This way, the framework overcomes the limitation of SMPSs runtime that only complete functions may be treated as tasks, and further eases the process of proposing taskifications.

4.3 Tracer

Valgrind [9] is a virtual machine that uses just-in-time (JIT) compilation techniques. The original code of an application never runs directly on the host processor. Instead, the code is first translated into a temporary, simpler, processor-neutral form called Intermediate Representation (IR). Then, the developer is free to do any translation of the IR, before Valgrind translates the IR back into machine code and lets the host processor run it.

Leveraging Valgrind functionalities, the tracer instruments the execution and makes two Dimemas traces: one describing the instrumented MPI execution; and the other describing the potential MPI/SMPSs execution. The tracer uses the following Valgrind functionalities: 1) intercepting the inserted *hooks* in order to track which task is currently being executed; 2) intercepting all memory allocations in order to maintain the pool of data objects in the memory; 3) intercepting memory accesses in order to identify data dependencies among tasks; and 4) intercepting all MPI calls in order to track MPI activity of the execution. Using the obtained information, the tracer generates the trace of the original (actually executed) MPI execution, while at the same time, it reconstructs what would be the trace of the potential (not executed) MPI/SMPSs execution.

The tool instruments accesses to all memory objects and derives data dependencies among tasks. By intercepting all dynamic allocations and releases of the memory (*allocs* and *frees*), the tool maintains the pool of all dynamic memory objects. Similarly, by intercepting all static allocations and releases of the



Note: The tracer describes the MPI traces by emitting two types of records: 1) computation record defining the length of computation burst; and 2) communication record specifying the parameters of MPI transfers. Conversely, it describes the MPI/SMPSs trace by breaking the original computation bursts into tasks and synchronizing the created tasks according to the identified data dependencies.

Fig. 5. Collecting trace of the original MPI and the potential MPI/SMPSs execution

memory (*mmaps* and *munmaps*), and reading the debugging information of the executable, the tool maintains the pool of all the static memory objects. The tracer tracks all memory objects, intercepting and recoding accesses to them at the granularity of one byte. Based on these records, and knowing in which task the execution is at every moment, the tracer detects all read-after-write dependencies and interpret them as dependencies among tasks.

The tool creates the trace of the executed MPI run, and at the same time, considering identified task dependencies, it creates what would be the trace of the potential MPI/SMPSs run (Figure 5). When generating the original trace, the tool describes the actually executed run by putting in the trace two types of records: 1) computation record stating the length of computation burst in terms of the number of instructions 2) communication record specifying the parameters of the executed MPI transfer. Additionally, when reconstructing the trace of the potential MPI/SMPSs run, the tracer breaks the original computation bursts into tasks, and then synchronizes the created tasks according to the identified data dependencies.

4.4 Replay Simulator

Dimemas is an open-source tracefile-based simulator for analysis of message-passing applications on a configurable parallel platform. The communication model, validated in [4], consists of a linear model and nonlinear effects, such as network congestion. The interconnect is parametrized by bandwidth, latency, and the number of global buses (denoting how many messages can concurrently travel throughout the network). Also, each processor is characterized by the number of input/output ports that determine its injection rate to the network. Finally, the simulated output of Dimemas can be visualized in Paraver.

We extended Dimemas to support synchronization of tasks in a way that allows Paraver to visualize all data dependencies. We implemented a task synchronization using an intra-node instantaneous MPI transfer that specifies the source and the destination tasks. This way, Paraver can visualize the simulated time-behavior showing both MPI communications among processes and data dependencies among tasks. Using this feature, the developer can visually detect each execution bottleneck and further inspect its causes.

5 Experiments

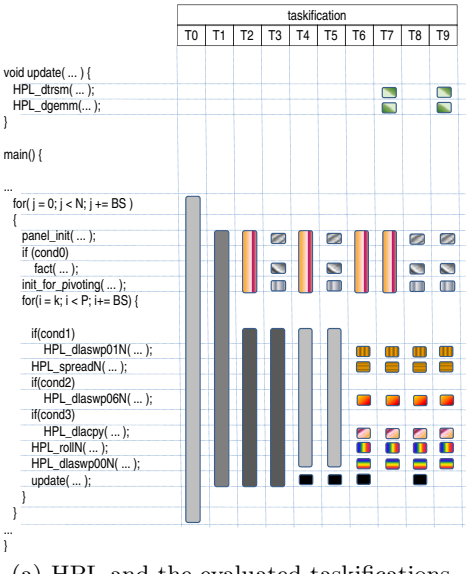
Our experiments explore MPI/SMPSs execution of HP Linpack on a cluster of many-core nodes. We used HPL with the problem size of 8192 and with 2x2 (PxQ) data decomposition. Also, we test various granularities of execution by running HPL with block sizes (BS) of 32, 64, 128, and 256. Our target machine consists of four many-core nodes, with one MPI process running on each node. We are primarily interested in the MPI/SMPSs potential parallelism inherent in the code, so we make most of the measurements for unlimited resources on the target machine – infinite number of cores per node and ideal interconnect

between the nodes. These results represent the upper bound of achievable parallelism. Finally, we show how this potential parallelism inherent in the application results in speedup when the application executes on a realistic target machine.

The major part of our experiment consist of exploring the potential MPI/SMPSs taskifications of HPL. In a case study with HPL, we present a top-to-bottom approach that uses a trial-and-error method, requires no knowledge of the studied code, and finally leads to exposing dataflow parallelism in the code. The approach uses the following method: 1) we propose a coarse-grained taskification for the code; 2) given the taskification, the environment estimates potential speedup and offers visualization of the resulting MPI/SMPSs execution. 3) based on the output, we choose a finer-grained taskification and return to step 2. We start from the most coarse-grain taskification ($T0$) that puts whole MPI process into one task and actually presents the traditional MPI execution (Figure 6(a)). Then using $T0$ as the baseline, we determine the *potential parallelism of* $Ti(1 \leq i \leq 9)$ *normalized to* $T0$ as the speedup of Ti over $T0$ when both these taskifications execute on a machine with unlimited number of cores per node and unlimited network performance (Figure 7(b)).

5.1 Results

First, the framework instruments the application to obtain the profile that guides the taskification process. Table 6(b) shows the accumulated time spent in each



(a) HPL and the evaluated taskifications.

		granularity			
task name			BS-32	BS- 64	BS-128	BS-256
	outer	panel_init	0.0003	0.0002	0.0001	0.0000
		fact	0.7525	1.2071	1.8795	3.2077
		init_for_pivoting	0.0246	0.0487	0.0925	0.1795
	inner	HPL_dlaswp01N	0.2583	0.2917	0.2906	0.2815
		HPL_spreadN	0.1599	0.0800	0.0378	0.0181
		HPL_dlaswp06N	0.1222	0.1359	0.1367	0.1274
		HPL_rollN	0.3267	0.1619	0.0762	0.0363
		HPL_dlacpy	0.0857	0.0932	0.0929	0.0912
		HPL_dlaswp00N	0.3706	0.4485	0.4736	0.4837
	update	HPL_dtrsm	0.8269	1.6674	2.8347	5.0772
		HPL_dgemm	97.0683	95.8614	94.0813	90.4935

(b) Distribution of total execution time spent in tasks (%).

		granularity			
task name			BS-32	BS- 64	BS-128	BS-256
	outer	panel_init	0.0003	0.0003	0.0003	0.0003
		fact	1.3468	3.7670	11.3572	37.8463
		init_for_pivoting	0.0221	0.0761	0.2797	1.0594
	inner	HPL_dlaswp01N	0.0073	0.0289	0.1130	0.4392
		HPL_spreadN	0.0022	0.0040	0.0073	0.0141
		HPL_dlaswp06N	0.0034	0.0135	0.0532	0.1987
		HPL_rollN	0.0046	0.0080	0.0148	0.0283
		HPL_dlacpy	0.0024	0.0092	0.0361	0.1423
		HPL_dlaswp00N	0.0052	0.0222	0.0921	0.3773
	update	HPL_dtrsm	0.0117	0.0826	0.5514	3.9605
		HPL_dgemm	1.3677	4.7492	18.3016	70.5900

(c) Average function duration (ms).

Note: In Tables 6(b) and 6(c), apart from statistic for each function of the code, we present the statistics for two logical sections: **outer** – consisting of *panel_init*, *fact* and *init_for_pivoting*; and **inner** consisting of *HPL_dlaswp01N*, *HPL_spreadN*, *HPL_dlaswp06N*, *HPL_rollN*, *HPL_dlacpy* and *HPL_dlaswp00N*.

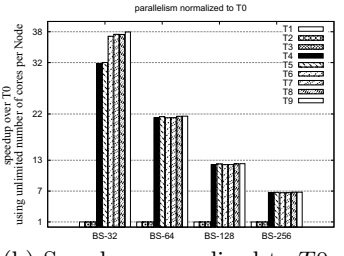
Fig. 6. Taskifications evaluated for HPL and duration and time spent in each function

function of the application. This information identifies instances of which functions need to execute concurrently in order to achieve significant parallelism. In this example, those are instances of functions *update*, because the application spends in that function from 95.57% (for $BS = 256$) to 97.83% (for $BS = 32$). On the other hand, Figure 6(c) shows the average duration of each function. This information identifies which function is a good candidate to be broken down into smaller tasks. In this example, function *panel_init* is very short so breaking it into smaller tasks makes little sense. Also, it is important to note that decreasing BS reduces execution time of most of the functions, so this could also be a way to make finer-grained execution.

Considering the data showed on previous tables, we start the process of exposing parallelism by: 1) proposing a taskification ($T1$ - $T9$ in Figure 6(a)); 2) testing how many tasks we created (Figure 7(a)); and 3) testing the potential speedup of the taskification (Figure 7(b)). $T0$ is the baseline taskification that makes only one task per MPI process. $T1$ puts each iteration of the outer loop in one task, but this strategy gives no additional parallelism compared to $T0$. Furthermore, $T2$ breaks down the code into section *outer* and separate iterations of the inner loop, still giving no improvement in speedup. $T3$ additionally breaks

		granularity			
		BS-32	BS-64	BS-128	BS-256
taskification	T1	1024	516	260	132
	T2	66.314	16.778	4.298	1.130
	T3	69.898	18.570	5.194	1.578
	T4	131.600	33.040	8.336	2.128
	T5	135.184	34.832	9.232	2.576
	T6	327.458	81.826	20.450	5.122
	T7	425.382	106.216	26.502	6.614
	T8	331.042	83.618	21.346	5.570
	T9	428.966	108.006	27.398	7.062

(a) Total number of tasks created.



(b) Speedup normalized to $T0$.

Note: In Figure 7(b), all taskifications ($T0$-$T9$) execute in MPI/SMPSs fashion on an ideal target machine. Then, the speedup of taskification Ti over taskification $T0$ represents the parallelism of taskification Ti normalized to taskification $T0$.

Fig. 7. Number of task instances and the potential parallelism of each taskification

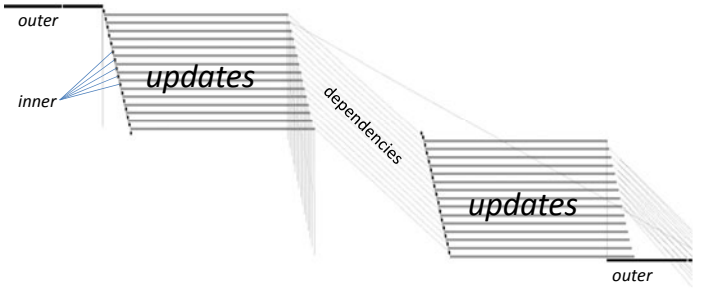
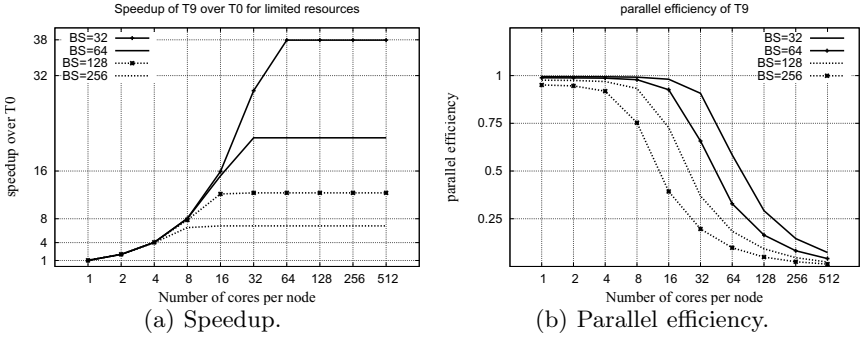


Fig. 8. Paraver visualization of the first 63 tasks and the dependencies among them (taskification T4, BS=256)

down section *outer*, but with no increases in speedup. This can be explained by Paraver visualization (results not presented in the paper) that shows that in $T2$ and $T3$, each iteration of the inner loop depends on the previous iteration, and thus impedes parallelism. Finally, $T4$ compared to $T2$ separates section *inner* from function *update* and releases the significant amount of parallelism. Namely, it achieves the speedup of 6.76, 12.28, 21.48 and 32.02 for block sizes of 256, 128, 64 and 32, respectively (Figure 7(b)). Also, $T4$ significantly increases the number of tasks in the application to 2.128, 8.336, 33.040 and 131.600 for block sizes of 256, 128, 64 and 32, respectively (Figure 7(a)). Now, Paraver visualization reveals that in $T4$: 1) each section *inner* depends on the section *inner* in the previous iteration of the inner loop; and 2) each *update* depends on the section *inner* in the same iteration of the inner loop. Thus, because section *inner* is much shorter than *update*, all dependent sections *inner* can execute quickly, and then independent instances of *update* can execute concurrently (Figure 8).

Further breaking down of *outer*, *inner* and *update* contributes little to the potential speedup (Figure 7(b)). Breaking of *outer*, for block sizes of 256, 128 and 64, causes slightly higher parallelism of $T5$, $T8$ and $T9$, compared to $T4$, $T6$ and $T7$. On the other hand, breaking of *inner*, for block size of 32, causes significantly higher parallelism of $T6$, $T7$, $T8$ and $T9$, compared to $T4$, $T5$. This effect happens because for very high concurrency of *update* (speedup is higher than 30), the critical path of the execution moves and starts passing through section *inner*. In these circumstances breaking of *inner* significantly increases parallelism by allowing concurrency of functions *HPL_dlaswp00N*, *HPL_dlaswp01N* and *HPL_dlaswp06N*. Finally, breaking of *update*, for block size 32, causes slightly higher parallelism of $T9$ compared to $T8$.

Figure 9 shows the speedup and parallel efficiency of $T9$ for different number of cores per node. The results show that high parallelism in the application is useful not to achieve high speedup on a small parallel machine, but rather to deploy efficiently a large parallel machine. Figure 9(a) shows that for a machine



(a) Speedup. (b) Parallel efficiency.

Note: Parallel efficiency denotes the ratio between the application's speedup achieved on some parallel machine and the number of cores of that parallel machine. Infact, the metric presents the overall average core utilization in the whole machine.

Fig. 9. Speedup and parallel efficiency for T9 for various number of cores

with 4 cores per node, $T9$ with all block sizes achieve a speedup of around 4, with difference between the highest and the lowest of less than 2%. However, for a machine with 32 cores per node, $T9$ with block sizes of 256, 128, 64 and 32, achieves the speedup of 6.80, 12.34, 21.57 and 29.47, respectively. Furthermore, Figure 9(b) shows parallel efficiency (core utilization) – the ratio between the application's speedup achieved on some parallel machine and the number of cores in that machine. Adopting that an application efficiently utilizes a machine if the parallel efficiency is higher than 75%, the results show that $T9$ with block sizes of 256, 128, 64 and 32, can efficiently utilize the machine of 8, 15, 26 and 47 cores per node, respectively. Therefore, to efficiently employ many-core machine with hundreds of cores per node, HPL has to expose even more parallelism, for instance, by making finer-grain taskification with further reduction of block size.

6 Related Work

Back in 1991 the community started claiming that instruction-level parallelism is dead [15], and consequently in the following 20 years appeared many programming models that exploit task-level parallelism. OpenMP [11] is the most popular programming model for shared memory that was founded with the idea of parallelizing loops, but from version 3.0 provides support for task parallelism. Cilk [2] implements a model of spawning various tasks and specifying a synchronization point where these tasks are waited for. MPI tasklets [5] parallelize SMP tasks by incorporating dynamic scheduling strategy into current MPI implementations. There are also proposals that originated from the industry, such as: TBB [12] from Intel and TPL [6] from Microsoft. Still, all these proposals suffer from the limitations of fork-join based programming models. On the other hand, SMPSs [10] is a programming model in which the programmer specifies dependencies among tasks, rather than specifying synchronization points. Then, based on the specified dependencies, the runtime schedules tasks in dataflow manner, potentially extracting very distant parallelism. Furthermore, SMPSs can be integrated with MPI, allowing better messaging behavior. Marjanovic *at. el.* [8] demonstrate that compared to MPI, MPI/SMPSs provides superior performance as well as higher tolerance to network reduction and external noise.

However, there is little development support for these programming models. Alchemist tool [16] identifies parts of code that are suitable for thread-level speculation. Embla [7] estimates the potential speed-up of fork-join based parallelization. Starsscheck [3] checks correctness of pragma annotations for STARSs family of programming models. Our work adds up to these efforts by designing a framework that estimates the potential parallelism of MPI/SMPSs. Furthermore, our work goes beyond the state-of-the-art tools because: 1) it deals with complex execution model that integrates MPI with task-based dataflow execution; 2) it allows to study MPI/SMPSs execution before the original MPI application is ported to MPI/SMPSs; 3) it provides an estimation of the parallelism on the configurable target platform; and 4) it provides visualization of the simulated execution.

7 Conclusion

Tasks-based parallel programming languages are promising in exploiting additional parallelism inherent in MPI parallel programs. However, the complexity of this type of execution impedes an MPI programmer from anticipating how much dataflow parallelism he can obtain in his application. Moreover, it is nontrivial to determine which parts of code should be encapsulated into tasks in order to expose the parallelism and still avoid creating unnecessary tasks that increase runtime overhead. To address this issue, we have developed a framework that automatically estimates the potential dataflow parallelization in applications. We show how, using the framework, one can find optimal taskification choice for any application through a trial-and-error iterative approach that requires no knowledge of the studied code. We prove the effectiveness of this approach on a case study in which we explore the taskification of High Performance Linpack (HPL). The results show that HPL expresses substantial amount of potential dataflow parallelism that allows the application to efficiently utilize cluster of nodes with up to 47 cores per node. Moreover, we show that the global partitioning significantly impacts parallel efficiency, and thus, in order to efficiently utilize higher number of cores, finer-granularity of execution should be used.

Acknowledgements. We thankfully acknowledge the support of the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the TEXT project (IST-2007-261580), and the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), and the Generalitat de Catalunya (2009-SGR-980).

References

1. Top500 List: List of top 500 supercomputers, `http://www.top500.org/`
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. J. Parallel Distrib. Comput. 37, 55–69 (1996)
3. Carpenter, P.M., Ramirez, A., Ayguade, E.: Starsscheck: A tool to find errors in task-based parallel programs. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 2–13. Springer, Heidelberg (2010)
4. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communication model for mpi collective operations. In: PVM/MPI, pp. 39–46 (2000)
5. Kale, V., Gropp, W.: Load Balancing for Regular Meshes on SMPs with MPI. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 229–238. Springer, Heidelberg (2010)
6. Leijen, D., Hall, J.: Parallel performance: Optimize managed code for multi-core machines. MSDN Magazine (2007)
7. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 26–37. Springer, Heidelberg (2010)

8. Marjanovic, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: ICS, pp. 5–16 (2010)
9. Nethercote, N., Seward, J.: Valgrind, `http://valgrind.org/`
10. Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: CLUSTER, pp. 142–151 (2008)
11. Proposed Industry Standard. Openmp: A proposed industry standard api for shared memory programming
12. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., Sebastopol (2007)
13. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. The MIT Press, Cambridge (1998)
14. Subotic, V., Sancho, J.C., Labarta, J., Valero, M.: A Simulation Framework to Automatically Analyze the Communication-Computation Overlap in Scientific Applications. In: CLUSTER 2010 (2010)
15. Wall, D.W.: Limits of Instruction-Level Parallelism. In: ASPLOS (1991)
16. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: CGO 2009 (2009)