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

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

1 Introduction

New proposals for large-scale programming models are persistently spaw
most of these initiatives fail because they attract little interest of the co
It takes a giant leap of faith for a programmer to take his already wor
allel application and to port it to a novel programming model. This is
problematic because the programmer cannot anticipate how would his
tion perform if it was ported to the new programming model, so he m
whether the porting is worth the effort. Moreover, the programmer usu
developing tools that would make the process of porting easier.

MPI/SMPSs is a new hybrid dataflow programming model that s
be efficient for numerous applications. In a manner similar to MPI/
MPI/SMPSs parallelizes computation of the distributed-memory no

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

parallelism (parallelism of code sections that are mutually "far" from ea
Finally, MPI/SMPSs outperforms MPI in numerous codes [8], among
the High Performance Linpack (HPL), the application that is used to
parallel machines on the top 500 supercomputers lists [1].

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

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

2 SMPSs Programming Model

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

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

```
#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. Co
code in dark gray presents the annotations needed to mark the *taskification choice*, wh
in light gray presents the annotations needed to declare the *directionality of parameter*

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

automatically renames data objects to avoid all false dependencies (dep
caused by buffer reuse).

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

3 Motivation

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

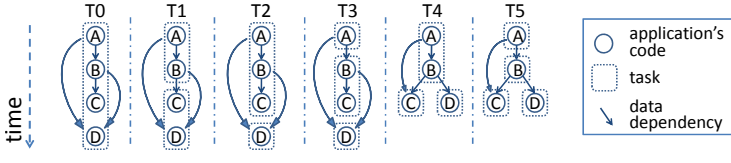


Fig. 2. Execution of different possible taskifications for a code composed of

anticipates the potential parallelism of a particular taskification. We de
environment and we show how it should be used to find the optimal
tion. In this paper, as a case study we present a black-box approach to
High Performance Linpack (HPL) from MPI to MPI/SMPSs. First, the
ment instruments the studied application and generates quantitative
the execution. Then, considering the obtained profile the interactive
error process can start following this method: 1) the programmer pr
coarse-grained taskification for the code; 2) given the taskification, the
ment estimates potential parallelism and offers the visualization of the
MPI/SMPSs execution. 3) based on the output, the programmer pr
finer-grained taskification and returns to step 2. This interactive algori
verges into the optimal taskification.

4 Framework

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

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

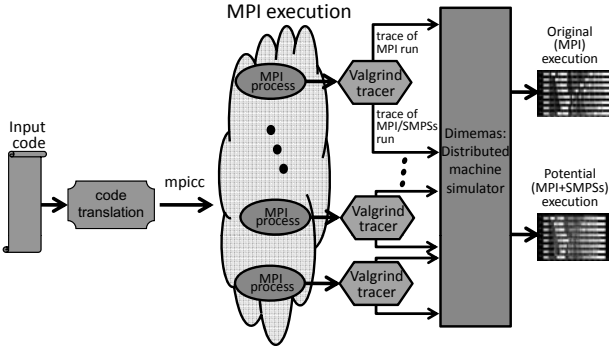


Fig. 3. The framework integrates Mercurium code translator, Valgrin
Dimemas simulator and Paraver visualization tool

the input code in the pure MPI code with inserted functions annotatin
and exits from tasks. Then the obtained code is compiled and execute
MPI fashion. Each MPI process runs on top of one instance of Valgrin
machine that implements a designed tracer. The tracer makes the tra
(actually executed) MPI execution, while at the same time, it reconstru
would be the traces of the (potential) MPI/SMPSs execution. Dimema
tor merges the obtained traces and reconstructs time-behavior of these
a parallel platform. Finally, Paraver can visualize the simulated time-
and allow to profoundly study the differences between the (instrumen
and the (corresponding simulated) MPI/SMPSs execution. In our p
[14], we used a similar idea to estimate the potential benefits of ov
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 ann
The input code has to specify which functions (parts of code) should be
as tasks, but not the directionality of the function parameters. Thus,
code can be an MPI code, only with annotations specifying which
should be executed as tasks. Figure 4 on the left shows an example o
code with annotated *taskification* choice.

4.2 Code Translator

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

Input code

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

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

Translated code

```
void compute(float *A, float *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 in
code only needs to mark all entries/exits from each task. Thus, as shown, the input cod
MPI application only with a specified proposed taskification.

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

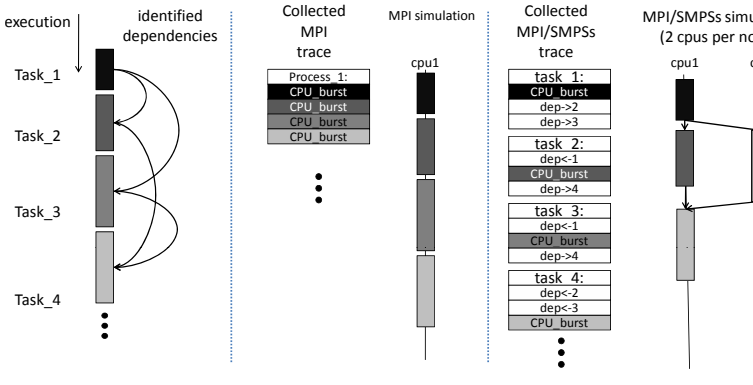
further eases the process of proposing taskifications.

4.3 Tracer

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

Leveraging Valgrind functionalities, the tracer instruments the exec
makes two Dimemas traces: one describing the instrumented MPI e
and the other describing the potential MPI/SMPSs execution. The tr
the following Valgrind functionalities: 1) intercepting the inserted *ho*
der to track which task is currently being executed; 2) intercepting al
allocations in order to maintain the pool of data objects in the memo
tercepting memory accesses in order to identify data dependencies amo
and 4) intercepting all MPI calls in order to track MPI activity of the e
Using the obtained information, the tracer generates the trace of th
(actually executed) MPI execution, while at the same time, it reconstru
would be the trace of the potential (not executed) MPI/SMPSs execu

The tool instruments accesses to all memory objects and derives da
dencies among tasks. By intercepting all dynamic allocations and relea
memory (*allocs* and *frees*), the tool maintains the pool of all dynan
ory objects. Similarly, by intercepting all static allocations and releas



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

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

at the granularity of one byte. Based on these records, and knowing task the execution is at every moment, the tracer detects all read-a dependencies and interpret them as dependencies among tasks.

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

4.4 Replay Simulator

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

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

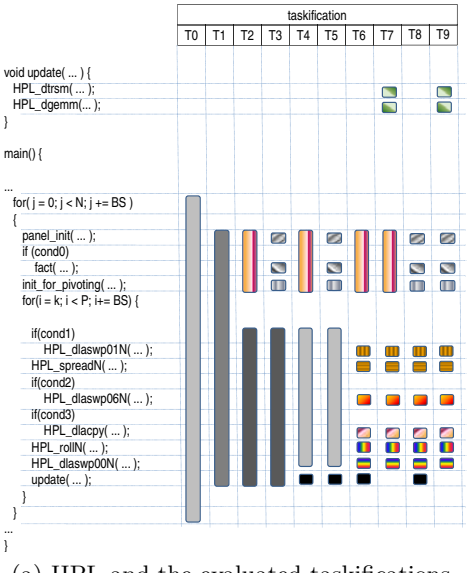
5 Experiments

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

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

5.1 Results

First, the framework instruments the application to obtain the profile th
the taskification process. Table 6(b) shows the accumulated time spen



(a) HPL and the evaluated taskifications.

			granul	
			BS-32	BS- 64
task name	outer	panel_init	0.0003	0.0002
		fact	0.7525	1.2071
		init_for_pivoting	0.0246	0.0487
	inner	HPL_dlaswp01N	0.2583	0.2917
		HPL_spreadN	0.1599	0.0800
		HPL_dlaswp06N	0.1222	0.1359
		HPL_rollN	0.3267	0.1619
		HPL_dlacpy	0.0857	0.0932
		HPL_dlaswp00N	0.3706	0.4485
	update	HPL_dtrsm	0.8269	1.6674
		HPL_dgemm	97.0683	95.8614

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

			granul	
			BS-32	BS- 64
task name	outer	panel_init	0.0003	0.0003
		fact	1.3468	3.7670
		init_for_pivoting	0.0221	0.0761
	inner	HPL_dlaswp01N	0.0073	0.0289
		HPL_spreadN	0.0022	0.0040
		HPL_dlaswp06N	0.0034	0.0135
		HPL_rollN	0.0046	0.0080
		HPL_dlacpy	0.0024	0.0092
		HPL_dlaswp00N	0.0052	0.0222
	update	HPL_dtrsm	0.0117	0.0826
		HPL_dgemm	1.3677	4.7492

(c) Average function durat

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

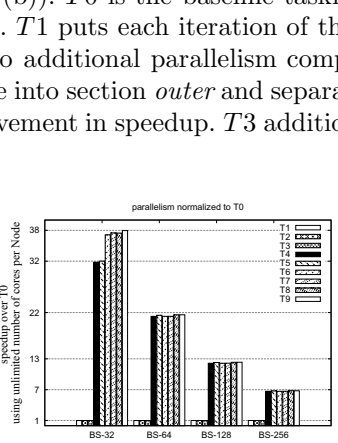
Fig. 6. Taskifications evaluated for HPL and duration and time spent in eac

spends in that function from 95.57% (for $BS = 256$) to 97.83% (for B
On the other hand, Figure 6(c) shows the average duration of each funct
information identifies which function is a good candidate to be broken
smaller tasks. In this example, function *panel_init* is very short so br
into smaller tasks makes little sense. Also, it is important to note that d
BS reduces execution time of most of the functions, so this could also
to make finer-grained execution.

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

		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
machine. Then, the speedup of taskification Ti over taskification $T0$ represents the pa
taskification Ti normalized to taskification $T0$.

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

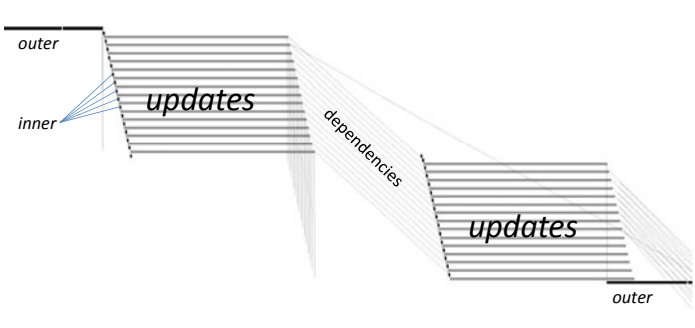
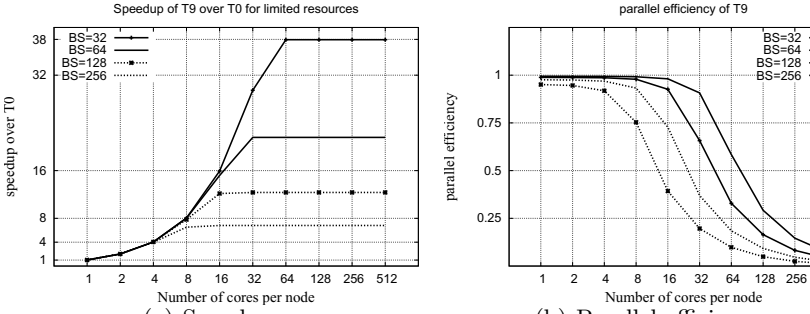


Fig. 8. Paraver visualization of the first 63 tasks and the dependencies am
(taskification T4, BS=256)

thus impedes parallelism. Finally, $T4$ compared to $T2$ separates sect
from function *update* and releases the significant amount of parallelism
it achieves the speedup of 6.76, 12.28, 21.48 and 32.02 for block size
128, 64 and 32, respectively (Figure 7(b)). Also, $T4$ significantly incr
number of tasks in the application to 2.128, 8.336, 33.040 and 131.600
sizes of 256, 128, 64 and 32, respectively (Figure 7(a)). Now, Parav
ization reveals that in $T4$: 1) each section *inner* depends on the sect
in the previous iteration of the inner loop; and 2) each *update* depen
section *inner* in the same iteration of the inner loop. Thus, because sect
is much shorter than *update*, all dependent sections *inner* can execut
and then independent instances of *update* can execute concurrently (F

Further breaking down of *outer*, *inner* and *update* contributes litt
potential speedup (Figure 7(b)). Breaking of *outer*, for block sizes of
and 64, causes slightly higher parallelism of $T5$, $T8$ and $T9$, compare
$T6$ and $T7$. On the other hand, breaking of *inner*, for block size of 3
significantly higher parallelism of $T6$, $T7$, $T8$ and $T9$, compared to $T4$,
effect happens because for very high concurrency of *update* (speedup
than 30), the critical path of the execution moves and starts passing thr
tion *inner*. In these circumstances breaking of *inner* significantly incre
allelism by allowing concurrency of functions *HPL_dlaswp00N*, *HPL_d*
and *HPL_dlaswp06N*. Finally, breaking of *update*, for block size 32, cause
higher parallelism of $T9$ compared to $T8$.

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



(a) Speedup. (b) Parallel efficiency.

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

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

achieves the speedup of 6.80, 12.34, 21.57 and 29.47, respectively. Fur
Figure 9(b) shows parallel efficiency (core utilization) – the ratio bet
application's speedup achieved on some parallel machine and the numbe
in that machine. Adopting that an application efficiently utilizes a m
the parallel efficiency is higher than 75%, the results show that $T9$ w
sizes of 256, 128, 64 and 32, can efficiently utilize the machine of 8, 15,
cores per node, respectively. Therefore, to efficiently employ many-core
with hundreds of cores per node, HPL has to expose even more parall
instance, by making finer-grain taskification with further reduction of b

6 Related Work

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

However, there is little development support for these programm
els. Alchemist tool [16] identifies parts of code that are suitable for th
speculation. Embla [7] estimates the potential speed-up of fork-join bas
lelization. Starsscheck [3] checks correctness of pragma annotations for
family of programming models. Our work adds up to these efforts by de
framework that estimates the potential parallelism of MPI/SMPSs. Furt
our work goes beyond the state-of-the-art tools because: 1) it deals with
execution model that integrates MPI with task-based dataflow execut
allows to study MPI/SMPSs execution before the original MPI appl
ported to MPI/SMPSs; 3) it provides an estimation of the parallelis
configurable target platform; and 4) it provides visualization of the s
execution.

tional parallelism inherent in MPI parallel programs. However, the com
this type of execution impedes an MPI programmer from anticipating h
dataflow parallelism he can obtain in his application. Moreover, it is n
to determine which parts of code should be encapsulated into tasks in
expose the parallelism and still avoid creating unnecessary tasks that
runtime overhead. To address this issue, we have developed a framew
automatically estimates the potential dataflow parallelization in app
We show how, using the framework, one can find optimal taskificati
for any application through a trial-and-error iterative approach that re
knowledge of the studied code. We prove the effectiveness of this app
a case study in which we explore the taskification of High Performance
(HPL). The results show that HPL expresses substantial amount of
dataflow parallelism that allows the application to efficiently utilize
nodes with up to 47 cores per node. Moreover, we show that the glo
tioning significantly impacts parallel efficiency, and thus, in order to
utilize higher number of cores, finer-granularity of execution should be

Acknowledgements. We thankfully acknowledge the support of the
Commission through the HiPEAC-2 Network of Excellence (FP7/ICT
the TEXT project (IST-2007-261580), and the support of the Span
istry of Education (TIN2007-60625, and CSD2007-00050), and the Ge
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
   Y.: Cilk: An Efficient Multithreaded Runtime System. J. Parallel Dist
   put. 37, 55–69 (1996)
3. Carpenter, P.M., Ramirez, A., Ayguade, E.: Starsscheck: A tool to find
   task-based parallel programs. In: D'Ambra, P., Guarracino, M., Talia,
   Euro-Par 2010. LNCS, vol. 6271, pp. 2–13. Springer, Heidelberg (2010)
4. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communicat
   for mpi collective operations. In: PVM/MPI, pp. 39–46 (2000)
5. Kale, V., Gropp, W.: Load Balancing for Regular Meshes on SMPs w
   In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 201
   vol. 6305, pp. 229–238. Springer, Heidelberg (2010)
6. Leijen, D., Hall, J.: Parallel performance: Optimize managed code for
   machines. MSDN Magazine (2007)
7. Mak, J., Faxén, K.-F., Janson, S., Mycroft, A.: Estimating and Exploiting
   Parallelism by Source-Level Dependence Profiling. In: D'Ambra, P., G
   M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 26–37. Springe
   berg (2010)

10. Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based
ming environment for multi-core architectures. In: CLUSTER, pp. 142–
11. Proposed Industry Standard. Openmp: A proposed industry standar
shared memory programming
12. Reinders, J.: Intel threading building blocks: outfitting C++ for multi
cessor parallelism. O'Reilly Media, Inc., Sebastopol (2007)
13. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI:
plete Reference. The MIT Press, Cambridge (1998)
14. Subotic, V., Sancho, J.C., Labarta, J., Valero, M.: A Simulation Framew
tomatically Analyze the Communication-Computation Overlap in Scient
cations. 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 de
distance profiling infrastructure. In: CGO 2009 (2009)