2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Identifying Critical Code Sections in Dataflow Programming Models

Vladimir Subotic, Jose Carlos Sancho
Barcelona Supercomputing Center
Barcelona, Spain

Jesus Labarta, Mateo Valero
Universitat Politecnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain

***Abstract*—The years of practice in optimizing applications point that the major issue is focus – identifying the critical code section whose optimization would yield the highest overall speedup. While this issue is mainly solved for sequential applications, it remains a serious hurdle in the world of parallel computing. Furthermore, the newest dataflow parallel programming models expose very irregular parallelism, making the identification of the critical code section even harder.**

To address this issue, we designed an environment that identifies critical code sections in applications. The programmer can use this environment to estimate the potential benefits of the optimization for a specific parallel platform. This is very important because the programmer can anticipate the benefits of his optimization and assure that the optimization is worth the effort. Furthermore, we showed that in many applications, the choice of the critical code section decisively depends on the configuration of the target machine. For instance, in HP Linpack, optimizing a task that takes 0.49% of the total computation time yields the overall speedup of less than 0.25% on one machine, and at the same time, yields the overall speedup of more than 24% on a machine with different number of cores.

***Keywords*-parallelism; dataflow; MPI; SMPSs;**

I. Introduction

Task-based dataflow programming models showed to be very powerful in exposing high level of parallelism. Dataflow can extract very irregular parallelism. Also, dataflow introduces significant asynchronism in the execution, providing performance that is more tolerant to external contention such as OS noise [16]. Due to all these potential benefits, dataflow became the parallelization strategy for various programming models [17][11].

However, due to irregularity of dataflow parallelism, it is very hard to anticipate and control the parallelism exposed in the application. By varying granularity of execution or task decomposition of the code, the programmer generates different number of tasks, thus changing the potential parallelism of the execution [19]. If he generates too few tasks, the application may expose insufficient parallelism to keep the target machine utilized. However, if he generates too many tasks, the runtime overhead seriously harm performance [16]. Thus, to achieve optimal performance, the programmer must control the parallelism released in the application and suit it for the parallelism offered by the target machine.

Moreover, to fine-tune the execution of his application, the programmer may try to optimize some section of the code – either by rewriting the code of that section, or by executing that section on an accelerator. However, prior to any optimization effort, the programmer must identify **the critical section** – the code section whose optimization would bring the highest reduction of the total execution time for the target system.

In this paper, we describe a new development environment for task-based dataflow programming models. Without losing generality, we focus on the MPI/SMPSs [16] parallel programming model – programming model that integrates MPI [18] and task-based dataflow programming model SMPSs [17]. The environment provides two specific contributions:

1) **Simulating dataflow parallelism**: Up to our knowledge, this is the first simulation framework that can simulate a complex parallel execution that integrates MPI with shared-memory dataflow programming model. The simulation allows varying properties of both the application and the target platform. Furthermore, the environment provides the visualization of the simulated time-behaviors.
2) **Identifying critical code sections**: Based on the designed environment, we devised an automatic approach that identifies the critical code sections of the execution. Compared to the state-of-the-art research, our approach accounts for more influences, and in advance evaluates of the potential benefits of the optimization.

The rest of the paper is organized as follows. Section II gives background on MPI/SMPSs programming model. Section III presents a very simple SMPSs code and identifies its critical sections. Furthermore, Section IV presents the related work on the mainstream approaches in identifying critical sections. In Section V, we describe our development environment, while in Section VI we identify optimization opportunities in a set of parallel applications. Finally, Section VII concludes the paper.

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.15

II. Context: SMPSs programming model

SMPSs [17] 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 (Figure 1). There are two essential annotations needed to port a sequential application to SMPSs:

- *task decomposition* – to mark with pragma statements which functions should be executed as task; and
- *directionality of parameters* – to mark inside pragmas how are the passed arguments used within the taskified function. The specified directionality can be *input*, *output* and *inout*. Apart from the memory regions declared in pragmas, the function should have no side effects.

Given the SMPSs annotations, the runtime can 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 that function as a task and proceeds. Based on the parameters' directionality, the runtime places the obtained task instance in the dependency graph of all tasks. Then, considering the dependency graph, the runtime schedules out-of-order execution of tasks.

In addition, SMPSs provides numerous mechanisms to increment parallelism and optimize the execution. To increase parallelism, the runtime automatically renames data objects to avoid false dependencies (dependencies caused by buffer reuse). The runtime allows configuring the amount of memory used for double buffering, to avoid the hazard of intensive swapping. Also, the runtime allows limiting the dynamic size of the dependency graph, thus avoiding the overhead of maintaining a large graph. Finally, for heterogeneous architectures, SMPSs allows simple semantics to declare that some task should be executed on hardware accelerators.

Dataflow parallelism introduced in SMPSs showed to be powerful in extracting parallelism, and it is finding its place in the standards of the mainstream parallel programming models. Perez *at. el.* [17] demonstrated that in many applications, SMPSs significantly outperforms fork-join based programming models such as OpenMP [9] and Cilk [12]. Based on these promising results, task-based dataflow parallelism steadily enters the OpenMP3.0 standard [5].

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

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

Figure 1. Porting sequential C to SMPSs

A. Integration of SMPSs and MPI

SMPSs integrates with MPI [18] in a manner similar to integration of MPI and OpenMP. The result is a hybrid MPI/SMPSs [16] programming model, in which the work is parallelized across separate address spaces using MPI, while the work of each MPI process is parallelized using SMPSs.

MPI/SMPSs synergizes dataflow execution with message passing, providing high and robust performance. MPI/SMPSs allows a programmer to taskify functions with MPI transfers and thus relate the messaging events to dataflow dependencies. For example, a task with *MPI_Recv* of some buffer gets that buffer from the network and locally stores (*output* pragma directionality) it to the memory. This way, the arrival of the MPI message is related to data-dependencies among tasks. Then, the runtime can schedule SMPSs tasks in a way that overcomes strong synchronization points of pure MPI execution. Marjanovic *at. el.* [16] showed that apart from better peak GFlops/s performance, compared to MPI, MPI/SMPSs delivers better tolerance to network limitations and external perturbations, such as OS jitters.

III. Motivation

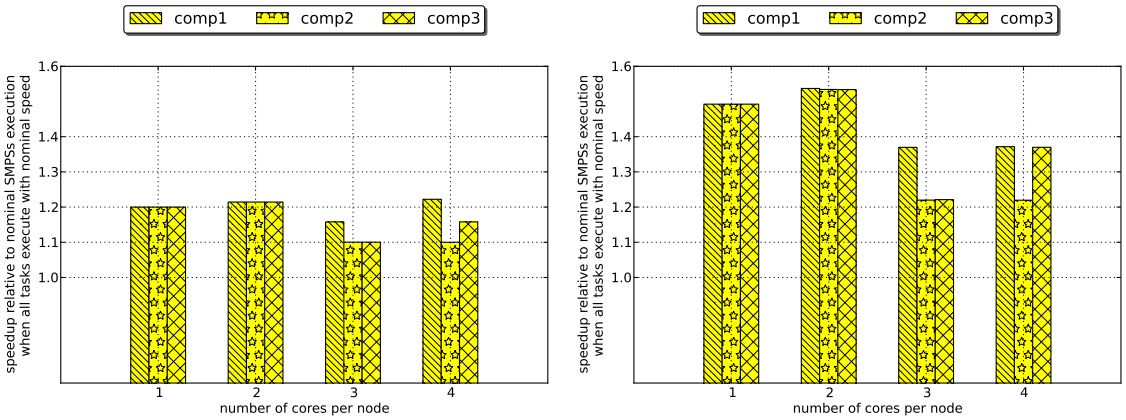
In this section, we present a simple SMPSs code (Figure 2) and explore the potential benefits of optimizing different sections of the code. There are three functions ($comp1$, $comp2$ and $comp3$) of equal duration. These functions are encapsulated into tasks. The code consists of one loop that calls each of the functions in every iteration. Depending on the loop index, each function cyclically changes the buffers it operates on. The only difference among the functions is the

```
#pragma css task  inout(A,B)
void comp1(float *A, float *B);
#pragma css task  inout(A,B)
void comp2(float *A, float *B);
#pragma css task  inout(A,B)
void comp3(float *A, float *B);

#define buf_cnt       19
#define iterations    11
#define step1         9
#define step2         10
#define step3         11

int main () {
    static float  a[buf_cnt][buffer_length];
    for (int i=0; i<iterations; i++) {
        comp1( a[(i*step1)%buf_cnt], a[(i*step1+1)%buf_cnt] );
        comp2( a[(i*step2)%buf_cnt], a[(i*step2+1)%buf_cnt] );
        comp3( a[(i*step3)%buf_cnt], a[(i*step3+1)%buf_cnt] );
    }
}
```

Figure 2. Code of the motivating example.



(a) Speedup when one task is speeded up by a factor of 2

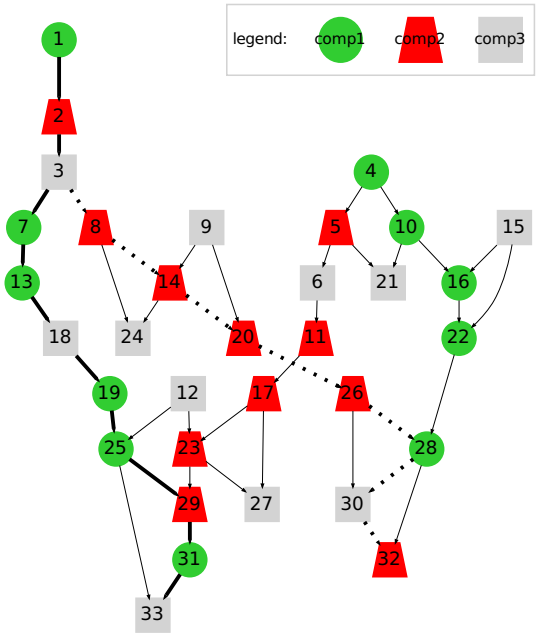
(b) Speedup when one task is speeded up by a factor of 100

Note: The figure shows the resulting execution speedup when some of the tasks is speeded up. The overall speedup is calculated over the original execution – execution in which each task has its original duration.

Figure 3. The resulting speedup when accelerating different tasks.

step ($step1$, $step2$ and $step3$) based on which each function determines which buffers to use.

Although the application is simple, the resulting parallelism is very irregular. The task dependency graph (Figure 4) has sections of low concurrency, as well as sections of high concurrency (e.g. tasks number 7, 8, 9, 5, 10, 15 can all execute concurrently). This irregular parallelism leads to a scalability that is hard to anticipate (Table I). Thus, when executing on 2 cores the speedup is almost ideal. On the other hand, for 3 cores the speedup is ideal, while, surprisingly, adding the fourth core gives no additional speedup.



Note: The numbers in the nodes annotate the ordinal number of the task.

Figure 4. Tasks dependency graph for the motivating example.

Table I
SPEEDUP FOR DIFFERENT NUMBER OF CORES.

cores	1	2	3	4
speedup	1.00	1.94	3.00	3.00

This example shows that Amdahl's law [3] can hardly anticipate potential parallelism of SMPSs. The dependency graph (Figure 4) shows that there are no sequential parts of code. In fact, the graph shows that for each task, there is at least three other independent tasks that can execute concurrently with it. For instance, task number 1 can execute concurrently with tasks 9, 4 and 15. On the other side of the graph, task number 33 can execute concurrently with tasks 27, 26, 21 and 22. However, the experiment shows that in parallel execution on four cores, the overall parallelization cannot be higher than 3.

Although the tasks in the code (Figure 2) appear to be of the same importance, accelerating different tasks yields very different overall speedup (Figure 3). Figure 3(a) shows optimization speedup if one of the tasks is speeded up by a factor of 2. Here, while on the machine with 1 or 2 cores all tasks are equally critical, on the machine with 3 and 4 cores $comp1$ is significantly more critical than the other two tasks. Thus, the choice of the critical task depends on the configuration of the target machine. On the other hand, Figure 3(b) shows optimization speedup if one of the tasks is speeded up by a factor of 100. Now, on the machine with 4 cores, tasks $comp1$ and $comp3$ are equally critical. Thus, the choice of the critical task also depends on the potential acceleration factor. Therefore, in order to identify the critical code section, one has to take into account both the target machine, as well as the acceleration factor of the possible optimization.

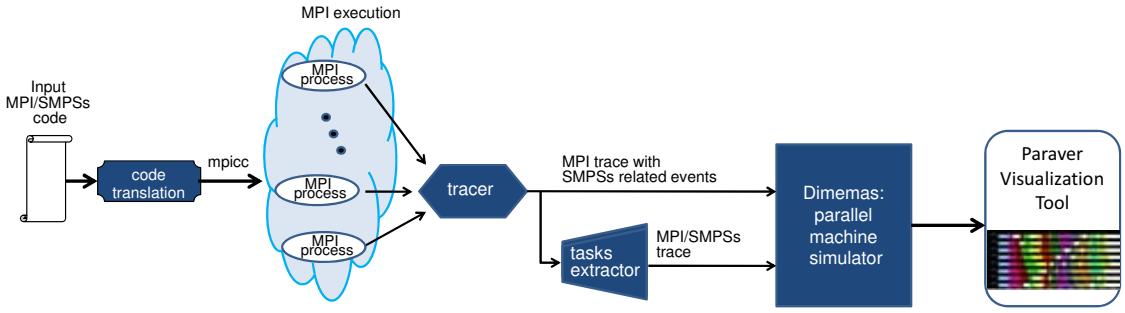


Figure 5. The environment integrates Mercurium code translator, MPISS tracer, tasks extractor, Dimemas simulator and Paraver visualization tool.

IV. Related Work: Motivation example interpreted by the state-of-the-art techniques

Profiling is the most popular technique for identifying critical code sections. The traditional profilers report the percentage of computation time spent in each function. Thus for a sequential application, gprof [14] identifies the most time consuming code section, and that section is automatically the critical section. However, the same information derived from the parallel profilers [15][21] cannot directly determine what is the critical section of the code. As shown in the motivation example from Section III, all functions take the same portion of the total computation time, but not all functions are equally critical.

Other approaches [8][2] promote to the critical section the section that contributes the most to the critical path. In the motivation example, these techniques would identify only one critical path of length 11 (the bold path marked in Figure 4). Then, they would identify $comp1$ as the critical function, since it has 6 instances in the critical path. However, these tools would overlook the path of length 10 (marked with dotted bold lines in Figure 4). This path is slightly shorter, but it has only two instances of task $comp1$. Thus, accelerating $comp1$ would faster reduce the first path than the second one, and the second path would prevail. One of the illustrations of this behavior is that in Figure 3(b), when executing on 1, 2 or 4 cores, accelerating functions $comp1$ and $comp3$ gives the same overall speedup.

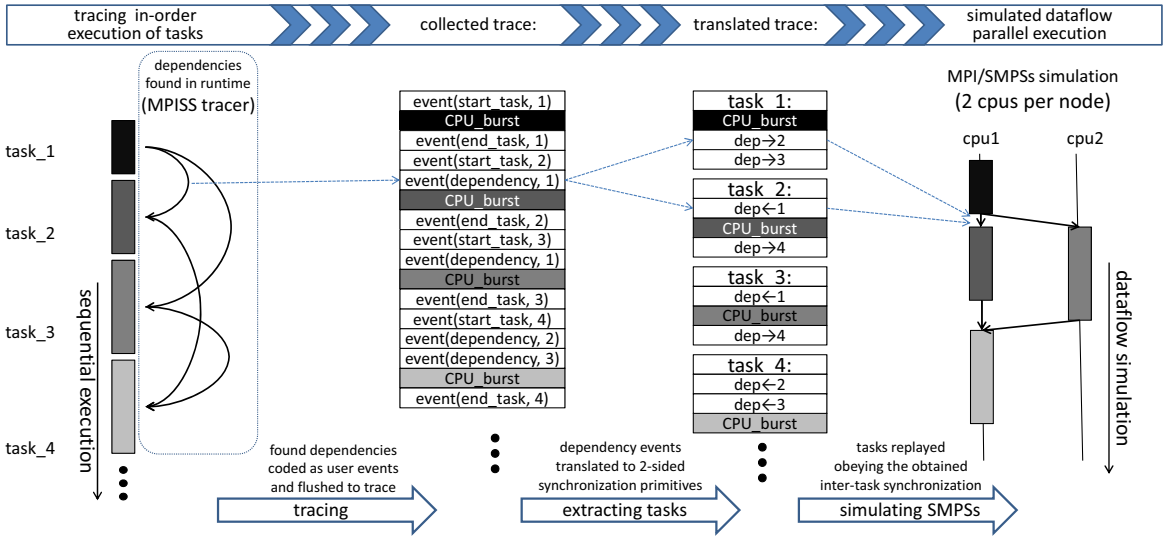
The newest approaches identify the sections of execution with low parallelism and try to find which sections of code to blame. Quartz [4] and Intel Thread Profiler [7] try to quantify the potential parallelism of each section of the code. Similarly, Tallent *at. el.* [20] try to identify threads that are responsible for synchronizations that introduce stalls. Furthermore, they consider different policies of spreading the blame for low parallelization, from contexts in which spin-waiting occurs (victims), to directly blaming a lock holder (perpetrators). However, in the example from Section III, seeing the execution on 3 cores (Table I), these approaches would see no stalls, so they would be unable to identify any part of code as more critical than some other.

We show that the previous approaches cannot capture some influences that are shown to be very decisive in the motivation example. First, that the choice of the critical section depends on the parallel target machine on which the application executes. Second, that depending on the factor of acceleration different sections may be the most beneficial to accelerate. Finally, all the preceding techniques work with "measure-modify" approach – from the profile of one run the technique points to the critical section, the programmer optimizes the section, and then runs again the application "hoping" that the optimization resulted in overall speedup. Conversely, our approach provides anticipation of benefits prior to any optimization effort – the programmer knows the potential optimization speedup before any optimization effort.

V. Environment

The main idea of our methodology is 1) to instrument the execution without task-based dataflow parallelism; and then 2) to reconstruct what would be the execution with task-based dataflow parallelism. The input code executes without task parallelism (ignoring the SMPSs pragma annotations). Still, in runtime, we evaluate pragma annotations and incorporate the evaluation results into the trace of the run. Then, offline, from the collected trace, we reconstruct what would be the trace of the potential run with task parallelism. Finally, we replay the reconstructed trace, by scheduling all tasks in dataflow manner.

Our environment deals with a very complicated parallel execution, offers fast and flexible simulation and provides a rich output. Up to our knowledge this is the first simulation environment that can simulate parallel execution that integrates MPI with task-based programming model. Tracing part of the simulation is very light,primitives reporting to the trace only events related to MPI and SMPSs activity. On the other hand, post-mortem simulation of the parallel execution is very fast, allowing to process 4.000 tasks in less than 10 seconds, and 2 million tasks in about 10 hours. Also, the environment provides conventional techniques for speeding up trace driven simulation, such as filtering and



MPISS tracer intercepts functions inserted by code translation, detects dependencies among tasks and marks these dependencies in the trace (example: at the beginning of $task\_2$ the tracer identifies dependency from $task\_1$, and emits to the trace $event(dependency, 1)$).
Tasks extractor translates dependency user events into two sided synchronizations (example: $event(dependency, 1)$ is translated into outgoing dependency ($dep \rightarrow 2$) in $task\_1$ and incoming dependency ($dep \leftarrow 1$) in $task\_2$).
Dimemas replays the tasks obeying the defined dependencies (example: due to events ($dep \rightarrow 2$) in $task\_1$ and ($dep \leftarrow 1$) in $task\_2$, $task\_2$ cannot start before $task\_1$ finishes).

Figure 6. Environment methodology

sampling. Finally, we believe that the biggest contribution of the environment is its flexibility and rich output. The user can easily change the target platform and visually (qualitatively) inspect how will the dataflow parallelism react. In addition, a simple system of scripts can automatize the process of tracing and simulating, resulting in identification of the critical code section of the application for the target platform.

The environment (Figure 5) consists of Mercurium based **code translator**, **MPISS tracer**, **tasks extractor**, Dimemas **replay simulator** and Paraver **visualization tool**. A **Mercurium based tool** translates the input SMPSs code into the sequential code with inserted functions that annotate pragma primitives. The obtained code executes sequentially. While tracing that execution, **MPISS tracer** emits to the trace additional user events that signal SMPSs primitives. From the collected trace, the **tasks extractor** reconstructs the trace of the potential SMPSs execution. Then, **Dimemas** replays the reconstructed trace to simulate the potential SMPSs execution. Finally, **Paraver** can visualize the simulated execution. In the following subsections, we in detail describe the methodology steps in simulating SMPSs execution. Note that simulating MPI/SMPSs codes differs only in tracing, where the environment traces the corresponding MPI execution, and then extracts SMPSs tasks.

Input code	Translated code
#pragma css task *input(A) output(B)*	
void compute(float *A, float *B) {	void compute(float *A, float *B) {
…	…
}	}
int main () {	int main () {
…	…
	mpisstrace_start_task("compute");
	mpisstrace_input_par("A", a, float);
	mpisstrace_output_par("B", b, float);
compute(a,b); →	compute(a,b);
	mpisstrace_end_task("compute");
…	…
}	}

Figure 7. The code translation inserts functions that signal SMPSs pragma annotations.

A. Code translator

Our Mercurium based tool mark occurrences of SMPSs pragmas (Figure 7). The translated code is a sequential code with empty functions annotating occurrences of SMPSs primitives. Thus, for each section of the execution, the introduced functions specify: 1) to which task that section belongs; and 2) what are input and output parameters of that section. The translated code is executed sequentially and traced with MPISS tracing library.

B. Tracer (MPISS tracing library)

The most important feature of the tracer is to detect dependencies in run-time and mark them into the trace (Figure 6). In addition to the functionality of MPI tracing library, MPISS tracer intercepts the functions inserted in code translation. On intercepting $mpisstrace\_start\_task$ ($mpisstrace\_end\_task$), the tracer emits to the trace an event to mark start (end) of a task. Thus, the collected trace also carries the information of which section of trace belongs to which task. More importantly, on intercepting function

$mpisstrace\_input\_par$ and $mpisstrace\_output\_par$, MPISS tracer calculates data-dependencies among the identified tasks and emits that information into the trace. Thus, the trace obtained with MPISS tracer specifies: 1) how is the sequential execution decomposed into SMPSs tasks; and 2) what are the data-dependencies among the instantiated tasks.

C. Tasks extractor (Trace translator)

From the collected trace, tasks extractor reconstructs the potential trace with task-based dataflow parallelism. For each task, tasks extractor crops from the sequential trace the segment that belongs to that task, and instantiates a new task in the SMPSs trace. Moreover, for each dependency user event from the sequential trace, task extractor creates a two-sided synchronization event in the reconstructed SMPSs trace (Figure 6). Thus, the trace derived with tasks extractor instantiates a separate trace entity for each task, and defines the dependencies among the tasks using semantics that Dimemas simulator can process.

D. Replay simulator

We extended Dimemas [13] in order to simulate MPI/SMPSs execution (Figure 6). We implemented a task synchronization as an intra-node instantaneous MPI transfer that specifies the source and the destination task. This implementation allows Paraver to visualize both MPI communications among processes and data dependencies among tasks. Furthermore, we implemented a new scheduling policy that optimizes MPI/SMPSs execution. The new scheduling policy optimizes parallel execution by favoring tasks that are on the critical path of execution. Furthermore, when simulating MPI/SMPSs execution, the scheduler assigns outstanding priority to task that include MPI calls, and allows these tasks to preempt regular (non-MPI) tasks. The last scheduling feature significantly improves messaging behavior of execution.

VI. Experiments

We partition the experiments section into two parts: one studying SMPSs codes and other studying MPI/SMPSs codes. For each application, we identify sections of code that should be optimized in order to increase parallelism.

A. SMPSs codes

We study four representative application kernels: Jacobi, Cholesky, LU and HM transpose. Jacobi executes on a problem size of 8192 with block size of 128. Cholesky executes on a problem size of 8192 with a block of size 128. LU factorization executes on a problem size of 4096 with the block size of 128. Finally, HM transpose executes on a problem size of 4096 with the block size of 256. For Cholesky and HM transpose, we simulate SMPSs execution of first four iterations of the main loop. For Jacobi, we simulate SMPSs execution on the first 32 iteration of the main loop, because the major source of parallelism comes from concurrency among iterations of the main loop. Finally, for LU factorization, we simulate the entire execution.

Figure 8 shows the distribution of the computation time on different tasks. Applications HM transpose and Jacobi have one very dominant task – HM transpose spends 96.63% of time in *transpose_exchange*, while Jacobi spends 90.54% of time in *jacobi*. Other two applications have two dominant tasks – Cholesky spends 59.29% of time in *sgemm_tile* and 39.12% in *strsm_tile*, while LU factorization spends 54.44% of time in *block_mpy_add* and 38.29% in *bmod*. In a sequential execution of all the four applications, these dominant tasks would be the best candidates for optimization.

However, in the parallel execution, the function that is dominant in the total computation time is not necessarily the function that is critical for the execution time. Figure 9 shows the speedup obtained by speeding up some of the functions by a factor of 2. In HM transpose and Jacobi, for a target machine with a low number of cores, optimizing the identified dominant function brings the greatest benefits. However, as the number of cores in the target machine grows, these benefits significantly drop. Nevertheless, Cholesky and LU show even more drastic behavior. For a machine with a small number of cores per node, the critical function is the most dominant one – *sgemm_tile* for Cholesky, and *block_mpy_add* for LU factorization. Nevertheless, as the target machine gets more cores, other tasks are becoming critical. Thus, in Cholesky, accelerating *sgemm_tile* by a factor of 2 yields the overall speedup of more than 40% if the machine has less than 16 cores, and the same time, yields the overall speedup of less than 2% if the machine has 256 cores (Figure 9(b)). Similarly, in LU factorization, accelerating *block_mpy_add* by a factor of 2 yields the overall speedup of more than 35% if the machine has less than 16 cores, and the same time, yields the overall speedup of less than 3% if the machine has more than 64 cores (Figure 9(c)).

Identification of the critical tasks can be brought in relation with the parallelization pattern of the application. In HM transpose and Cholesky, the dominant task is the critical one. In HM transpose, speeding up task that takes a small portion of computation time has no significant effect on the execution of the application (Figure 10(b)). On the other hand, speeding up $transpose\_exchange$ by a factor of 2 (Figure 10(c)), significantly reduces both the computation and the execution time. It is interesting to note that the peak parallelism drops, as well. This happens because, as $transpose\_exchange$ lasts shorter, drops the probability of concurrent execution of different instances of that task.

Conversely, in Jacobi and LU factorization, a task that takes small share of the total computation time may become a task that is critical for the total execution time. In LU factorization (Figure 11(a)), the total computation time is $47.29s$, while the total execution time is $1.55s$. When speeding up $block\_mpy\_add$ by a factor of 2, the computation

HM transpose	
task name	percentage [%]
main_task	0.61
transpose	2.77
transpose_exchange	96.63

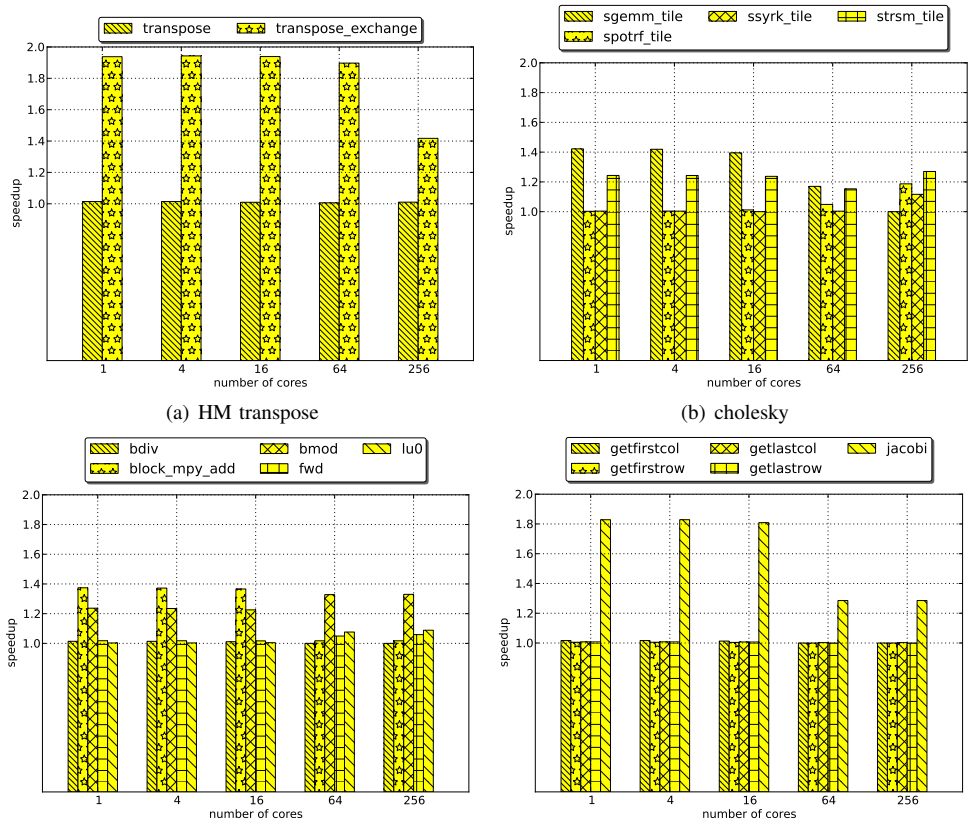
Cholesky	
task name	percentage [%]
main_task	0.33
sgemm_tile	59.29
spotrf_tile	0.52
ssyrk_tile	0.75
strsm_tile	39.12

LU factorization	
task name	percentage [%]
main_task	0.28
bdiv	2.76
block_mpy_add	54.44
bmod	38.29
fwd	3.49
lu0	0.53

Jacobi	
task name	percentage [%]
main_task	2.35
getfirstcol	3.18
getfirstrow	0.98
getlastcol	1.48
Getlastrow	1.46
jacobi	90.54

Note: The tables suppress functions that take less 0.2% of the total computation time of the application.

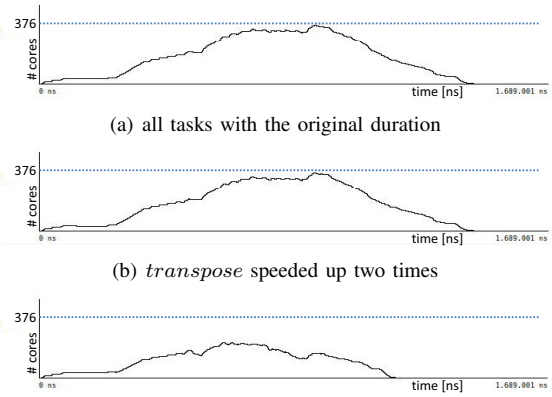
Figure 8. Tasks profile



(a) HM transpose (b) cholesky

(c) LU factorization (d) jacobi

Figure 9. Speedup when one task is speeded up by 2x (SMPSs codes).

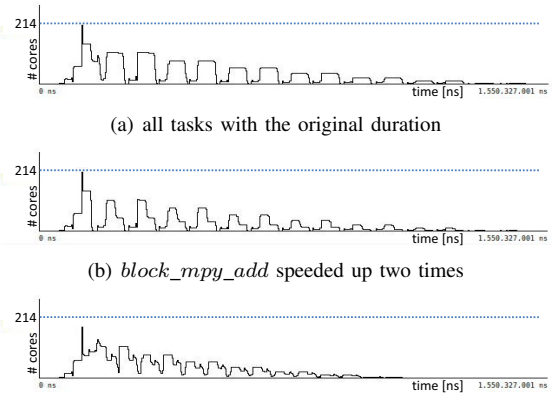


(a) all tasks with the original duration

(b) $transpose$ speeded up two times

(c) $transpose\_exchange$ speeded up two times

Figure 10. HM transpose: number of active cores



(a) all tasks with the original duration

(b) $block\_mpy\_add$ speeded up two times

(c) $bmod$ speeded up two times

Figure 11. LU factorization: number of active cores

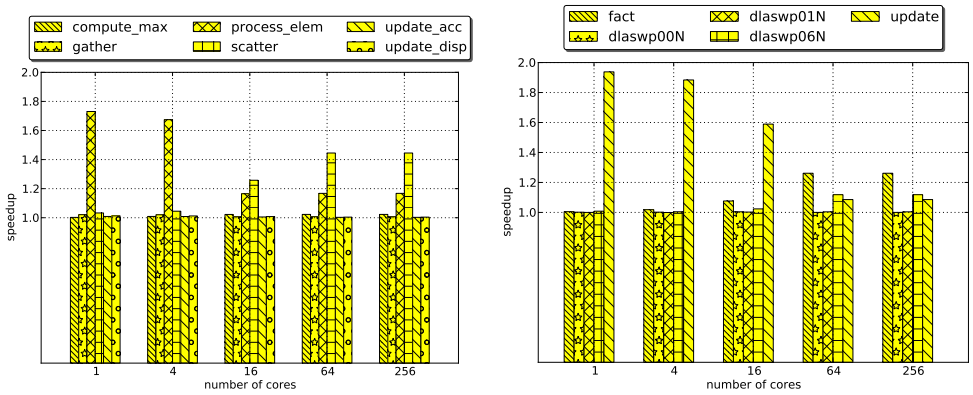
Note: These Paraver views show the number of cores that are active in each moment of execution. All the applications execute with the unlimited number of cores on the target machine. It is interesting to note that the surface below the plot represents the application's computation time, while the total length of the plot represents the application's total execution time.

time drops for $\frac{54.44\%}{2} = 27.22\%$, while the total execution time drops for only 1.97% (Figure 11(b)). On the other hand, when speeding up $bmod$ by a factor of 2, the computation time drops for $\frac{38.29\%}{2} = 19.14\%$, while the total parallel execution time drops for 32.48% (Figure 11(c)). The figures show that speeding up $block\_mpy\_add$ shrinks the regions with high parallelism. On the other hand, speeding up $bmod$ shrinks the regions with low parallelism, thus bringing the regions with high parallelism closer together. Therefore, a small reduction in the total computation time can cause a significant reduction in the total execution time.

SPECFEM-3D	
task name	percentage [%]
compute_max	0.46
gather	4.15
process_elem	84.60
scatter	6.37
update_acc	1.75
update_disp	2.55

HP Linpack	
task name	percentage [%]
fact	0.49
dlaswp00N	0.34
dlaswp01N	0.63
dlaswp06N	1.29
update	96.99

Figure 12. Tasks profile



(a) SPECFEM-3D (b) HP Linpack

Figure 13. Speedup when one task is speeded up by 2x (MPI/SMPSs codes).

B. MPI/SMPSs codes

In the study of MPI/SMPSs applications, we use the codes of HP Linpack [10] and SPECFEM-3D [6]. Linpack is one of the most famous parallel applications used to rank parallel machines on the top 500 list [1]. We run it for the problem size of 65536 and block size of 128. SPECFEM-3D simulates earthquakes in complex three-dimensional geological models. We run it with the input data $NSPEC = 198352$ and $NGLOB = 12912201$. We simulate only four iterations of the main loop for each application. Both applications execute with 16 MPI process with each MPI process running on a separate node in the machine.

Finally, MPI/SMPSs applications especially emphasize the phenomenon where a task that takes small portion of the computation time becomes the critical task in the execution. In SPECFEM-3D, $scatter$ takes only 6.37% of the total computation time (Figure 12). Thus, speeding up $scatter$ by a factor of 2 reduces the total computation time for $\frac{6.37\%}{2} = 3.14\%$. However, when the application executes with 64 cores per node, this small reduction of total computation time results in the overall execution speedup of 44.16% (Figure 13(a)).

HP Linpack expresses even more extreme behavior. The dominant task in HP Linpack ($update$), takes 96.99% of the total execution time. Speeding up $update$ by a factor of 2, reduces the computation time for $\frac{96.99\%}{2} = 48.50\%$. When HP Linpack executes with only 1 core per node, this reduction of computation time results in the overall speedup of 95.46%. However, when HP Linpack executes with 64 cores per node, the same reduction of the computation time results in less than 10% of overall speedup. On the other hand, $fact$ is the fourth most dominant function, taking only 0.49% of the total computation time (Figure 12). Speeding up $fact$ by a factor of 2 reduces the computation time for $\frac{0.49\%}{2} = 0.24\%$. Still, when HP Linpack executes with 64 cores per node, this small reduction in computation time brings the overall application speedup of 24.54% (Figure 13(b)).

This effect happens because task instances of $update$ are highly parallelizable, while task instances of $fact$ are extremely non-parallelizable. Although $update$ takes big portion of the computation time, instances of $update$ parallelize across available cores without throttling the execution. On the other hand, $fact$ takes a small portion of computation time, but instances of $fact$ cannot execute concurrently on different cores. Furthermore, $fact$ has numerous MPI communications so execution of some instance of $fact$ in one MPI process also may condition some other instance of $fact$ in other MPI process.

VII. Conclusion

Task based dataflow programming models potentially extract very irregular parallelism, making it hard to estimate the potential parallelism in applications. For real application with thousands of tasks, this estimation exceeds the prediction ability of any human programmer. To tackle this problem, we design an environment that quickly estimates the possible parallelization of an application on a parallel platform. Furthermore, the environment pinpoints the causes that limit the dataflow parallelism. These two features of the environment can significantly facilitate development and optimization of applications based on dataflow programming models.

We showed how, using the designed environment, a programmer can easily identify the critical task – a task that should be optimized in order to increase the scalability of the code. We identified that many parallel applications express the phenomenon in which a task that takes a small portion of the total computation time, may decisively contribute to the total parallel execution time. In particular, in the case of HP Linpack, function $fact$ takes only 0.49% of the total computation time. However, when the application executes

with 16 MPI processes, with 64 cores for each MPI process, speeding up task $fact$ by a factor of 2 results in the overall execution speedup of 24.54%. Also, we observed that this phenomenon is especially significant at large-scale.

Our environment provides very fast and flexible simulation, so various influences on dataflow can be studied quickly. Moreover, the environment allows visualization support so the programmer can qualitatively inspect the simulated dataflow execution. Therefore, we believe that our environment can be of great help to all newcomers to the dataflow programming models.

ACKNOWLEDGMENT

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] M. Agarwal and M. I. Frank. Spartan: A software tool for parallelization bottleneck analysis. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 56–63, Washington, DC, USA, 2009. IEEE Computer Society.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 115–125, New York, NY, USA, 1990. ACM.

[5] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In *IWOMP*, pages 1–12, 2007.

[6] L. Carrington, D. Komatitsch, M. Laurenzano, M. M. Tikir, D. Michéa, N. L. Goff, A. Snavely, and J. Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In *SC*, page 60, 2008.

[7] I. Corporation. Intel thread profiler. http://software.intel.com/file/17321.

[8] I. Corporation. Intel vtune amplifier xe. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[9] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[10] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Memory - Sequoia: programming the memory hierarchy. In *SC*, page 83, 2006.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[13] S. Girona, J. Labarta, and R. M. Badia. Validation of Dimemas Communication Model for MPI Collective Operations. In *PVM/MPI*, pages 39–46. 2000.

[14] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[15] A. N. Laboratory. The FPMPI-2 MPI profiling library. http://www-unix.mcs.anl.gov/fpmpi/.

[16] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *ICS*, pages 5–16, 2010.

[17] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *CLUSTER*, pages 142–151. 2008.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI: The Complete Reference. In *The MIT Press*. 1998.

[19] V. Subotic, R. Ferrer, J. C. Sancho, J. Labarta, and M. Valero. Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications. In *Euro-Par (1)*, pages 39–51, 2011.

[20] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPOPP*, pages 269–280, 2010.

[21] J. Vetter and C. Chambreau. The mpiP MPI profiling library. http://mpip.sourceforge.net/.