A Taxonomy of Task-Based Technologies for High-Performance Computing

Peter Thoman[1], Khalid Hasanov[2], Kiril Dichev[3], Roman Iakymchuk[4], Xavier Aguilar[4], Philipp Gschwandtner[1], Erwin Laure[4], Herbert Jordan[1], Pierre Lemarinier[2], Kostas Katrinis[2], Dimitrios S. Nikolopoulos[3], Thomas Fahringer[1]

[1] University of Innsbruck, 6020 Innsbruck, Austria
{petert,philipp,herbert,tf}@dps.uibk.ac.at
[2] IBM Ireland, Dublin 15, Ireland {khasanov,pierrele,katrinisk}@ie.ibm.com
[3] Queen's University of Belfast, Belfast BT7 1NN, United Kingdom
{K.Dichev,D.Nikolopoulos}@qub.ac.uk
[4] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
{riakymch,xaguilar,erwinl}@kth.se

Abstract. Task-based programming models for shared memory, for example OpenMP and Cilk, have existed for decades, and are well documented. However, with the increase in heterogeneous, many-core and parallel systems, a number of research-driven projects have developed more diversified task-based support, employing various programming and runtime features. Unfortunately, despite the fact that dozens of different task-based systems exist today and are actively used for parallel and high-performance computing, no comprehensive overview or classification of task-based technologies for HPC exists.
In this paper, we provide an initial task-focused taxonomy for HPC technologies, which covers both programming interfaces and runtime mechanisms. We demonstrate the usefulness of our taxonomy by classifying state-of-the-art task-based environments in use today.

Keywords: Task-based parallelism, taxonomy, API, runtime system, scheduler, monitoring framework, fault tolerance.

1 Introduction

A large number of task-based programming environments have been developed over the past decades, and even well-established programming languages like C++ have now integrated tasks, allowing their use for shared memory parallelism. For the purpose of this work, we define a *task* as follows

> A **task** is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.

The Cilk language [19] allows task-focused parallel programming, and is an early example of efficient task scheduling via work stealing. Language extensions like OpenMP [4] (since version 3.0) integrate tasks into their programming interface. Industry-standard and well-supported parallel libraries based on task parallelism

have emerged, such as Intel TBB [24]. Task-based environments for heterogeneous hardware have also naturally developed with the emergence of accelerator and GPU computing; StarPU [8] is an example of such an environment.

In addition, task-based parallelism is increasingly employed on distributed memory systems, which constitute the most important target for high-performance computing (HPC). In this context, tasks are often combined with a global address space (GAS) programming model, and scheduled across multiple processes, which together form the *distributed execution* of a single task-parallel program. While some examples of global address space environments with task-based parallelism are specifically designed languages such as Chapel [6] and X10 [18], it is also possible to implement these concepts as a library. For instance, HPX [10] is an asynchronous GAS runtime, and Charm++ [23] uses a global object space.

This already very diverse landscape is made even more complex by the recent appearance of task-based runtimes using novel concepts, such as the data-centric programming language Legion [1]. Many of these task-based programming environments are maintained by a dedicated community of developers, and are often research-oriented. As such, there might be relatively little accessible documentation of their features and inner workings.

Crucially, at this point, *there is no up to date and comprehensive taxonomy and classification of existing common task-based environments.* This makes it very difficult for researchers or developers with an interest in task-based HPC software development to get a concise picture of the alternatives to the omnipresent MPI programming model. In this work, we attempt to address this issue by providing a taxonomy and classification of both state-of-the-art task-based programming environments and more established alternatives. We consider a task-based environment as consisting of two major components: a *programming interface* (API) and a *runtime system*; the former is the interface that a given environment provides to the programmer, while the latter encompasses the underlying implementation mechanisms. We present a set of API characteristics allowing meaningful classification in Section 2. For discussing the more involved topic of runtime mechanisms, we further structure our analysis into the overarching topics of scheduling, performance monitoring, and fault handling (see Section 3). Finally, based on the taxonomy introduced, we classify and categorize existing APIs and runtimes in Section 4.

2 Task-Parallel Programming Interfaces (APIs)

The Application Programming Interface (API) of a given task-parallel programming environment defines the way an application developer describes parallelism, dependencies, and in many cases other more specific information such as the work mapping structure or data distribution options. As such, finding a way to concisely characterize APIs from a developer's perspective is crucial in providing an overview of task-parallel technologies.

In this work, we define a set of characterizing features for such APIs which encompasses all relevant aspects while remaining as compact as possible. A subset of these features was adapted from previous work by Kasim et al. [11]. To these existing characteristics we added additional information of general importance – such as technological readiness levels – as well as features which relate to new

capabilities particularly relevant for modern HPC like support for heterogeneity and resilience management. We will now define each of these characteristics and their available options for categorization. Note that *explicit* (**e**) support generally refers to features which are supported, but require extra effort or implementation by the application developer, while *implicit* (**i**) support means that the toolchain manages the feature automatically given a default representation of the program in the API.

Technology Readiness The technology readiness of the given API and its implementations according to the European Commission definition.[5]

Distributed Memory Whether targeting distributed memory systems is supported. Options are *no* support, *explicit* support, or *implicit* support. *explicit* refers to, for example, message passing between address spaces, while automatic data migration would be an example of *implicit* support.

Heterogeneity Indicates whether tasks can be executed on accelerators (e.g. GPUs). Again, *explicit* and *implicit* as well as *no* support are possible, where the former means that the application developer has to actively provision tasks to run on accelerators, using a distinct API.

Worker Management Whether the worker threads and/or processes need to be started and maintained by the user (*explicit*) or are provided automatically by the environment (*implicit*).

Task Partitioning This feature indicates whether each task is atomic – can, thus, only be scheduled as a single unit – or can be subdivided/split.

Work Mapping Describes the way tasks are mapped to the existing hardware resources. Possibilities include *explicit* work mapping, *implicit* work mapping (e.g. stealing), or *pattern-based* work mapping.

Synchronization Whether tasks are synchronized in an *implicit* fashion, e.g. by regions or the function scope, or *explicitly* by the application developer.

Resilience Management Describes whether the API has support for task resilience management, e.g. fine-grained checkpointing and restart.

Communication Model Either *shared memory* (smem), *message passing* (msg), or *Partitioned Global Address Space* (pgas).

Result Handling How the tasking model supports handling the results of task computation: *implicit* via write-back to existing data, or explicitly provided task result types (which might, for example, be accessed as *futures*).

Graph Structure The type of task graph dependency structure supported by the given API: a *tree* structure, an *acyclic graph* (dag) or an *arbitrary graph*.

Task Cancellation Whether the tasking model supports cancellation of tasks: *no* cancellation support; cancellation is supported either *cooperatively* (only at task scheduling points) or *preemptively*.

Implementation Type How the API is implemented and addressed from a program. A tasking API can be provided either as a *library*, a *language extension*, e.g. pragmas, or an entire *language* with task integration.

[5] `https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf`. Accessed: 2017-05-03

3 Many-Task Runtime Systems

Many-task runtime systems serve as the basis for implementing these APIs, and are considered a promising tool in addressing key issues associated with Exascale computing. In this section we provide a taxonomy of many-task runtime systems, which is summarized and illustrated in Figure 1.

A crucial difference among various many-task runtime systems is the **target architecture** they support. The evolution of many-task runtime systems started from *homogeneous shared-memory* computers with multiple cores and continued with runtimes for *heterogeneous* shared-memory and/or *distributed-memory* systems. The way different runtimes support distributed-memory systems is not uniform in terms of distribution of computations across the nodes. In case of *implicit data distribution*, data distribution is handled by the runtime, without putting any burden on the application developer. On the other hand, in *explicit data distribution*, distribution across the nodes is explicitly specified by the programmer.

The increase in the number and type of compute units in HPC systems naturally requires efficiency not only in total execution times of applications, but also in power and/or energy. Thus, whether the runtime provides additional **scheduling objectives** other than the total execution time is another important distinction. At the same time, there is not a single standard **scheduling methodology** that is being used by all many-task runtime systems. Some of them provide automatic scheduling within a single shared-memory machine while the application developer needs to handle distributed-memory execution explicitly, while others provide uniform scheduling policies across different nodes.

Many-task runtimes may require **performance introspection** and **monitoring** to facilitate implementation of different scheduling policies. While traditionally it was not part of runtimes, requirements for on-the-fly performance information have surfaced. Thus, most task-based runtimes already provide and make use of introspection capabilities.

Fault tolerance is another key factor that is important in many-task runtime systems in the context of Exascale requirements. As detailed in Section 3.3, a runtime may have no resilience capabilities, or it may target task faults or even process faults.

3.1 Scheduling in Many-Task Runtime Systems

Task Scheduling Targets Depending on the capabilities of the underlying many-task runtime system, its scheduling domain is usually limited to a single shared-memory homogeneous compute node, a heterogeneous compute node with accelerators, homogeneous distributed-memory systems of interconnected compute nodes, or in a most generic form to heterogeneous distributed-memory systems. By supporting different types of heterogeneous architectures, the runtime can facilitate source code portability and support transparent interaction between different types of computation units for application developers.

Traditionally, the *execution time* has been the main objective to minimize for different scheduling policies. However, the increasing scale of HPC systems makes it necessary to take the energy and power budgeting of the target system into account as well. Therefore, some many-task runtime systems have already started

Fig. 1: Taxonomy of Many-Task Runtime Systems.

providing *energy-aware* [14] scheduling policies[6]. In addition, recent research projects, such as AllScale[7] focus on *multi-objective* scheduling policies trying to find optimal trade-offs among conflicting optimization objectives like execution time, energy consumption and/or resource utilization.

Task Scheduling Methods Extensive research has been conducted in task scheduling methodologies. We do not try to list all different techniques for task scheduling, but rather highlight methods used in state-of-the-art many-task runtime systems. The task scheduling problem can be addressed either in *static* or *dynamic* way. In the former case, depending on the decision function it is assumed that either one or more of the following inputs are known in advance: the execution times of each task, inter-dependencies between tasks, task precedence, resource usage of each task, the location of the input data, task communications, and synchronization points. This is by no means an exhaustive list but it gives an indication of the multiple possible a priori inputs for static scheduling. Using all this information the scheduling can be performed offline during com-

[6] http://starpu.gforge.inria.fr/doc/html/Scheduling.html#Energy-basedScheduling
[7] The AllScale EC-funded FET-HPC project: `allscale.eu`.

pilation time. On the other hand, dynamic scheduling is mainly used in the case where there is not enough information in advance or obtaining such information is not trivial. Additionally, *hybrid* policies which integrate static and dynamic information are possible.

Most static scheduling algorithms used in many-task runtime systems are based on the *list scheduling* methods. Here, it is assumed that the scheduling list of tasks is statically built before any task starts executing and the sequence of the tasks in the list is not modified. The list scheduling approach can easily be adapted and used for dynamic scheduling by re-computing and re-sequencing the list of tasks. As a matter of fact, heuristic policies based on list scheduling and performance models are employed in some many-task runtime systems [8].

Work-stealing [2] can be considered as the most widely used dynamic scheduling method in task-based runtime systems. The main idea in work-stealing is to distribute tasks between per-processor work queues, where each processor operates on its local queue. The processors can steal tasks from other queues to perform *load-balancing*. There are two main approaches in implementing work-stealing, namely, *child-stealing* and *parent-stealing*. In parent-stealing, which is also called *work-first* policy, a worker executes a spawned task and leaves the continuation to be stolen by another worker. Child-stealing, which is also called *help-first* policy, does the opposite, namely, the worker executes the continuation and leaves the spawned task to be stolen by the other workers. Another approach to dynamic scheduling for many-task runtime systems is the **work-sharing** strategy. Unlike the work-stealing, it schedules each task onto a processor when it is spawned and it is usually implemented by using a *centralized task pool*. In work-sharing, whenever a worker spawns a new task, the scheduler migrates it to a new worker to improve load balancing. As such, migration of tasks happens more often in work-sharing than that of in work-stealing.

Few of the existing many-task runtime systems provide *energy efficient* scheduling policies. In the primitive case it is assumed that the application can provide an energy consumption model which can be used by a scheduling policy as part of its objective function. In more advanced cases, the runtime provides offline or online profiling data, such as, instructions per cycle (IPC) and last level cache misses (LLCM). This data is used to build a look-up table that maps each frequency setting with the triple of IPC, LLCM, and the number of active cores. Then, a scheduling decision based on this information [14].

3.2 Performance Monitoring

The high concurrency and dynamic behavior of upcoming Exascale systems poses a demand for performance observation and runtime introspection. This performance information is very valuable to guide HPC runtimes in their execution and resource adaption, thereby maximizing application performance and resource utilization.

When targeting performance observation, performance monitoring software is either generating data to be used **online** [7,1,13,15,16] or **offline** [15,8,5,1]. In other words, whether the collected data is going to be used while the application still runs or after its execution. Furthermore, this taxonomy can be extended with respect to who is consuming data – either the end user (*performance analysis*)

or the runtime itself (*introspection* and *historical data*). Real-time performance data (*introspection* and performance models from *historical data*) will play an important role in Exascale for runtime adaptation and optimal task-scheduling.

3.3 Task, Process, and System Faults

For this topic, we extend a recent taxonomy [22] from the HPC domain to include the concept of task faults. We retain detectability of faults as the main criterion, but distinguish three levels of the system: distributed execution, process, and task (see Figure 2). Each of these levels may experience a fault, and each of them has a different scope.

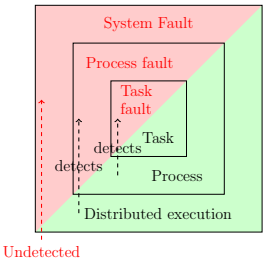

Fig. 2: A taxonomy of faults based on the detection capabilities: task faults, process faults, and system faults.

Task Faults: Tasks have the smallest scope of the three; still, a failure of a task may affect the result of a process, and subsequently of a distributed run. A typical example are undetected errors in memory. The process which runs a task is generally capable of detecting task faults. There are several examples of shared-memory runtimes, where task faults within parallel regions have been detected and corrected [20,17].

Process Faults: A process may also fail, which leads to the termination of all underlying tasks. For example, a node crash can lead to a process failure. In such a scenario, a process cannot detect its failure; however, in a distributed run, another process may detect the failure, and trigger a recovery strategy across all processes. A recovery strategy in this case may rely on one of two redundancy techniques: checkpoint/restart or replication.

System Faults: On the last level, a distributed system execution may fail in cases of severe faults like switch failure, or power outage. In this case, a failure cannot be detected. No recovery strategy can be applied in such scenarios.

4 Classification

Table 1 classifies the existing task-parallel APIs according to the API taxonomy (see Section 2), however with additional clarifications. First, for an API to support a given feature, this API must not require the user to resort to third party libraries or implementation-specific details of the API. For instance, some APIs offer arbitrary task graphs via manual task reference counting [21]. This does not qualify as support in our classification. Second, all APIs shown as featuring task cancellation do so in a non-preemtive manner due to the absence of OS-level preemption capabilities.

Some entries require additional clarification. In C++ STL, we consider the entity launched by `std::async` to represent a task. Also, while StarPU offers shared memory parallelism, it is capable of generating MPI communication from a given task graph and data distribution [8], hence it is marked with explicit support for distributed memory using a message-based communication model. Furthermore, PaRSEC includes both a task-based runtime that works on user-specified task graph and data distribution information, as well as a compiler that

	Technological Readiness	Distributed Memory	Heterogeneity	Worker Management	Task Partitioning	Work Mapping	Synchronization	Resilience Management	Communication Model	Result Handling	Graph Structure	Task Cancellation	Implementation Type
C++ STL	9	×	×	**i**	×	**i/e**	**e**	×	smem	**i/e**	dag	×	Library
TBB	8	×	×	**i**	×	**i**	**i**	×	smem	**i**	tree	✓	
HPX	6	**i**	**e**	**i**	✓	**i/e**	**e**	×	pgas	**e**	dag	✓	
Legion	4	**i**	**e**	**i**	✓	**i/e**	**e**	×	pgas	**e**	tree	×	
PaRSEC	4	**e**	**e**	**i**	×	**i/e**	**i**	✓	msg	**e**	dag	✓	
OpenMP	9	×	**i**	**e**	×	**i**	**i/e**	×	smem	**i**	dag	✓	Extension
Charm++	6	**i**	**e**	**i**	✓	**i/e**	**e**	✓	pgas	**i/e**	dag	×	
OmpSs	5	×	**i**	**i**	×	**i**	**i/e**	✓	smem	**i**	dag	×	
StarPU	5	**e**	**e**	**i**	✓	**i/e**	**e**	×	msg	**i**	dag	×	
Cilk	8	×	×	**i**	×	**i**	**e**	×	smem	**i**	tree	×	Lang.
Chapel	5	**i**	**i**	**i**	✓	**i/e**	**e**	×	pgas	**i**	dag	×	
X10	5	**i**	**i**	**i**	✓	**i/e**	**e**	✓	pgas	**i**	dag	×	

Table 1: Feature Comparison of APIs for Task Parallelism.

accepts serial input and generates this data. As the latter is limited to loops, we only consider the runtime in this work.

Several observations can be made from the data presented in Table 1. First, all APIs with distributed memory support also allow task partitioning and support heterogeneity in some form. APIs offering implicit distributed memory support employ a global address space. Second, among APIs lacking distributed memory, only OmpSs offers resilience (via its Nanos++ runtime), and distributed memory APIs only recently started to include resilience support [3] – likely driven by the continuous increase in machine sizes and hence decreased mean-time-between-failures. Finally, some form of heterogeneity support is provided in almost all modern APIs, though it often requires explicit heterogeneous task provisioning by the programmer.

Table 2 provides the corresponding classification with respect to the runtime system and its subcomponents (see Section 3). It is worth mentioning that there are various contributions extending runtime features, but these contributions are not part of the main release yet. We do not consider such extended features in our taxonomy. For instance, recent work in X10 [12] extends the X10 scheduler with distributed work-stealing algorithms across nodes; however, we classify X10 as not (yet) having a distributed scheduler. The same applies to StarPU and OmpSs. Namely, new distributed-memory scheduling policies are being developed for both runtimes, however, they are not part of their main release yet[8]. Also, for Chapel, X10, and HPX, there is automatic data distribution support (runtime feature); however, these runtimes require explicit work mapping in distributed memory environments (API feature).

[8] We received feedback from their developers

	Data Distribution	Scheduling on shared memory	Scheduling on distr. memory	Performance Monitoring	Fault Tolerance	Target Architecture
OpenMP	×	**m**	×	**off/on**	×	**sm**
TBB	×	**ws**	×	**off**	×	**sm**
Cilk-Plus	×	**ws**	×	**off**	×	**sm**
StarPU	**e**	**m**	×	**off/on**	×	**d**
OmpSs	**i**	**m**	×	**off/on**	**tf**	**d**
Charm++	**i**	**m**	**m**	**off/on**	**pf**	**d**
X10	**i**	**ws**	×	**off**	**pf**	**d**
Chapel	**i**	**ws**	×	**off**	**pf**	**d**
HPX	**i**	**ws**	×	**off/on**	×	**d**
ParSEC	**e**	**m**	**l**	**off**	**tf**	**d**
Legion	**i**	**ws**	**ws**	**off/on**	**pf**	**d**

i implicit
e explicit
m multiple (incl. ws)
l limited
ws work stealing
tf task faults
pf process faults
sm shared memory
d distributed memory
on online use
off offline (post-mortem) use

Table 2: Feature Comparison of Runtimes for Task Parallelism.

Most of the runtime systems have similarities in scheduling within a single shared-memory node and work-stealing is the most common method of scheduling. On the other hand, there is no established method for inter-node scheduling. For instance, ParSEC [9] only provides a limited inter-node scheduling based on remote completion notifications, while Legion uses distributed work-stealing.

5 Conclusions

The shift in HPC towards emerging task-based parallel programming paradigms has led to a broad ecosystem of different task-based technologies. With such diversity, and some degree of isolation between individual communities of developers, there is a lack of documentation and common classification, thus hindering researchers to have a complete view of the field. In this paper, we provide an initial attempt to establish a common taxonomy and provide the corresponding categorization for many existing task-based programming environments.

We divided our taxonomy into two broad categories: *API characteristics*, which define how the programmer interacts with the system; and *many-task runtime systems*, classifying the underlying technologies. For the latter, we analyze the types of scheduling policies and goals supported, online and offline performance monitoring integration, as well as the level of resilience and detection provided for task, process and system faults.

Acknowledgement

This work was partially supported by the AllScale project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671603.

References

1. M. E. Bauer. *Legion: programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.
2. R.D. Blumofe and C.E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
3. D. Cunningham. Resilient x10: Efficient failure-aware programming. In *Proceedings of PPoPP14*, pages 67–80. ACM, 2014.
4. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
5. A. Duran et al. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
6. B. L. Chamberlain et al. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
7. C. Augonnet et al. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par 2009*, pages 56–65. Springer, 2009.
8. C. Augonnet et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
9. G. Bosilca et al. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45, Nov 2013.
10. H. Kaiser et al. Hpx: A task based programming model in a global address space. In *PGAS'14*, page 6. ACM, 2014.
11. H. Kasim et al. Survey on parallel programming model. In *Proceedings of NPC'08*, pages 266–275. Springer, 2008.
12. J. Paudel et al. On the merits of distributed work-stealing on selective locality-aware tasks. In *2013 42nd International Conference on Parallel Processing*, pages 100–109, Oct 2013.
13. J. Planas et al. Self-adaptive ompss tasks in heterogeneous environments. In *IPDPS13*, pages 138–149. IEEE, 2013.
14. J.C. Meyer et al. Implementation of an Energy-Aware OmpSs Task Scheduling Policy. `http://www.prace-ri.eu/IMG/pdf/wp88.pdf`. Accessed: 2017-05-02.
15. K. Huck et al. An early prototype of an autonomic performance environment for exascale. In *Proceedings of ROSS13*, page 8. ACM, 2013.
16. K. Huck et al. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66, 2015.
17. O. Subasi et al. Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In *PDP'15*, pages 99–102, 2015.
18. Ph. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA05*, pages 519–538. ACM, 2005.
19. R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.
20. S. Hukerikar et al. Opportunistic application-level fault detection through adaptive redundant multithreading. In *Proceedings of HPCS14*, pages 243–250, 2014.
21. General Acyclic Graphs of Tasks in TBB. `https://software.intel.com/en-us/node/506110`. Accessed: 2017-05-02.
22. M. Hoemmen and M.A. Heroux. Fault-tolerant iterative methods via selective reliability. In *Proceedings of SC11. IEEE Computer Society*, page 9, 2011.
23. L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOSPLA93*, pages 91–108. ACM, 1993.
24. Th. Willhalm and N. Popovici. Putting intel threading building blocks to work. In *IWMSE08*, pages 3–4. ACM, 2008.