Oral Comprehensive Exam

Position Paper

Performance Analysis of Many-Task Runtimes

Nicholas A. Chaimov
nchaimov@cs.uoregon.edu
University of Oregon

January 27, 2016

1 Introduction

Parallel programming is difficult. Switching from sequential to parallel programming introduces entire new classes of errors for the programmer to make, such as deadlock and race conditions, which are difficult to debug and complicate testing and correctness proofs [80]. Yet there are entire classes of programs with computational demands so great that sequential solutions are infeasible. We do parallel programming because we care about performance.

How do we know if we are getting good performance? We must observe the execution of our programs to determine if they are making good use of the resources available to them. Once we have made observations, how can we use those observations to improve performance? We can use autotuning to identify variants and parameters that give better performance than others, but this process itself is slow, so we can attempt to synthesize performance data into empirical models to guide the process. Alternately, we may ask not simply for better performance, but for an *explanation* of performance: automated performance diagnosis. These techniques are all well-developed for the current high-performance computing environment, but the advent of exascale computers will be a disruptive change which will require new techniques for performance monitoring and analysis.

In this paper, I first introduce the models of parallel programming which are currently in wide use. I then discuss how existing systems collect performance information. I then discuss automated systems which make use of performance data once it has been collected: autotuning systems, which measure the performance of many implementations of a code to identify good-performing variants; automated modeling systems, which construct models from performance data by which the performance of code can be predicted without running it; and performance diagnosis systems, which reason about performance data to arrive at hypotheses about the causes of bad performance. I then discuss the challenges of the coming exascale era of high-performance computing, and discuss new parallel programming models which are emerging to meet those challenges. Finally, I discuss problems with

existing systems for collecting and making use of performance data in the exascale era and describe the features necessary for future such systems.

2 Current Programming Models

All current supercomputers consist of many nodes, each of which contain many individual processors, which are often supplemented by accelerator devices such as GPUs; the two fastest such systems, Tianhe-2 and Titan, contain 3.12 million spread across 16,000 nodes and 560,640 cores spread across 18,688 nodes, respectively, with the former equipped with Intel Xeon Phi accelerators and the latter with NVIDIA K20 GPUs [115]. Thus we need ways of exploiting available parallelism both on-node and between nodes. By far the most popular solutions for this are OpenMP and MPI [87].

2.1 Shared Memory: OpenMP

OpenMP [91] is the most common method of exploiting on-node parallelism. It uses the *fork-join* model: programs begin executing sequentially, eventually *fork* into multiple threads of execution which operate in parallel before *joining* back into a single, sequential thread of execution (Figure 1). It uses a *shared memory* model: all threads of execution within a program share the same address space and access the same memory. It is *directive-based*: parallelism is expressed by taking what would otherwise be an ordinary, purely sequential program and annotating it with directives indicating which parts of the program should be executed in parallel and how access to shared memory should be managed. Thus the code

```
do_work();
```

can be made to run multiple times in parallel by adding an annotation

```
#pragma omp parallel
do_work();
```

causing multiple threads to be spawned, each of which execute the function `do_work` before joining, with the main flow of program execution continuing sequentially once all the threads have finished `do_work`.

When running several instances of `do_work` in parallel, it may happen that the separate instances attempt to use the same memory. OpenMP provides several annotations for controlling access to memory: `#pragma omp critical` marks sections of code which only one thread should be allowed to execute at a time. `#pragma omp single` marks sections which only one thread should execute *at all*, while `#pragma omp master` marks sections which a specific thread – the one which existed when the program started and will continue to exist after leaving the parallel region – should execute. `shared` and `private` clauses indicate whether threads should share one copy of a variable or should each operate on a local copy, while `reduction` clauses specify how per-thread local variables should be reduced to a single value which persists in the master thread after the end of a parallel region.

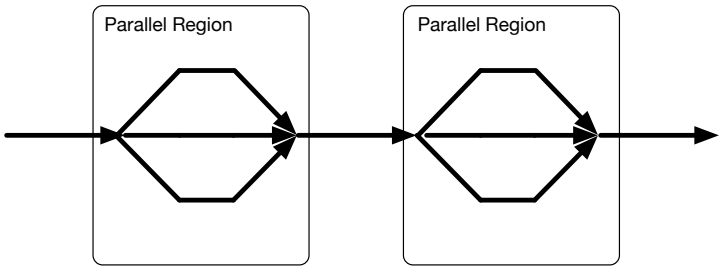


Figure 1: The Fork-Join model as used in OpenMP. There is ordinarily one thread of execution, which *forks* to become multiple threads in parallel regions. When exiting a parallel region, the threads *join* back into a single thread of execution.

In the above example, every thread executes the exact same code, which is almost certainly not what we want – different threads should process different data. Threads can be distinguished by a thread number which can be retrieved with a call to `omp_get_thread_num`, so that threads can identify which data they should be processing, but the more common usage is to use work-sharing constructs which automatically distribute work to threads, so that if we have a loop

```
for(int i = 0; i < 1000; ++i) {
  do_work(x[i]);
}
```

we can add a directive

```
#pragma omp parallel for shared(x)
for(int i = 0; i < 1000; ++i) {
  do_work(x[i]);
}
```

which causes the iterations of the loop to be automatically distributed across the threads. Several clauses are provided which allow the programmer to customize this distribution.

2.2 Distributed Memory: MPI

MPI [35] is the most common method of exploiting between-node parallelism. It uses the *communicating sequential processes* model: multiple instances of a program begin executing simultaneously, but each instance executes sequentially. These processes coordinate by sending messages to one another (Figure 2). It uses a *distributed memory* model: every process has its own address space, so every process has its own copy of each variable and changing a variable in one process does not change the value in any other process. A process may change the state of another process only by sending it a message. It provides a low-level API: unlike OpenMP, which provides directives which modify execution of an otherwise sequential program, MPI programs contain explicit API calls which carry out communication.

MPI processes all execute the same code. Processes can distinguish themselves by calling `MPI_Comm_rank` to obtain their *rank*. Unlike OpenMP, this is the only way that processes can determine that they should process different data: there is no equivalent to OpenMP's loop constructs, so the programmer is responsible for explicitly partitioning work.

Messages can be point-to-point or collective. Point-to-point messages are sent by a process through a call to `MPI_Send`, whose arguments specify the source, size, type and tag of the data to be sent. A corresponding `MPI_Recv` call must be executed on the destination to receive the message. Both calls are blocking; neither the sender nor the receiver will continue executing until the communication has completed. This limits the possibility for overlapping communication and computation and creates the potential for deadlock when communication is cyclic, so nonblocking `MPI_ISend` and `MPI_IRecv` versions are also provided. A set of collectives are also provided for efficient communication between multiple ranks.

MPI also supports *one-sided communication*, in which data can be sent to (*put*) or retrieved from (*get*) without an explicit call on the remote rank, using Remote Direct Memory Access (RDMA). In this mode, memory must be pre-registered (`MPI_Win_create`) to make it a valid target of subsequent `MPI_Put` and `MPI_Get` calls. These calls are always nonblocking, and explicit synchronization (`MPI_Win_fence`) is required to ensure that the operations have completed before using the values sent or retrieved through one-sided communication.

2.3 Partitioned Global Address Space: UPC

A disadvantage to MPI is lack of orthogonality: local communication occurs through direct access to the local memory, using ordinary features built in to the language, while remote communication occurs through API calls. The *Partitioned Global Address Space* – or *PGAS* – approach uses a common syntax for local and remote communication [131]. The address space is global – every process can access memory in every process – but is also partitioned: every address is *owned* by a particular process, and a pointer consists not only of an address but also a tag indicating who the owner is. When a process reads or writes through a pointer to locally-owned memory, this is translated into a local memory address as normal. When a process reads or writes through a pointer to remotely-owned memory, this is translated into a message sent over the network which triggers a read or write of the

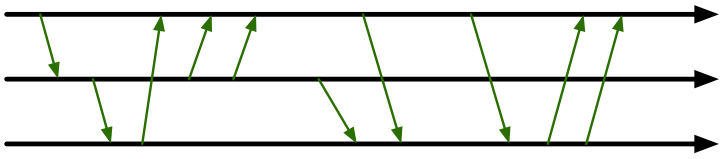


Figure 2: The Communicating Sequential Processes model as used in MPI. There are multiple threads of execution (black), each of which runs sequentially. They communicate with one another by sending messages (green).

address in its owning process and, in the case of a read, a reply message containing the value stored at the address.

Unified Parallel C [116] is a language which extends C99 [59] with support for *shared pointers* to a partitioned global address space and a variety of work-sharing constructs similar to those provided by OpenMP. As in MPI, multiple copies of the same executable are launched, and these execute the same code. In UPC, pointers and arrays can be declared `shared`, making them globally available. For example,

```
shared double a[3*THREADS];
```

declares an array `a` of doubles with three elements per thread (a UPC thread corresponds to an MPI rank) which is globally accessible. By default, ownership – or *affinity* – of memory in an array is assigned cyclically, so that the memory located at the address `a + i` is physically located on thread `i % THREADS`. Arrays can also be divided into blocks of elements which are distributed cyclically, or each thread can be assigned a contiguous block of the array.

Unlike MPI, UPC provides built-in support for partitioning work across threads. The `upc_forall` loop, when encountered by a thread, runs only those loop iterations which have affinity to the thread that encountered the loop. For example,

```
shared double x[N], y[N], z[N];
// initialize x and y
int main() {
  upc_forall(int i=0; i < N; ++i; i) {
      z[i] = x[i] + y[i];
    }
}
```

resembles an ordinary C `for` loop with the exception of an additional parameter to the loop. This parameter specifies the affinity, and the value of `i` here means that a thread encountering the loop will run all iterations for which `i % THREADS == MYTHREAD`. Like MPI, UPC provides synchronization primitives such as `UPC_barrier` and a variety of collective communication operations.

2.4 Accelerators

As noted above, the current generation of top supercomputers feature accelerators, as will the next generation of supercomputers which will be installed in 2017 and 2018. Accelerators generally feature a larger number of cores than general purpose CPUs, but each core is less capable than those in a CPU.

2.4.1 CUDA

NVIDIA GPUs are available with up to 4,096 cores, but these cores do not have all features typical of a CPU core: notably, cores are not capable of independently fetching and scheduling instructions. Rather, a group of cores share fetch and schedule hardware and always execute identical instructions during the same clock cycle, differing only in the memory addresses read and written by those instructions. Figure 3 shows the NVIDIA architecture: all of the cores share L2 cache and access to

the memory and PCIe buses, while sets of 32 cores share L1 cache, fetch and dispatch units, registers, load-store units and Special Function Units, while each core has its own floating point and integer arithmetic units. AMD GPUs (Figure 4) use a similar architecture.

To allow programming NVIDIA GPUs, NVIDA developed CUDA [89], C language extensions and APIs for writing code which will execute on a GPU and for transferring data between host and GPU memories. CUDA kernels are C functions which are annotated `__global__`, indicating that they will run on a GPU but can be invoked from the host. A kernel function differs from an ordinary function in that many copies of the function will execute simultaneously. A given instance of the function must examine its local copies of the `blockIdx` and `threadIdx` variables to determine which portions of the input data it should process.

CUDA maintains separate memory spaces for the host and each device. Running a kernel on a device then involves the host explicitly allocating memory on the device (`cudaMalloc`), copying input data to the device (`cudaMemcpy`), specifying how the input data is to be partitioned into blocks, launching the kernel, and copying output data back onto the host.

2.4.2 Xeon Phi

The Intel Xeon Phi accelerator architecture features fewer cores than are found in GPUs (61 cores and 244 hardware threads) which are considerably more complex than GPU cores but which are still simpler than the cores typically found in a host processor [97]. The cores are connected together by a bidirectional ring bus, which they share with a distributed, globally coherent L2 cache (Figure 5). Each core features four hardware threads, can dispatch two instructions per cycle, and is required to switch between hardware threads once per cycle, which results in the

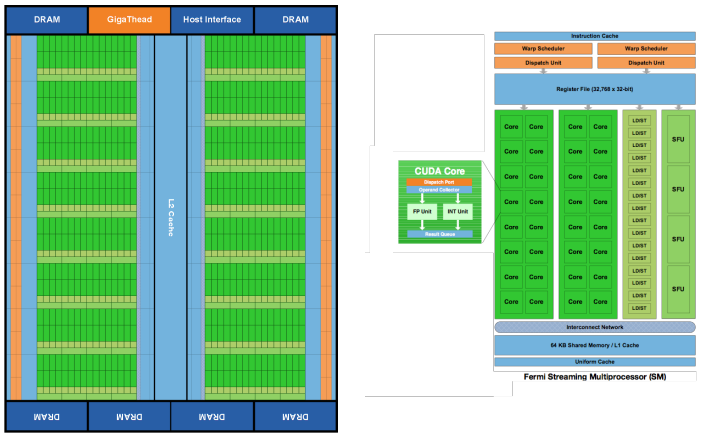


Figure 3: The architecture of the NVIDIA Fermi GPU family.

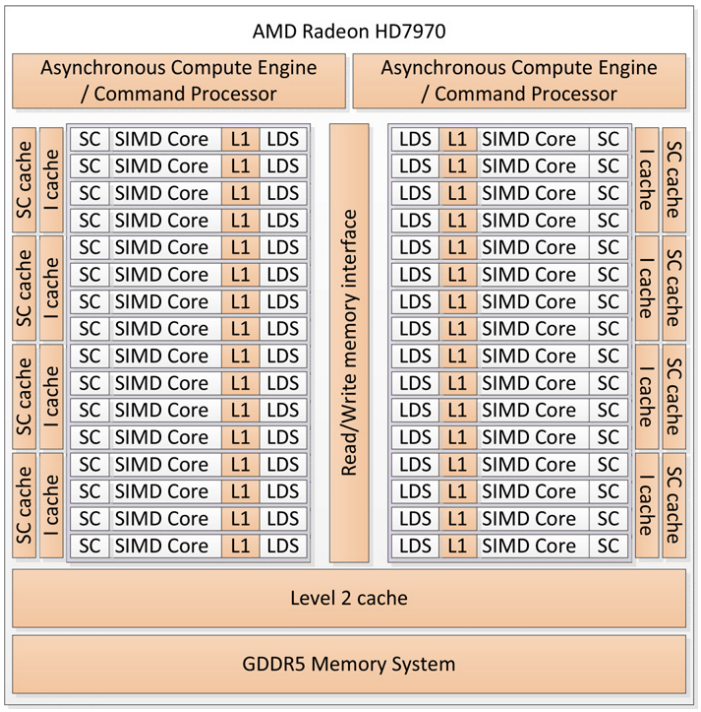


Figure 4: The architecture of the AMD Radeon 7000 GPU family.

requirement that enough work be available that instructions can actually be issued every cycle – if an insufficient number of threads are used, the issuing hardware will remain idle every other cycle. The cores use in-order execution but feature SIMD units with twice the width of current x86-64 chips.

Xeon Phi accelerators themselves run Linux and can be programmed through several mechanisms, including a native mode using traditional MPI and/or OpenMP, as well as an offload mode [88] which uses `pragma` annotations and/or language keywords to specify work which should be executed on the accelerator, from which the compiler will automatically synthesize the necessary memory copy and kernel launch code.

2.4.3 Cross-architecture Programming Models

There are several projects aimed at providing programming models which allow a single code to target multiple types of accelerators. OpenCL [107], an industry standard maintained by the Khronos group, is one such model. Its structure and syntax are similar to those of CUDA, but with additional abstractions for devices, compute units, processing elements, and private, local and global memories which a driver for a device maps onto physical hardware. Drivers are available for many de-

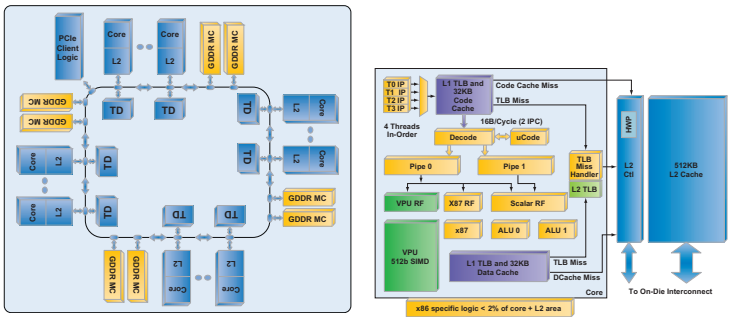


Figure 5: The architecture of the Intel Xeon Phi *Knights Corner* family. Images provided by Intel.

vices, including NVIDIA and AMD GPUs, Intel Xeon Phis, Intel and AMD CPUs, as well as FPGAs [30, 25]. Since the target hardware is not necessarily known at compile time, kernel code is stored as a string and is provided to the device driver for compilation just prior to kernel invocation.

In addition to low-level approaches to portability, higher-level approaches also exist. OpenACC [47] is a `pragma`-based model for device programming, similar to OpenMP, in which loops are annotated to indicate that their iterations should execute in parallel on accelerator devices. As accelerator devices have separate memory spaces from the host, additional `data` directives are added to specify data to be copied and allocated on the device. OpenMP itself is also being extended with device support through `target` directives [74].

3 Capturing Performance Data

Once we have a parallel program – most likely written using one of the programming models discussed in Section 2 – how can we determine whether it performs well? To do this, we must first determine when events occur during execution of the program by means of *instrumentation* [85].

To instrument code, we cause additional instructions to be executed which record events and facts about those events, such as the time or number of cycles elapsed between two events. There are several ways to accomplish this. We can use *source code instrumentation*, where we modify the source code, inserting function calls at the beginning and end of functions or around loops. In order for facts about events to be useful, we must be able to map events back onto the source code so that we know where changes should be made to address any performance problems found. Directly modifying the source code allows us to most easily map events back onto source, since each event generated by inserted code can be given a unique name. However, source code instrumentation requires that we have the original source available, and that we are able to parse the source code in order to modify

it. Source code instrumentation is supported by systems such as TAU [103] and VampirTrace [68].

Alternately, we can modify the binary, either through rewriting prior to execution or dynamically at runtime, through libraries such as Dyninst [96] as used by TAU, or through performance analysis tools which directly implement binary analysis, such as HPCToolkit [2]. Such systems analyze the binary, identifying entry and exit points for functions and inserting calls to log events. This type of approach makes it straightforward to dynamically adjust instrumentation points at runtime through self-modifying code, allows instrumentation of binaries for which the original source is not available or which is written in a language for which automated source instrumentation tools are not available, and eliminates overhead for runs in which instrumentation is not desired (in which case the binary is run unmodified). However, it is more difficult to map events back onto the source code, as compiler optimizations applied in creating the binary may disrupt the relation between instructions and the source line which caused them to be emitted.

For systems such as MPI, OpenMP, and UPC which feature runtime libraries, instrumentation can be performed at the runtime level rather than the application level. This can be accomplished by preloading a library which exposes the same interface as the actual runtime which logs events before forwarding function calls to the actual runtime. Such interposed libraries include mpiP for MPI [117] and ompP for OpenMP [37]. Runtimes can also expose callback interfaces through which a performance monitoring tool can register functions which will be called by the runtime when certain events occur, such as OMPT for OpenMP [32], CUPTI for CUDA [90, 76], and the OpenCL event profiling interface [58] and GPUPerfAPI [3] for OpenCL.

Finally, we can use *sampling*, where we request that an interrupt be called periodically or when a hardware performance counter reaches a certain value or overflows. The interrupt transfers control to the performance monitoring tool, which can record the address which was being executed prior to the interrupt. Sampling allows fine control over the tradeoff between overhead and error: by increasing the sampling rate, we get a more precise picture of what the program is doing and are less likely to miss events which occur infrequently, while at the same time we increase the proportion of time spent running the monitoring routines instead of program code. By decreasing the sampling rate, we reduce overhead at the cost of increased likelihood of missing infrequent events.

Any of these techniques – source-level instrumentation, binary instrumentation, library interposition, runtime instrumentation and sampling – can be used to generate events. When events are generated, what should the performance monitoring system do with them? Generally, they will be used to generate either a profile or a trace [85]. In *profiling*, events mapped to a particular code region are used to create an aggregate measure of performance for that code region [44]. If function-level instrumentation is used, then, the profile might record the number of calls to the function and the time spent in that function aggregated across all calls to it. There are different choices to be made as to the level at which aggregation occurs. For example, in *call-path profiling* [50], the performance monitoring system stores separate profiles for a function depending on the call path through which the function

was reached, so that if `A()` calls `Z()` and `B()` calls `Z()` we would see two separate profiles for `Z()`. This can help account for input-dependent behavior, as different uses of a function may use different data and thus exhibit different performance characteristics. In *phase-based profiling* [75], separate profiles are stored for *phases* of an application, such as particular algorithms or iterations of iterative algorithms.

In *tracing*, events are simply separately recorded along with a timestamp [44]. In a distributed system, traces collected on separate nodes must be merged so as to maintain ordering on systems which do not have synchronized clocks. Traces provide a large amount of information with which to diagnose performance problems and allow phases of program execution to be automatically discovered: given the full list of events, we can infer causality between events. However, the volume of data generated can be exceptionally large, particularly for runs using large portions of a supercomputer. Traces grow both with the number of processes used (more processes each generating events) and with the runtime of the application.

4 Autotuning

Once we have a mechanism to acquire performance data, how can we use that data to improve performance? One approach is *automatic performance tuning*, or *autotuning* [9]. Autotuning arises from the idea that the best-performing implementation of some code is not the same everywhere: it depends on the architecture of the processors on which the code will execute, the operating system, networking infrastructure, and other system parameters [125], on properties of the input data such as size [105] or the number and distribution of nonzero elements in a sparse matrix [104], and on the interaction between system parameters and input data. Many runtimes, such as MPI and OpenMP, also have parameters which can be set to control scheduling of work or use of network resources [18]. In an autotuning system, we generate code variants and/or modify runtime parameters and perform instrumented runs, which we use to determine which variants and parameters result in the best performance. The space of possible variants and parameters is very large for all but trivial problems, so heuristic search algorithms are used to avoid exhaustive enumeration and testing of the entire space.

Basu et al. [9] identify three categories of autotuning systems: *self-tuning libraries*, in which autotuning support is built directly in to a library and is run at install time or runtime; *programmer-directed autotuning*, in which the programmer of a piece of software exposes runtime parameters to a search system; and *compiler-directed autotuning*, in which a library of code variants are generated by a compiler or source-to-source translator. They envision a system in which all of these techniques can be combined through the use of a centralized search engine and performance database (Figure 6).

4.1 ATLAS

One approach to autotuning is to build autotuning support directly into a library. An early and widely-used such library is the linear algebra library ATLAS [122] (Automatically Tuned Linear Algebra Software). Traditionally, hardware, operating

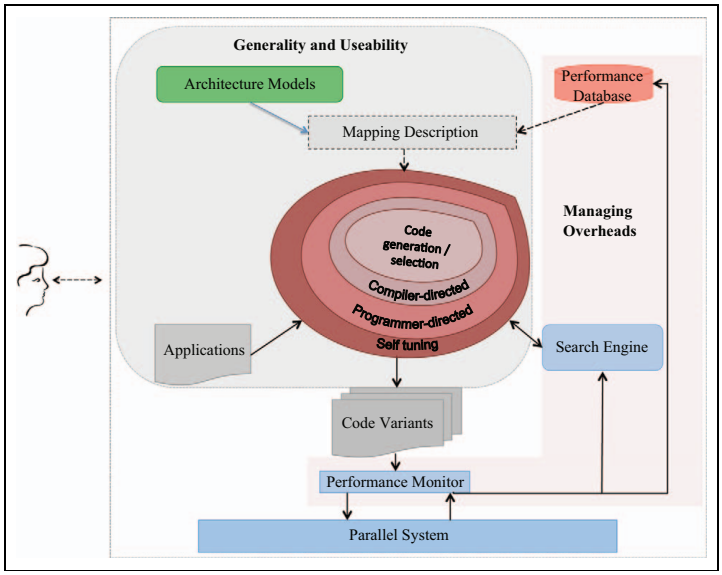


Figure 6: A hypothetical architecture for a unified autotuning system, in which multiple types of autotuning are present in a single application and share a search engine and performance database. From [9]

system and compiler vendors have generated hand-tuned linear algebra routines for developers using their products. ATLAS represents a different approach, shipping a variety of parameterized function implementations which are tested during compilation. The developers of ATLAS identify four requirements for the application of empirical optimization [121]:

- Isolation of performance-critical routines.
- A method of adapting software to differing environments.
- Robust, context-sensitive timers.
- Appropriate search heuristics.

ATLAS performs its tuning at compile time. This is beneficial in that it does not introduce any delays at runtime due to the need to select an implementation at that time, but this also limits the ability of ATLAS to adapt to a changing execution environment or to the input data, which is only known at runtime (for example, to adapt to different sizes of input matrices, if a given program tends to use matrices of one of a few fixed sizes.)

4.2 FFTW

Another approach is that used in FFTW3 [36], a Fast Fourier Transform library. In FFTW, the user of the library invokes the library with a description of the problem to be solved (e.g., which discrete transform is to be calculated) and the sizes and memory layouts of the input arrays. FFTW includes code, called the *planner*, which will then test many different functions for calculating the desired transform on problems of the indicated size and layout, and select and return the best-performing one.

This technique allows FFTW to adapt to changes to its execution environment (such as in the case of migration) and to properties of the input data. However, if only a small number of transforms of a particular type and for particular input types are performed, then the cost of performing the tests will outweigh the increased performance from using tuned variants, and overall program execution time will be slower.

4.3 SPIRAL

Spiral [94] is a general-purpose digital signal processing library in which DSP algorithms are expressed in a domain specific language, SPL, which is ultimately translated into C or Fortran. Optimizations can be applied at both the DSL and target language levels and can take into consideration properties of the domain that enable optimizations that are not generally applicable to all domains. Some optimizations use a static cost model to determine whether they should be applied, while others use search algorithms to explore the space of optimizations, for which exhaustive and random search, dynamic programming, evolutionary algorithms and hill-climbing search algorithms are provided.

The evolutionary algorithm mode is particularly interesting: genes are represented as *ruletrees*, which specify the recursive structure of a transform with leaf nodes representing particular implementations. Mutations are made by swapping an implementation for another, while cross-breeding occurs by swapping subtrees. Additionally, SPIRAL uses empirically-generated models by timing subtrees within a ruletree.

4.3.1 OSKI

Oski [118] is a sparse linear algebra kernel library which uses a similar approach to FFTW, performing tuning at runtime based upon known input parameters. The library provides a set of functions for specifying *hints* about input sizes, coefficient values, data formats, and the number of times different operations are expected to be performed. The tuning process can then generate specialized variants, and, because the estimated frequency of operations is provided, OSKI can determine how much time should be spent on tuning particular operations based on whether it is likely to be executed enough times to amortize the cost of tuning.

pOSKI [16], a system for generating optimized sparse matrix-vector multiplication routines, combines offline autotuning with model-driven online autotuning combined with a history database. The offline tuning, which happens when the library is initially installed, tests combinations of storage format (CSR or BCSR),

size of register blocks, prefetching policy, and SIMDization policy for a set of likely block sizes. At runtime, when the actual matrix is available and its sparsity therefore known, a simple model is used to select a block size, and therefore an implementation from among the pre-generated set of optimized implementations.

4.3.2 Orio

Orio [51] is an autotuning system providing pluggable code generators and search algorithms and using an annotation-based approach to specifying code transformations. Input code in a language such as C or Fortran is annotated with special comments indicating that the annotated code should be replaced with code generated by Orio according to specified transformation. A loop, then, can be annotated with a Loop transformation specifying that a version of the loop written in a restricted subset of C or a domain-specific language should be unrolled by some factor and tiled by some factor.

These annotations can be left with parameters (such as tile factor) unspecified and be wrapped in a *tuning specification*, which specifies the range of values valid for each parameter, what search algorithm should be used, and how the kernel can be tested in isolation: how input data can be generated, and the sizes of input data which should be tested. Each such tuning specification then describes a set of experiments, the output of which are tuned variants which are inserted into the source code, replacing the original implementation. As the tuning specifications and annotations are comments, the original source code can also be compiled unmodified to give the original implementation. Transformations are also available to generate CUDA [77] and OpenCL [19] code for use on accelerators.

4.3.3 CHiLL + Active Harmony

Active Harmony [114] is a general purpose search engine capable of rapidly exploring the parameter search space by testing multiple hypotheses in parallel, using the Parallel Rank Ordering algorithm to evaluate potential parameters, which is used both for online tuning of application and runtime parameters and for offline tuning by providing parameters to an external code generator. The user can specify parameters, ranges for the parameters, and constraints restricting the values parameters can take on. Active Harmony runs using a client-server architecture, in which a centralized Harmony server communicates with, and provides parameters to, multiple clients running on different nodes in a cluster. Using Parallel Rank Ordering, the system can provide different parameter values to different nodes in the cluster, allowing it to evaluate the search space in parallel. When used with a code generator, code servers can also be configured, which perform compilation of code variants and distribute compiled object files to the execution nodes [52].

Active Harmony has been used with CHiLL [23], a code variant generator which uses a "recipe" of high-level loop transformations which are applied together to generate variants of a loop. CHiLL uses the ROSE compiler [95] internally to parse code and applies transformations by making modifications to the ROSE AST. It uses a polyhedral model of loop transformations, in which the order of opera-

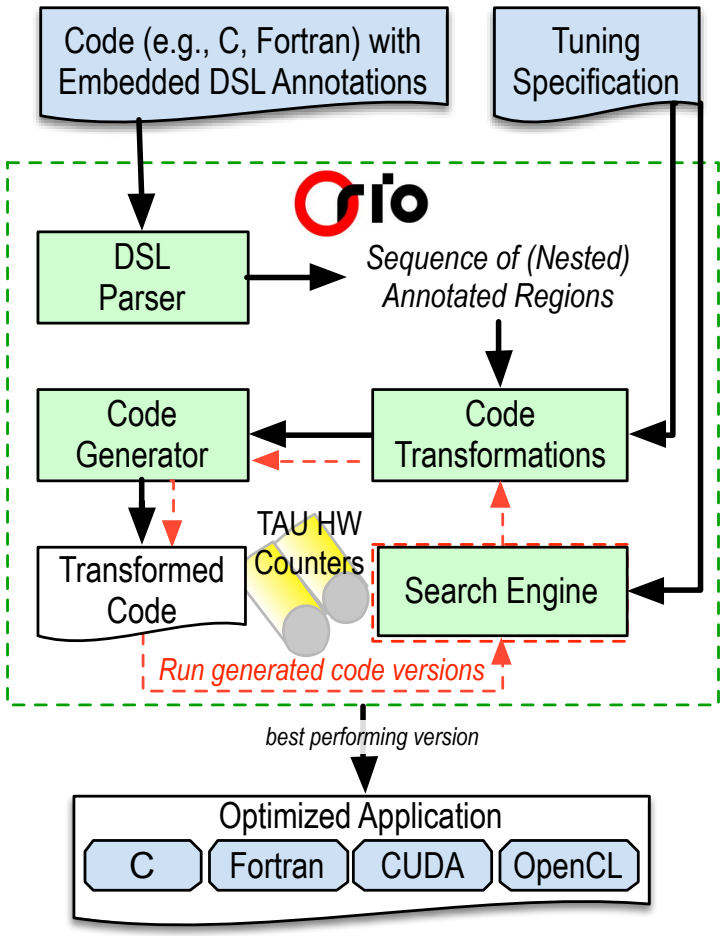


Figure 7: The architecture of Orio. From [19].

tions within nested loops are viewed as points inside a polyhedron, from which semantically-equivalent loops evaluating nests in different orders can be generated by applying geometric transformations to the polyhedron representing loop iterations [46]. CHiLL recipes can be parameterized, and autotuning can be performed by searching the space of parameters to available recipes. Transformations are available for generation of CUDA code through CUDA-CHiLL [100]. The combination of CHiLL and Active Harmony has also been used with the ROSE outliner, a system which extracts regions of code within a function into independent functions which can be separately tuned [113].

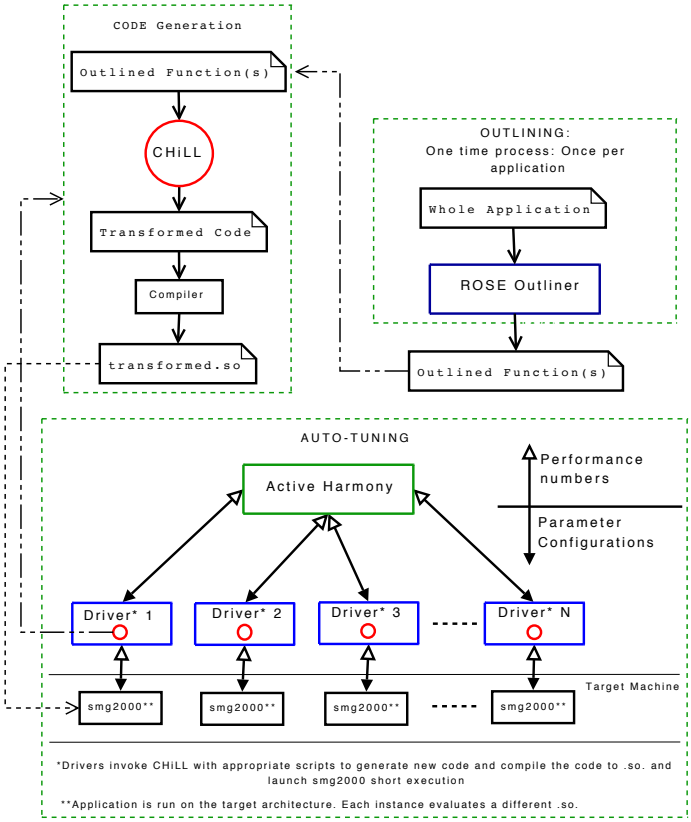


Figure 8: The architecture of an autotuning system using ROSE to outline functions, CHiLL to generate code variants, and Active Harmony to direct the search process. From [113].

4.4 Periscope

The AutoTune project [83] is developing the Periscope Tuning Framework, an extension to the earlier Periscope [11] performance analysis and diagnosis tool, described in more detail in Section 6.1. The architecture of PTF is shown in Figure 9. In PTF, tuning plugins are registered which interact with a set of *scenario pools*. Plugins can insert new scenarios into the *created scenario pool*; can pull created scenarios, process them, and insert the result into the *prepared scenario pool*; can create experiments from prepared scenarios, inserting them into the *experiment scenario pool*; and, once the execution engine has run an experiment from that pool and made it available in the *finished scenario pool*, can pull the results and process them to create a human-readable report.

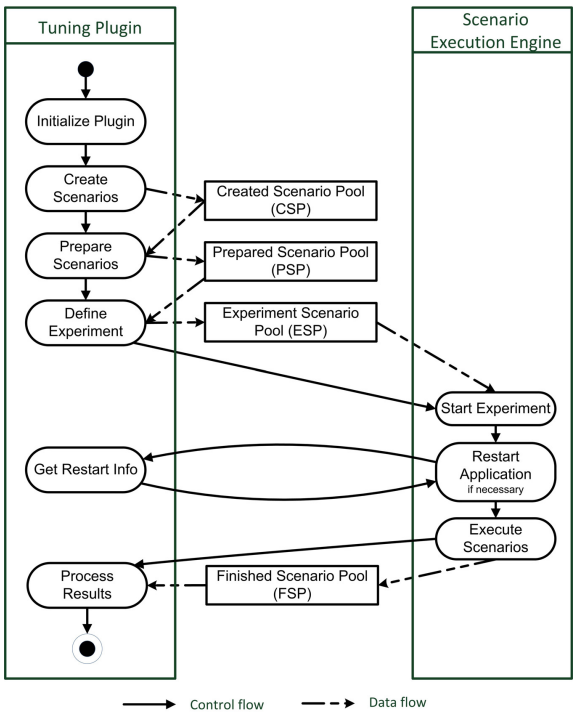


Figure 9: Architecture of the Periscope Tuning Framework. From [83].

4.5 Insieme: Multi-Objective Optimization

The Insieme framework [49], unlike most auto-tuning frameworks, is designed specifically for *multi-objective optimization*, which allows for objectives such as "minimize execution time used subject to constraints on the number of cores and the amount of energy used". When multiple objectives are present, the solution found is not a single best-performing configuration but rather a Pareto frontier, a set of points for which no objective can be improved without degrading some other objective. The best configuration given some particular set of tradeoffs is then always found on the Pareto frontier. Genetic algorithms map well onto the problem of finding the Pareto set [31], particularly differential evolution techniques in which the rate of evolution for different parameters itself evolves.

4.6 Collective Tuning

An alternate approach is used by Fursin et al. in their *Collective Mind* project [38]. Rather than enforcing a strict schema, they allow the user to encode *measured characteristics*, *choices*, *features* and *system state* in JSON format [14], which can be used

without requiring that a schema be provided. When in the course of a project a schema becomes necessary, it can be provided, also in JSON format. The user can provide modules which mediate between Collective Mind data and external tool. These modules are gradually composed into a workflow which specifies the overall experiments to be done. Collective Mind encompasses the earlier *Crowdtuning* [81] and *Collective Tuning* [39] projects, which made available a more restrictive central repository for performance data from the MILEPOST GCC compiler. The compiler generates a library of compiled versions of functions with different optimizations applied. At runtime, when a function of interest is executed, either the currently known best version or a different, proposed version is randomly selected and profiled, with the timings being sent to the central repository.

4.7 Online Adaptation

The Abstract Data and Communication Library [40] (ADCL) is used for runtime tuning MPI applications. A library of variant implementations of a communication routine, called a *function set*, is defined either by the library designer or the developer of the application. ADCL then uses either brute-force search or parameter-at-a-time search to evaluate the variants. In one case study [41], it was used to select from a set of neighborhood communication routines (in which each rank communicates with six neighbors in each iteration), which varied along three axes: the number of simultaneous communication partners (e.g., pair-at-a-time or all-to-all), mechanism for handling messages with contents not contiguous in memory (e.g., by packing the data into a contiguous array before communicating, or by defining a custom data type), and the underlying data transfer routines used (e.g, blocking vs non-blocking communication, two sided vs. one sided communication, etc.). Different variants were selected for different architectures, network hardware, and problem sizes. Interestingly, the best-performing variant for some configurations was the worst-performing variant for another, demonstrating the importance of autotuning in this case. The library includes pre-defined function sets for the standard MPI collectives [12].

A later version of ADCL adds the ability to focus the search process using data from previous runs [33]. The authors identify two primary obstacles to the use of historical data: that the system may not have stored performance results for the particular execution environment and problem now being encountered, and that changing conditions (such as degree of congestion on the network, or the physical location of ranks as assigned by a batch scheduler) mean that even if the system is encountering a problem which has been encountered before, the best performing variant as determined in the past may not be the best performing variant now. To work around these problems, ADCL uses a distance metric to select good-performing variants from history which are, according to that metric, most similar to the variant now being encountered, and requires that performance be within a user-specifiable tolerance of that recorded in the history file. If not, search is repeated.

The Open Tool for Parameter Optimization [18] (OTPO) uses search algorithms from ADCL to tune parameters exposed by the OpenMPI runtime. In OpenMPI, many runtime tasks are delegated to modules, which implement different ver-

sions of communication algorithms (such as collectives) and map MPI operations onto lower-level network operations (such as for TCP, InfiniBand, Cray Gemini/Aries, etc.). These modules expose a set of tunable parameters, called MCA parameters, of which a typical installation will have several hundred. Using search algorithms from ADCL, OTPO searches for parameters giving the best performance, as measured by latency or bandwidth of network operations.

OTPO finds good MCA parameter values, but requires a large number of evaluations to do so. To reduce the number of evaluations needed, Pellegrini et al. [92] evaluate the effect of different parameters on performance at compile-time and use this data at runtime to tune only those parameters most likely to have large performance effects. During installation of OpenMPI, a set of kernels, chosen to approximate the communication patterns of typical applications, are run with randomly-chosen parameter values. ANOVA is then used to identify which parameters have the greatest impact on performance.

5 Performance Modeling

We can also use performance data to attempt to construct *empirical models* which allow us to predict performance of the code on other systems or datasets. Such models can then be used to guide autotuning or performance diagnosis.

Prophesy [111, 110, 129] is an integrated system for automatically generating analytical performance models, comprising a source-code instrumentation component [129], a database component [110], and a model builder component [111]. Performance data is collected at the basic block level and stored in the performance database as a hierarchy, in which applications are made of modules, modules are made of functions, and functions are made of basic blocks, allowing for measurements to be viewed at an appropriate level of abstraction for the current task. The database stores information on applications (name, version, etc.), executables for applications (how it is compiled, what libraries it uses, and static analysis results such as control flow data), run information for particular runs of applications (machine and input information), and hierarchical performance measurements.

Prophesy then implements three modeling techniques: curve fitting, parameterization, and composition. Curve fitting is fully automated, while parameterization and composition require additional input from the user. Curve fitting attempts to model the performance of the application or functions of the application in terms of input parameters (such as size), but does not incorporate system-specific features and therefore can only be used to evaluate intra-system scalability and not to predict performance across systems. Parameterization incorporates coefficients representing system-specific parameters, but requires manual annotation of kernels to identify and count operations. Composition combines models stored in the database to allow application performance to be represented as the composition of models for the application's constituent kernels. Pairs of kernels are evaluated to determine the ef-

fect of running one kernel after another[1], resulting in an *coupling coefficient* C_{ij}, the effect on the performance of kernel j when it runs after kernel i. C_{ij} equals 1 when there is no interaction, is less than 1 when performance of j improves (such as when running i has resulted in data used by j being loaded into the cache), and is greater than 1 when performance of j is degraded.

An early comparison of empirical autotuning with model-based parameter selection was performed by Yotov et al. [132]. Looking specifically at matrix-matrix multiplication codes as generated by ATLAS, described in Section 4.1, they develop a simplified model of how cache behavior is affected by parameters to the matrix multiplication code generator and substitute the search module of ATLAS with the model. On two systems (SGI and Intel) their model yields performance within 1% of that produced by the full ATLAS search, but reduces installation time to 35% of its original value. On a third system (Sun) the model-predicted variant has 20% worse performance than the empirically-determined version. This demonstrates the promise of model-driven approaches, but also its limitations: much effort went into developing the models, which are specific to only one ATLAS routine.

Modeling can also be used in combination with autotuning, rather than strictly as a replacement. One of the major uses of modeling in combination with empirical autotuning is to avoid evaluating variants which a model predicts will have poor performance, thereby focusing the search on variants expected to perform well. Balaprakash et al. [8] used an active learning [102] technique customized for autotuning on HPC systems. They observed that a major problem with existing parallel active learning techniques was that when such an algorithm suggests multiple points to evaluate, the result for one such point can dramatically reduce the information gained from evaluating the remaining points in the proposed set, resulting in wasted effort evaluating code variants corresponding to such points. They modify the algorithm to attempt to avoid suggesting such points by 1) selecting an initial point x_i, 2) retraining the classifier assuming that the prediction for x_i was correct, and 3) selecting another point only from among those points whose informativeness was not substantially reduced by retraining.

Sarangkar and Qasem [101] describe MATS (**M**odel-driven **A**daptive **T**uning **S**ystem), an autotuning system which uses simple architectural models to constrain the set of transformation parameters to consider. Based on static code analysis to calculate reuse distance and models of effective data and instruction cache capacity, register set, and TLB size, parameter values for loop tiling, fusion, fission, interchange, and unroll are selected so as not to violate a user-specified tolerance value, which express, for example, that number of cache misses in considered variants should be no more than some percentage worse than the optimal value.

GROPHECY [82] predicts whether a CPU code is amenable to implementation on GPUs using an analytical model to determine whether the code is compute-bound or memory-bound. To do this, the user must first manually convert the CPU code into a *code skeleton* which lists only loops, memory loads and stores, and

[1]This is the formulation in the paper, although the same concept could also be used for two kernels running simultaneously, such as in a task based system. Scaling the technique to many simultaneous kernels may present problems, however.

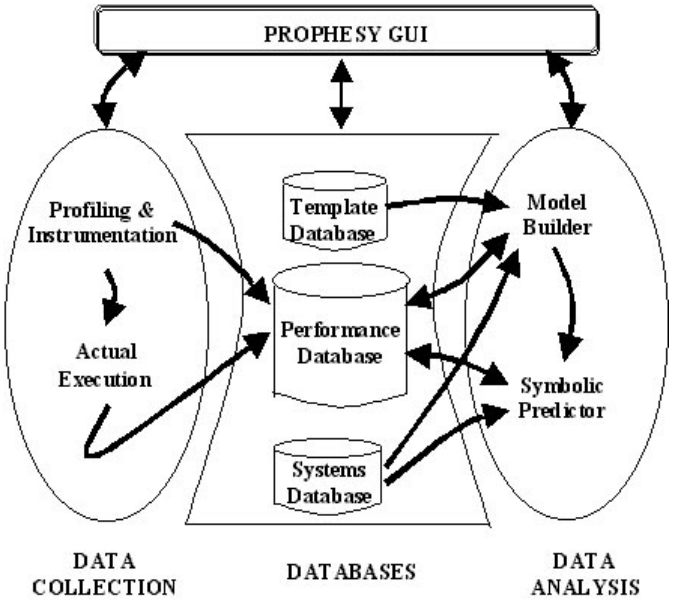


Figure 10: Architecture of Prophesy, from [111].

generic compute instructions. The skeleton is then converted into a set of possible GPU skeletons parameterized by many of the same parameters used in GPU code generation by autotuning frameworks. Instead of generating and running code, the model is used to estimate memory usage patterns.

Models need not attempt to determine the absolute performance of a code. In autotuning and runtime adaptivity, determining the expected performance of one code relative to another is useful. Models need not be based on performance at all. For example, Tang et al. [109] develop an empirical model of *contentiousness* and *sensitivity* when jobs are co-scheduled on a system and thus share resources. *Contentiousness* is the capacity of a program to degrade the performance of programs with which it is co-scheduled, while *sensitivity* is the propensity of a program to have its performance degraded when co-scheduled with a program of high contentiousness. These properties are distinct because contentiousness results from mere *use* of a shared resource, while sensitivity depends on a program *benefiting* from its use of shared resources. A program which reads large amounts of data from memory, processes it once, and never reuses data will make use of the caches, but will not gain a performance benefit from cache use due to lack of reuse. Such a program is nonetheless contentious. A program which reads a small amount of data and processes it repeatedly benefits greatly from cache use, and is therefore highly sensitive to other programs' use of the cache, whether or not those other programs benefit themselves from using the cache. The authors identify hardware performance coun-

ters (L2 and L3 cache lines input rate) and use regression to construct architecture and application-specific models which give relative contentiousness and sensitivity of applications. A scheduler can then use these to schedule high-contentiousness applications only with low-sensitivity applications.

Brainy [61] constructs architecture- and input-sensitive models for selecting the best C++ STL data structure for a given workload. For each architecture, a set of input programs are generated, instrumented, and tested, with each input program parameterized by the number of calls to each STL container interface function (e.g, *i* insertions, *j* finds, *k* in-order traversals, etc.) This allows the training set to include entries representative of a wide range of use cases. Timing and hardware performance counter data are collected for each call. These data are then used to train an artificial neural network which is used to predict the best-performing data structure for new applications based on the number of calls each makes to the various API functions. The architecture of Brainy is shown in Figure 11.

Rather than training a classifier based on program and system features, an alternative approach is to use clustering to identify programs with similar variation in performance across systems or systems with similar variation across programs. Such an approach was used by Cammarota et al. [17], who consider only program execution time, stored in a matrix M such that $M_{i,j}$ is the execution time of program i on system j. Having collected times for many programs on many systems, hierarchical clustering is used to group programs and systems according to similarity.

A major challenge with machine learning-based technique is in the selection of features. Leather et al. [73] automatically generate and test features using a genetic algorithm approach. A set of mathematical operators and functions operating on the compiler's intermediate representation are provided, together with a grammar describing how expressions using them can be formed. Every expression yields a real number. Genetic algorithms are then used to create new expressions from the existing population. Each proposed expression is tested by training a classifier using it as a feature and determining whether, and by how much, the addition of the feature improved the performance of the classifier. The degree of improvement is used as fitness. To evaluate this technique, the authors considered loop unrolling, exhaustively searching the space for a set of benchmark applications to determine the optimal value, and using the technique to create features, which performed better than human-selected features.

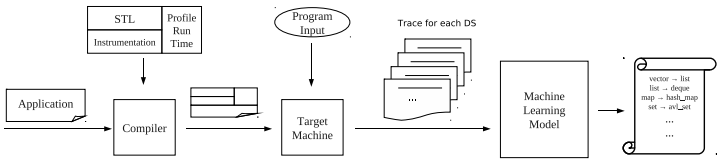


Figure 11: Architecture of the Brainy data structure selection system, from [61]

6 Performance Diagnosis

Autotuning, as described above, involves trying many variants or parameters, measuring their performance, and identifying variants and/or parameters that lead to good performance. Another approach to improving performance is *automatic performance diagnosis*, in which, rather than simply test a large number of variants, we analyze performance data from one run or a smaller set of runs and attempt to identify the specific *causes* of performance problems, so that we can develop targeted solutions to those problems.

6.1 Online Performance Diagnosis

Online performance diagnosis is the process of identifying performance problems *during* the run of a program. It is most useful for large-scale and/or long-running jobs in which collecting and making use of traces is not feasible.

Paradyn [84] is an early online performance diagnosis system designed to identify performance problems within a single run of a program, while minimizing the disruption it itself causes. It is based on a process of iterative search through a search engine called the Performance Consultant, which refines hypotheses, and on dynamic instrumentation: instrumentation is added at runtime when a hypothesis is being evaluated and, when the evaluation is done, the instrumentation is removed.

Search proceeds along three axes - "why", "where", and "when". Along the "why" axis, the system attempts to refine hypotheses; an example of a hypothesis hierarchy is shown in Figure 12. In that example, the system will first insert instrumentation to determine if a synchronization bottleneck is present. If not, it moves to a sibling hypothesis. If so, it will insert more specific instrumentation to test causes of the overall problem – are synchronization operations too frequent, or do synchronization operations take too long, *etc.*. Along the "where" axis, hypotheses are localized to resources, such as places in the program's code, nodes, particular synchronization objects, *etc.*. Search initially occurs at a high level in these hierarchies – such as, "does the *entire program* suffer from synchronization bottlenecks?" If so, the search is refined to locate parts of the program which suffer from synchronization bottlenecks and those which do not. Along the "when" axis, the system considers phases of execution, as performance problems may exist during some phases but not others.

Paradyn can use information from previous runs to focus future searches on the same code [66]. Inserting instrumentation for bottlenecks which are unlikely to exist unnecessarily perturbs performance, so hypotheses which have been disproved in many prior runs can be pruned from the hypothesis tree. Similarly, hypotheses which have proved true in many prior runs can be promoted so that they are searched earlier during program execution, allowing the most likely hypotheses to be tested even in short runs.

The original implementation of Paradyn is somewhat limited in scalability because the search process is centrally directed: one node is responsible for initiating instrumentation on all the nodes in the system, for processing measurements from all the nodes, evaluating hypotheses, and selecting new hypotheses to test. To increase

scalability, a Distributed Performance Consultant was developed [99]. Rather than one central search agent, each node runs its own agent which can communicate with other agents as necessary. In order to determine whether a hypothesis holds for the whole application, neighboring nodes communicate to determine whether a property holds for a local neighborhood. Graph clustering is used to identify neighborhoods with similarly properties, and these summarized data are propagated to other nodes, in order to eventually give an approximate representation of global behavior.

PERISCOPE [11] is an extensible performance diagnosis system based on a set of interacting agents. Its architecture is shown in Figure 13. Agents consist of several parts: the *search strategy* takes input from source code analysis and previous experiments and produces *candidate properties*, which are properties that would hold if the performance problem detected by the agent exists. These candidate properties are used to formulate experiments, which, when run, result in *measurement requests* being sent to the measurement system, describing what is to be measured (e.g., a set of of PAPI counters for a particular loop). When the results of the measurement request are available, they are stored in a performance database and the candidate property is evaluated in light of the new data. If the property holds, it is added to a set of *proven properties*, which are available to the search engine for its use in formulating new candidates. When no more candidates can be generated, the proven properties are analyzed to determine whether the performance problem is present or not.

6.2 Trace Based Systems

Wolf et al. [126] developed a system, KOJACK, to automatically diagnose performance problems in MPI and OpenMP codes. Programs are instrumented so that each process writes events to a process-specific log which are merged at program termination. Events which are logged include MPI sends, receives, and collectives, entry into and exit from OpenMP regions, and acquisition and release of OpenMP locks. A library of rules is constructed specifying patterns which indicate potential causes of performance problems. For example, one rule specifies that when a receive event is encountered while processing the event log, the corresponding send event should be located and the time between send and receive calculated to determine whether a "late sender" problem occurred, where an `MPI_Recv` call was made prior to the corresponding `MPI_Send`, resulting in the receiving process unnecessarily waiting. These rules are applied to the merged event log.

Scalasca [43, 42] is derived from KOJACK and addresses two problems: first, that creating a merged log is time consuming and can result in a file too large for some filesystems, and second, that serially scanning a merged log scales poorly as the number of processors in the traced application increases. In Scalasca, no merging is done; rather, each process writes its own local event log. The log is then processed in parallel, using the same number of processors as the application being analyzed. Rather than reducing all data to one node, the communication patterns of the original application are replayed, so that, for example, an `MPI_Send` in the original application becomes an `MPI_Send` in the replay with a payload indicating the parent events of the send.

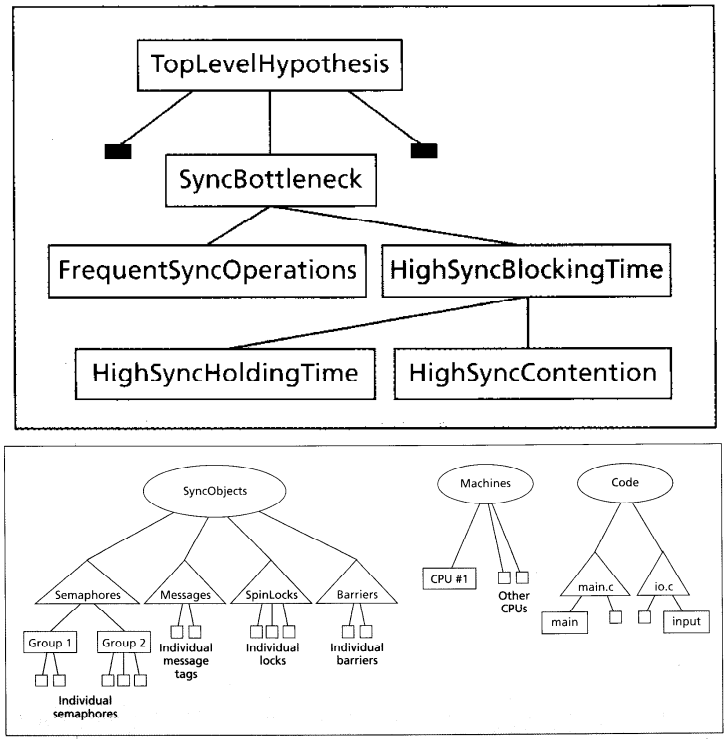


Figure 12: Examples of the Paradyn "why" and "when" hierarchies, from [84].

Scalasca has been subsequently extended with new analyses. One such analysis, described by Böhme et al. [13], aims to automatically determine the causes of load imbalance in MPI applications. A wait state can be either *direct* if it is caused by a process blocking on communication from another process because that other process has not yet completed a computation, or it can be *indirect* if it is waiting on a receive because the other process is in turn waiting on a communication. The authors extend Scalasca with a *backwards* replay, allowing wait states to be attributed to other wait states or to delays in computation, thereby building a graph showing the root cause of the delays.

6.3 Automatically Fixing Performance Problems

Of particular interest are systems which not only automatically diagnose performance problems, but also can suggest solutions to the problem or even automatically modify source code. Cong et al. [24] describe a system with a structure similar to KOJACK, described above, but which is closely integrated with IBM compilers, taking as input reports on what optimizations were applied to blocks of input code, and able to provide optimization settings to the compiler in response to diagnosed prob-

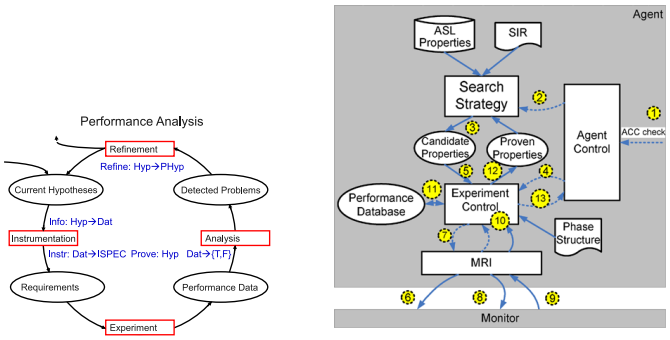
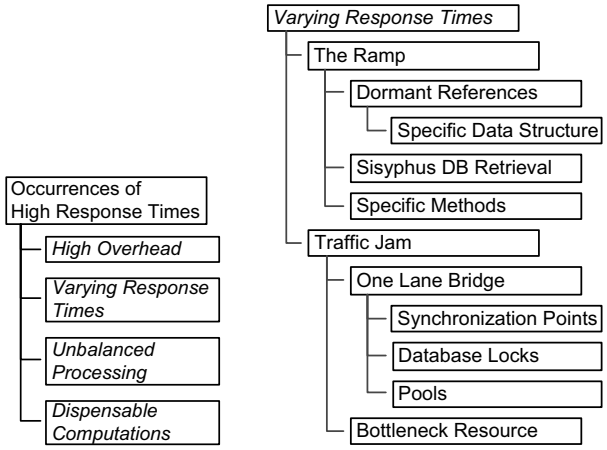


Figure 13: Architecture of the PERISCOPE, from [83].

lems, as well as transformation recipes to a polyhedral code optimization framework. Modeling or empirical testing are used to determine whether the proposed solution actually addresses the detected problem. Problems which cannot be addresses automatically result in suggestions to the user.

Recent versions of the PerfExpert system also implement automatic optimization [34], incorporating a central database which a set of modules access. Compilation modules encapsulate procedures for compiling and running input code. Measurement modules perform code instrumentation (which may entail cooperation with a compilation module), binary instrumentation, or monitoring through operating system facilities, and write measurements into the database. In this framework, measurements are distinct from metrics: a measurement is raw data collected during execution, while a metric has been further processed and rendered into a standard form. Analysis modules convert measurements into metrics, storing these into the database as well. Recommendation modules query the database, evaluating rules expressed as SQL queries. Each row returned by the query identifies a recommendation for an optimization and gives a ranking to that recommendation. The top-ranking recommendation is then applied using an optimization module, which first checks to verify that the recommendation actually applies and is valid given constraints inferred through static analysis of the input code. The recommendation, having been applied, results in new code which starts the process anew with a compilation module. This process continues until no more valid recommendations remain.

Wert et al. [120] perform automated performance diagnosis in the context of enterprise Java applications. In their system, a hierarchy of symptoms is specified, with each symptom in turn referring to a hierarchy of causes. An example of such a hierarchy is shown in Figure 14. For each symptom and cause, a detection strategy is provided, providing steps by which an automated experiment can be performed which will trigger the problem if the cause under consideration exists in the application being tested. The detection strategies specify a workload to apply to the application, measurements to be made, and a procedure for deciding whether the measurements support the hypothesized cause.



(a) Symptoms of known performance problems.

(b) Performance problems causing *Varying Response Times*.

Figure 14: Symptom and cause hierarchies as used by Wert et al., from [120]

6.4 Differential Analysis

Differential, or decremental, analysis is a technique for automated diagnosis of performance bottlenecks, with attribution to specific lines or operations in the original source code. First, binary analysis is performed using MAQAO [28], which produces a series of reports on degree of vectorization, utilization of execution units, and a series of performance estimates assuming that all memory requests are served from L1, that all memory requests are served from L2, that all memory requests are served from RAM, and finally a projection of performance for a fully-vectorized code. These reports are used to determine code regions for further analysis [69]. Selected loops are instrumented and run, with hardware performance counters related to the memory system being recorded. This generates hypotheses about the cause of performance bottlenecks. Finally, DECAN [70] performs differential analysis to determine the specific instructions causing the bottleneck. Given a binary executable, the instructions representing loops of interest are deleted or replaced with other instructions so as to suppress the effect of the instructions. This is done several times, yielding modified binaries in which certain classes of instructions are suppressed, such as one version suppressing load/store instructions and another suppressing floating-point instructions. These versions are then run with performance instrumentation, and the versions are compared to determine, for example, whether load/store (memory) or floating-point (compute) instructions are the performance bottleneck for the loop of interest.

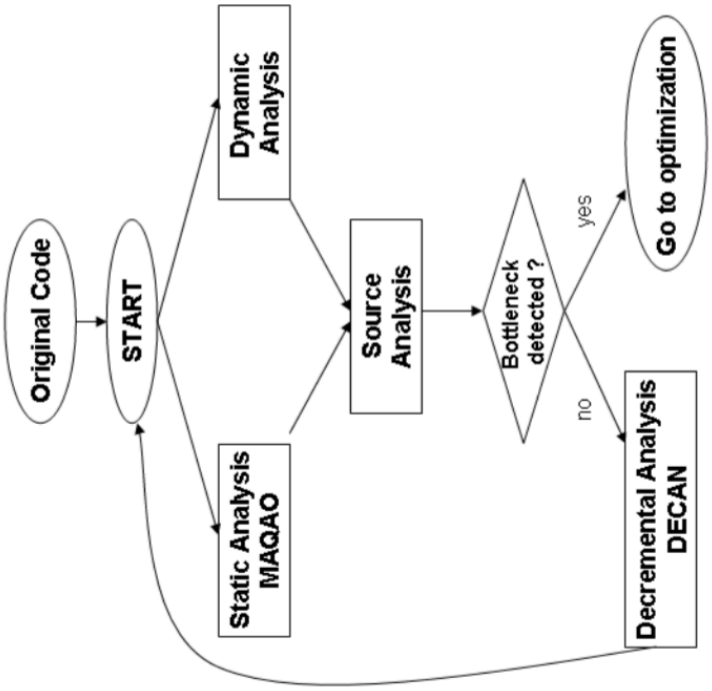


Figure 15: Methodology of DECAN, from [70]

7 Exascale Computing and Future Programming Models

All of the work described up to this point in the paper applies to existing supercomputers running existing codes written with traditional programming models such an MPI and OpenMP. The move to exascale, however, is likely to necessitate moves to other programming models [5]. An exascale system is one with peak performance of one exaflop (10^18 floating point operations per second), about 30 times greater than the peak performance of Tianhe-2, currently the world's fastest supercomputer [115]. Yet in order for system deployment to be feasible, total power consumption of the system must be kept below about 20 megawatts. Tianhe-2 uses 17 megawatts, so to reach exascale we must increase performance by 30 times while holding power consumption basically constant. This will require adding substan-

tially more concurrency at every level of the system: nodes must have more cores, cores must have more hardware threads, hardware threads must process SIMD instructions over more data at a time, all of which will result in the number of threads required to saturate the system growing from hundreds of thousands in current systems to tens to hundreds of billions in exascale systems. Providing enough work to generate these threads will require a different approach to programming [26].

Programming models that have been proposed for exascale systems tend to be *task based*. Rather than strictly dividing work across things like loop iterations, or partitioning work across nodes and running the same algorithms on every node on different parts of the data, task parallelism divides work into discrete chunks which carry dependency information. This dependency information can be expressed as a directed acyclic graph, allowing a runtime scheduler to proceed with executing a task as soon as its dependencies have been met. This allows task-parallel programs to spend less time idle compared to those using fork-join parallelism and communicating sequential processes, as shown in Figure 16. They also allow for easier adaptation to system variability by allowing work to migrate to address load imbalance caused by node variability; to do this, units of work are virtualized relative to hardware. Data is often also virtualized, so that data can be moved to work, or work can be moved to data, depending upon whichever is cheaper. Finally, by generating a very large number of tasks, latency can be hidden by swapping out a task waiting on a resource for another task [106].

In this section, I will review a number of task-based programming models. These differ by granularity (whether tasks are lightweight, at the level of loop iterations; medium-weight, at the level of functions; or heavy-weight, at the level of phases or steps in a workflow); by whether parallelism is explicit or implicit; by underlying source of parallelism (e.g, user-level threads, pthreads, systems built on top of MPI, etc.); by technology used by communication; by whether tasks may yield; by whether scheduling decisions are centralized or distributed; and by whether scheduling decisions are made statically or dynamically.

There are a number of node-local task based systems. While these could be combined with some other mechanism for inter-node parallelism, exascale systems

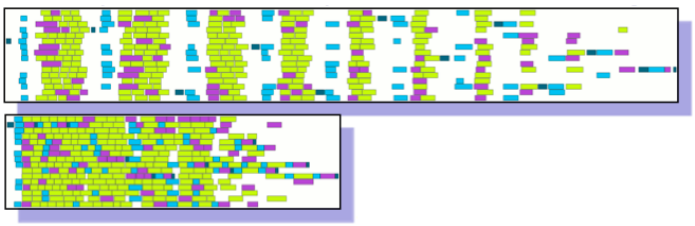


Figure 16: Execution trace of the same algorithm implemented using fork-join parallelism (top) and task-based parallelism (bottom). The bottom version executes in less time because worker threads can continue executing tasks as soon as the tasks' dependencies have been satisfied. From [130].

will likely require that intra- and inter- node parallelism be expressed using the same model. Therefore I will not describe node-local systems in detail. These systems include OpenMP Tasks [7], Intel Threading Building Blocks [93], Qthreads [123], StarPU [6], Cilk Plus [98], and Concurrent Collections (CnC) [15].

7.1 Charm++

Charm++[1] is among the oldest adaptive asynchronous task-based runtimes, first released in 1992. Its central abstraction is the *chare*, a special C++ object encapsulating data and methods which can be invoked remotely by receipt of a message. Programs do not interact with the chare directly. Rather, creation of a chare yields a *proxy object* through which messages are sent, invoking *entry methods*, which are specially designated methods with signatures defined in a domain-specific language from which glue code is generated by a source-to-source compiler. Entry methods are required to run to completion; the scheduler will not interrupt them.

All messages are asynchronous: upon sending a message, the sender immediately continues executing. Any reply to a message is implemented as an additional message. A chare's global ID indicates a home node for the chare; however, chares are *migratable*: at any time a chare may be moved to another node, with the original home node forwarding any messages it receives and notifying senders of the new location of the chare, which is cached by senders for future use. Application developers are encouraged to *overdecompose* their applications by breaking them down into many more tasks than there are processing units on which the tasks will run. This helps with load balancing by keeping a pool of work available to assign to processing units as they become available. Migratability provides additional opportunities for load balancing by enabling the moving of work, along with its associated data, to underutilized nodes [65].

The Charm++ runtime has built-in facilities for runtime adaptation. The Charm++ Load Balancing framework, the architecture of which is shown in Figure 17, is one such facility [135]. A Load Balance Manager runs on each node. During execution, the Manager stores statistics on load and idleness into a database. When criteria for rebalancing are met, the Manager invokes one or more Load Balancing Strategies, which can query the database for information on the load of the local node and remote nodes. Strategy instances are themselves chares and can communicate with one another through message passing. Strategies inform the Load Balance Manager of how chares should be migrated, which occurs through interaction with the Array Managers.

Three types of load balancers are described in [135]: centralized, decentralized, and hybrid. The centralized load balancers send all performance data to one node, which processes all the data and distributes migration decisions. The simplest of these are the Random strategy, which randomly assigns chares to processing units. The Greedy strategy processes chares in order from longest-running to shortest-running, assigning tasks to processors ordered from least-loaded to most-loaded. The Refinement strategy swaps chares to adjust an existing distribution. More sophisticated load balancing strategies take communication into consideration, attempting to place groups of chares which communicate heavily together while still

balancing load. These operate on the communication graph and include a Recursive Bisection strategy and a METIS [67] strategy. Variants of the above strategies are provided which consider that an application may be composed of several phases with different performance characteristics, which require gathering and using phase-specific load statistics.

As the size of the system increases, it becomes impractical to collect all the data needed for load balancing onto a single node. At the same time, making good load balancing decisions requires global information – we cannot decide to place work on the least-loaded node unless we know which node that is. Distributed strategies include neighborhood-based methods in which balancing occurs within a subset of nodes. This can be combined with a work-stealing approach, in which nodes in a neighborhood periodically send messages to one another informing them of their load, and idle nodes send messages to nodes which according to its view are overloaded, requesting that chares be migrated from the overloaded node to the idle node. These messages are prioritized for immediate processing, rather than being enqueued for later processing as with normal messages. The Hybrid strategy involves a tree of load balancing domains, with different strategies being used at different levels of the tree.

An adaptive runtime system called PICS [108] (Performance-analysis-based Introspective Control System) has been implemented, which allows Charm++ applications to register *control points* [29]. Control points specify what effect application parameters have on various categories of performance-effecting properties, a library of which are provided by the system. Control points can be registered for effect types of Degree of Parallelism, Grain Size, Priority, Memory Consumption, Cache

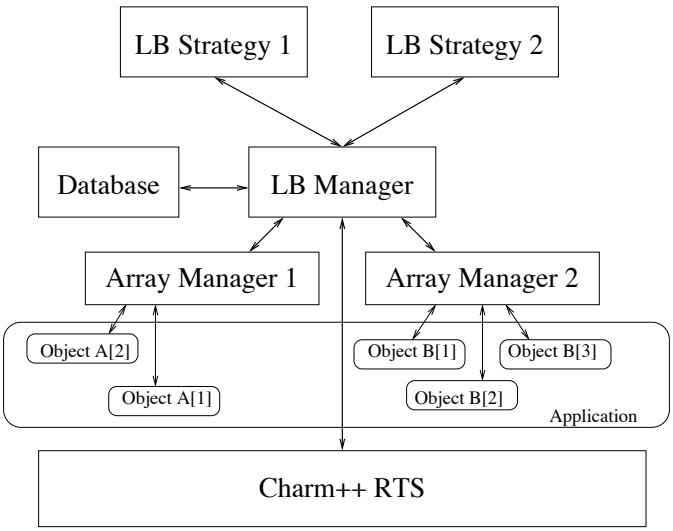


Figure 17: Architecture of the Charm++ Load Balancer (from [135])

Miss Rate, Overhead, Number of Messages, and Message Size. Control points are registered explicitly by the application developer and are not automatically discovered; for example, the application can register that a variable controlling the size of a subproblem will change the grain size and degree of parallelism. Based on runtime performance measurement, the system selects a property to adjust and adjusts registered control points according to a strategy shown in Figure 18.

A version of MPI, Adaptive MPI (AMPI), has been developed, which runs on top of the Charm++ runtime [54]. In AMPI, MPI processes are implemented as fully migratable Charm++ tasks, and MPI communications are implemented as Charm++ messages between tasks. The same load balancing strategies described above for native Charm++ programs can also be used for AMPI programs [55].

7.2 Swift

Swift [124][2] is a parallel scripting language designed for the specification of scientific workflows. Unlike general-purpose languages, Swift is not intended for performing mathematical operations but rather for sequencing and scheduling calls to external functions or entire executables written in other languages, such as C, C++, or Fortran. Swift is made aware of the types of inputs and outputs to such external computations, but they are otherwise treated as "black boxes" of which the Swift runtime has no knowledge.

A Swift program then consists of a series of parallel constructs, such as `foreach` loops, which contain external calls with specific inputs and outputs. Executions of a parallel construct implicitly specify tasks, so that, for example, two nested `foreach` loops each over 1,000 elements result in the construction of 1,000,000 tasks. Code such as

```
foreach i in [0:N-1] {
  foreach j in [0:N-1] {
    foreach k in [0:N-1] {
      foreach m in [0:N-1] {
        int r = f(i, j, k, m);
```

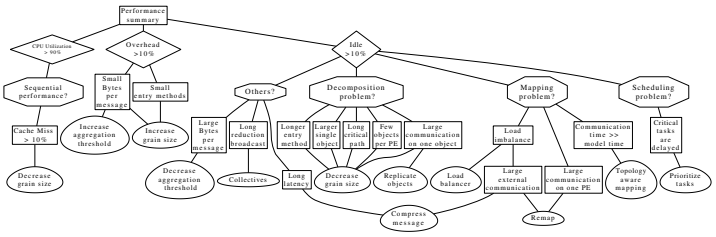


Figure 18: Decision procedure by which PICS decides which control points to adjust (from [108])

[2]Unrelated to the language of the same name from Apple.

```
      }
    }
  }
}
```

creates N tasks which run independently, while

```
A[0][0] = 0;
foreach i in [1:N-1] {
  A[i][0] = 0;
  A[0][i] = 0;
}
foreach i in [1:N-1] {
  foreach j in [1:N-1] {
    A[i][j] = f(A[i-1][j-1], A[i-1][j], A[i][j-1]);
 }
}
```

creates N-1 initialization tasks which run independently and N-1 tasks, each of which depends on predecessor tasks.

A limitation of the original Swift is that scheduling occurs only on the node executing the driver script, limiting the scalability of scheduling. Swift/T [128] resolves this issue by running Swift on top of a new runtime, Turbine [127]. A small subset of the nodes in a job are reserved as *control engines*, which run *control fragments*, which in turn schedule *leaf tasks* (that is, user-defined external functions or executables) on the *workers*, which are those nodes not reserved as control engines. Workers and control engines communicate through a global address space called the *distributed future store* which manages write-once variables by which tasks return results and signal completion.

Static dataflow analysis is used to determine dependencies between tasks, which are made available to the scheduler, which does not schedule a task for execution until all of its inputs are available. As tasks never execute that point, tasks do not yield during execution, instead always running to completion before the scheduler may reuse the resources consumed by the task. Because the scheduler must monitor dependencies itself, scheduling overhead is higher than in dependency-unaware runtimes. Swift is then intended for medium-granularity tasks, with fine-grained parallelism expressed in the native language used to define leaf tasks. This is in contrast to lightweight tasking runtimes, which are intended to support multiple task granularities.

7.3 X10

X10 [22] is a PGAS language based on Java, to which it adds the concepts of *places* and *asynchronous activities*. Places contain data and activities run in a place, and both data objects and activities are *not* independently migratable, unlike in Charm++. However, places themselves *may* move: they do not directly correspond to a node or processor. When a place migrates, all data objects and activities in that place move with it. Activities (equivalent to *tasks* in other languages and runtimes) are launched

with the code `async (p) S` where `p` is a place and `S` is a code block. An asynchronous activity invocation returns to the invoking process immediately. Waiting for a code block containing asynchronous activity invocations can be accomplished with `finish S`, where `S` is a code block. For example, in this simple implementation of Fibonacci,

```
static def fib(n:Int):Int {
 if(n < 2) return n;
 val f1:Int;
 val f2:Int;
 finish {
  async f1 = fib(n-1);
  f2 = fib(n-2);
 }
 return f1 + f2;
}
```

the statement `async f1 = fib(n-1)` launches a new activity which executes `fib(n-1)` (in the current place, since none is specified) and immediately continues to the next statement, `f2 = fib(n-2)`, which executes inside the current task. Since both statements are located inside a `finish` clause, once the second statement finishes the current task will wait for any subtasks launched within the block to complete before proceeding. Activities can be suspended during execution, unlike in Swift and Charm++.

As a PGAS language, X10 has support for arrays with elements resident in different places. Arrays are specified by *regions*, which specify the number of dimensions in the array and the extent of each dimension, and by *distributions*, which assign points in an array's region to a place. However, unlike traditional PGAS languages such as UPC, in which any node can access any address in the global address space, X10 restricts access to mutable (non-`final`) data to only the place in which it resides. For one place to access data stored in another place, the first place must launch an activity in the second place. If we have two arrays `A` and `B` such that `A[i]` and `B[j]` are located in different places, and we want to carry out the assignment `A[i] = B[j]`, we must launch multiple activities: one in the place where `B[j]` resides, to read its value, and one in the place where `A[i]` resides, to assign the value read from `B[j]`, as in this example:

```
finish async (B.distribution[j]) {
  final int bb = B[j];
  async (A.distribution[i]) {
    A[i] = bb;
  }
}
```

Here, the inner activity is able to read the value stored in `bb` because it is declared `final`, and non-mutable values can be read from any place.

Dependencies are managed through futures or through an abstraction called *clocks*, a version of a barrier in which an activity can be registered on an arbitrary number of clocks and can simultaneously advance all clocks on which it is registered,

which can be used to implement producer-consumer activities. The `next` statement causes the current activity to suspend until all clocks on which it depends have been advanced by calling `advance` on the clocks.

7.4 Chapel

Chapel [20] is a PGAS language providing abstractions which are very similar to those in X10, as described in Section 7.3. The statement `begin S` causes the current task to launch a new task which executes the code block `S`, while the current task immediately continues executing; this is equivalent to the `async` statement in X10. The statement `sync S` executes the statements in the code block S, then blocks until all subtasks created within `S` have completed; this is equivalent to `finish` in X10. As in X10, arrays can have arbitrary indices and customizable assignments of points to locales through user-definable *domain maps*, or *dmaps*. The primary difference between Chapel and X10 is that Chapel supports access to shared objects from any locale, as in traditional PGAS languages, while X10 restricts access to the place in which an object resides. Chapel also supports additional constructs for task creation, such as `cobegin S`, which launches a separate task for every statement in `S`, and `coforall E in C do S`, which launches a task executing the statements in `S` for every element `E` in the iterable collection `C`.

7.5 UPC++

UPC++ [136] is a C++ library which implements PGAS functionality as found in UPC along with asynchronous task support, which is not a feature of UPC. Rather than extend C++ with new keywords and types, as UPC did with C, UPC++ adds PGAS support purely as a library through the use of C++ templates. The UPC `shared` keyword applied to value types becomes the template `shared_var<underlying_type>`, while `shared` pointers become the template `global_ptr<underlying_type>`. Using a `shared_var` in a context in which the underlying type is expected is transparently converted to a local or remote memory access as needed by an implicit conversion operator. Dereferencing a `global_ptr` is also transparently converted to a local or remote memory access. A local `global_ptr` can also be cast to a plain pointer to reduce overhead when it is known not to be remote. Direct support is available for allocating memory from one node which is resident in the memory of another node, a feature not found in UPC. Multidimensional arrays are supported similarly to X10 and Chapel.

Asynchronous tasks can be launched using `future<T> f = async(place)(function, args)`, where `function` is a callable object returning `T`. The returned `future` can be used to retrieve the value computed by `function` by calling `f.get()`, which blocks until the task has completed. UPC also provides a `finish` construct analogous to the one in X10, and an event-based system for building a dependency DAG, in which an `async` optionally takes `event` objects to signal completion and to hold execution of a task until a set of events have been signaled. Unlike in Charm++ and Swift, tasks are non-migratable. Tasks are intended to be launched only on remote nodes.

Habanero-UPC++ [71] allows both local and remote task invocation and extends the runtime with additional work-stealing support.

7.6 Open Community Runtime

The Open Community Runtime [79] is an asynchronous task-based runtime. Unlike the other systems described thus far, OCR provides a runtime *only*; it is not accompanied by a user-facing language or library, and is intended as a target for third-party languages and libraries. OCR is based on three abstractions: *Event-Driven Tasks*, or EDTs, asynchronous tasks which, once started, are required to run to completion; *Data Blocks*, which represent globally-accessible data, and *Events*, which connect EDTs, Data Blocks, and other events together. EDTs have *input slots* and *output slots* which may connect directly to Data Blocks or to Events. An example DAG is shown in Figure 19 for a Fibonacci program.

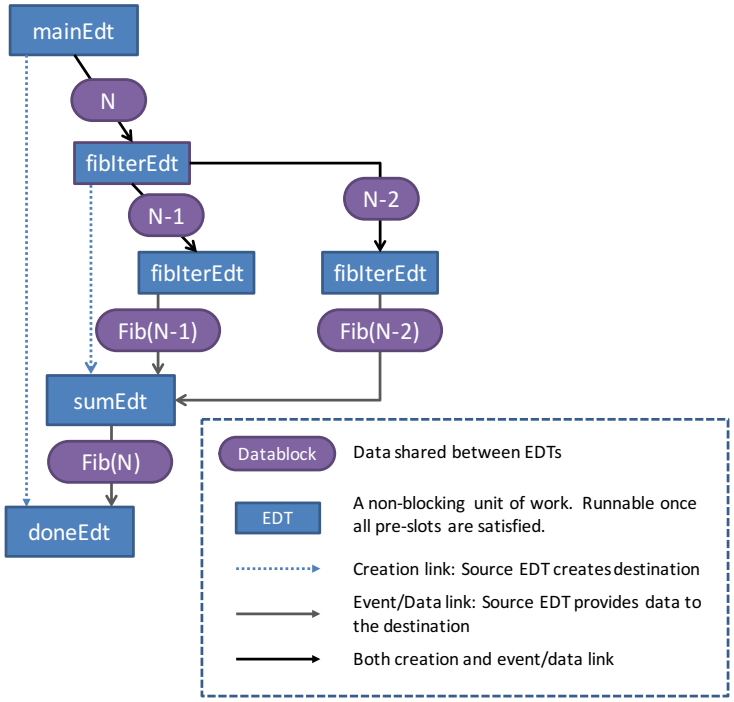


Figure 19: DAG for an OCR Fibonacci code (from [112]). Blue rectangles are EDTs, purple rounded rectangles are Data Blocks, and arrows are Events.

With Data Blocks, OCR makes data an explicit part of the dependency graph, unlike most other systems. Events linking Data Blocks to other objects carry information on how they are to be accessed, allowing the runtime additional optimization opportunities: by default, a Data Block is in *read-write mode*, so that the runtime can make no assumptions about which EDTs will access the Block. Also available are

exclusive write, in which only one EDT may write to the block at a time; *read only*, in which the Data Block provided by the event may not be written to by the target EDT, and *constant*, in which *no* EDT may write to the Block.

7.7 Legion

Legion [10] is a task-based runtime with a unique data abstraction called *Logical Regions*. As with OCR's Data Blocks, Logical Regions represent data in a global address space and associates with it access restrictions, namely *privileges* (read-only, read/write, *etc.*) and *coherence* (exclusive access, atomic access, *etc.*). As with arrays in X10 and Chapel, the assignment of ownership of array elements is separate from declaration of the array extent. However, unlike in OCR, X10, or Chapel, Legion's Logical Regions do not impose *any* physical data layout, deferring this decision until a task using the region is to be executed.

A Logical Region encodes what types of data are to be stored, but says nothing about the physical representation of the data. Regions are then *partitioned* into *subregions*, with partition operations being annotated as either *disjoint* (that is, no two subregions of the region share data) or *aliased* (subregions may overlap). At runtime, a *mapper* function determines the distribution of data to nodes and also the physical layout of subregions on a node. Legion provides a default mapper with functionality similar to distributions in X10 or domain maps in Chapel. Custom mappers can be provided which take into account architecture-specific properties (such as choosing structure-of-arrays vs array-of-structures depending on whether a CPU or GPU is targeted) as well as application-specific properties (such as a graph partitioner tuned to the properties of graphs used in an application). Different tasks can use different mappers for the same regions, in which case the runtime will dynamically reformat the physical representation.

7.8 Grappa

Grappa [86] is a task-based runtime and C++ library with generally similar features to UPC++, providing a tasking model with a partitioned global address space. As with X10, only the owner of a memory address is allowed to directly access it, with remote access being performed through remote task invocation. In most PGAS systems, such as UPC, UPC++, X10 and Chapel, memory partitions are associated with *nodes*, so that if thread A and thread B are located on the same node, and thread A accesses shared memory located in thread B, the access happens *directly* and does not go through the remote memory subsystem. Grappa does not partition memory in this way: ownership is associated with a *core*, not with a node. If worker A and worker B are running on two cores of the same node, and worker A runs a task which accesses memory owned by worker B's core, then a task must be scheduled on worker B to perform that access and return the result to the task on worker A. Tasks whose only purpose is to access remote memory are called *delegate* tasks and are not allowed to context switch or block. Full-fledged tasks may block, in which case they will be suspended and another task scheduled in their place.

The high-granularity memory partitioning used in Grappa enables an approach to global data structures with low contention, known as *flat combining* [53]. Instead of acquiring a lock to access the shared data structure, per-core lists of pending requests are maintained. When a worker attempts to access a non-local part of a global data structure, it adds the request to the list associated with the core owning the memory to be modified and then blocks, causing another task to be scheduled in its place. Periodically, combining workers are scheduled on each core, which process requests in the order in which they were received.

7.9 HPX

HPX [64] is an asynchronous task-based runtime and C++ library based on the ParalleX model [62]. The distinguishing feature of HPX is its adherence in design to C++ standards. C++11 [60] added node-local tasks to the C++ standard library in the form of `std::async` to launch a task, which returns an object of type `std::future` which can be used for synchronization and to retrieve the value returned by the task. HPX makes this same model available for distributed systems, so that an existing C++11 application making use of `std::async` and `std::future` for parallelism can be converted to an HPX application by simply replacing them with `hpx::async` and `hpx::future`. Remote invocation of a task is accomplished by passing an argument to `hpx::async` indicating on which *locality* the task should run. Sending data and work is accomplished by means of a *parcel* abstraction. Notably, HPX provides for transparent task migration, meaning that tasks can migrate without stopping other computations which are occurring on the node. During migration, any incoming messages intended for the tasks or data being migrated will be stored for automatic forwarding once migration is complete. The architecture of HPX is shown in Figure 20.

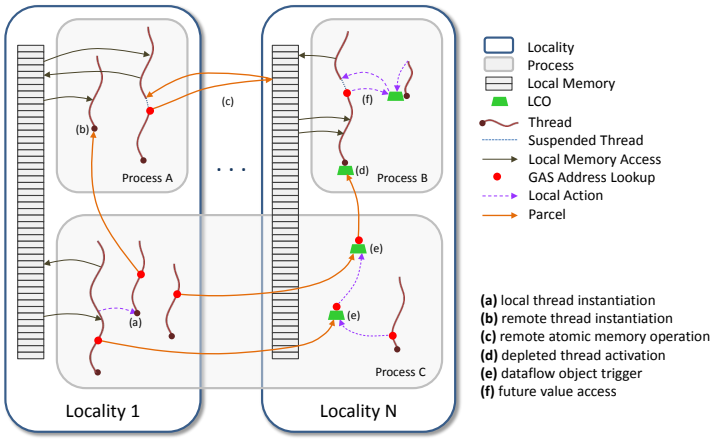


Figure 20:

HPX has recently been extended with a new mechanism for implicitly creating tasks, known as *executors* [63]. With executors, parallel implementations of Standard Template Library algorithms can allow decisions as to how to distribute work to be deferred to external libraries such as HPX. Algorithms which support executors take an executor object as the first argument, which in turn receives lambda functions from which it creates tasks. The executor is free to determine how much work to assign to a given task, and how to distribute tasks in a multi-node setting.

7.10 Spark

Spark [134] is based on a generalization of the Map-Reduce model [27] found in systems such as Hadoop to problems expressed as general data flow graphs, relaxing the restriction that the graphs be acyclic. Operations are carried out on *resilient distributed datasets* [133], or RDDs, which store data across nodes and which carry sufficient information to recompute their contents. Programs are expressed in terms of RDDs derived from transformations (of which map is only one) applied to other RDDs and actions (of which reduce is only one). The application developer can choose to request that certain RDDs be cached in memory or saved to disk. The developer therefore has to make decisions based on tradeoffs between the costs of storage (in memory and time) and recomputation (in time). RDDs are lazily evaluated, which can create challenges in attributing performance to particular lines or regions of code, as they do not execute until they are needed.

RDDs are composed of *blocks*, which represent data. Data storage is also managed by the runtime: while the runtime will attempt to keep data in memory, it is also free to evict data from memory, dropping it to disk instead, or to drop it entirely, requiring that it be recomputed if needed again in the future.

8 Conclusion and Future Directions

For programs written for current-generation supercomputers and using programming models such as MPI and OpenMP, a wide variety of performance analysis tools are available for collecting profiles and traces, for analyzing and visualizing profiles and traces, for offline tuning and online adaptation using automatic performance tuning, for automatic diagnosis of performance problems, and for construction of models from performance data. The move to exascale, however, will require such a large number of threads that programming using MPI and OpenMP will become difficult, and runtimes being investigated for exascale use a different structure for specifying programs: directed acyclic graphs of light-weight or medium-weight tasks for both intra- and inter-node parallelism. Existing techniques for collecting and making use of performance data are not suitable for analysis of systems of billions of light-weight tasks, so new techniques will need to be developed to go along with new programming models, runtimes, and languages at exascale. It will not be feasible, for example, to collect a trace of the start and stop times of many billions of tasks.

Many-tasks systems have many additional layers of abstraction over systems like MPI, and this can cause us to lose the connection between a source line and why

it is executing, or why it is not executing. In MPI, we can observe that we are waiting on a receive and work backwards to a cause, such as a late sender. In a DAG based system, the cause can be far removed from its effect, or can depend instead on scheduling policy: Why has task A not executed? Because it is waiting on data from task B. Why has task B not executed? It is eligible to; the scheduler has simply not scheduled it yet, as there are many tasks eligible for scheduling. What schedule yields the best throughput? Why is this task executing instead of some other task? Why is this worker idle now? How are hardware resources shared between worker threads? How can hardware counter values be attributed to tasks when there are multiple tasks and tasks can suspend and resume?

Because of the huge number of tasks in a system, we will need to answer these questions without using post-mortem analysis, as this would require saving too large a volume of data to disk, yet most existing studies of performance in task-based systems have used post-mortem analysis of short runs or on a small number of nodes [48, 21]. Performance monitoring at exascale will require *in-situ performance analysis* [72] and *online adaptation* [45]. This will require both node-local performance data and decision making as well as a *global view* [57] on performance through which nodes can become aware of the state of other nodes so that they can best make local decisions, as centralized control will likely be infeasible at exascale.

No in-situ system providing online adaptation for a task-based runtime through a global view currently exists. The adaptive load balancing system used in Charm++, described in Section 7.1 is close, but is limited to controlling migration and does not affect other system parameters, while Charm++'s PICS system operates on a per-node basis. Node-local adaptation based on contention for memory controller resources has been demonstrated for OpenMP tasks [4] and HPX [78]. A prototype in-situ performance monitoring tool providing a global view, GTI-OTFX [119], has been developed, but only supports traditional MPI applications.

We are currently developing a system, APEX [56], with prototype implementations for HPX and OpenMP tasks and planned support for other task-based systems such as OCR. APEX is built around the concept of a *policy*, which can be registered to respond to events of interest produced by an instrumented runtime. While policies ultimately run on a single node, they run as tasks within the task-based runtime and have access to the same communications infrastructure as any other task; thus, in HPX, they can communicate with one another using one-sided puts and gets in the global address space. Built-in support in HPX for efficient reductions can be used to aggregate performance data. We envision ultimately having a system in which a small portion of localities are reserved for performance analysis and adaptation, running analysis tasks which receive data from lighter-weight tasks which collect and forward performance data from compute localities.

References

[1] Bilge Acun et al. "Parallel Programming with Migratable Objects: Charm++ in Practice". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana:

IEEE Press, 2014, pp. 647–658. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.58. URL: http://dx.doi.org/10.1109/SC.2014.58.

[2] Laksono Adhianto et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[3] Inc. Advanced Micro Devices. *AMD GPU Performance API User Guide*. 2015. URL: http://developer.amd.com/wordpress/media/2013/12/GPUPerfAPI-UserGuide-2-15.pdf.

[4] Allan Porterfield et al. *Adaptive Scheduling Using Performance Introspection*. TR-12-02. RENCI, 2012. URL: http://www.renci.org/technical-reports/tr-12-02/ (visited on 05/01/2014).

[5] Saman Amarasinghe et al. "Exascale programming challenges". In: *Report of the 2011 Workshop on Exascale Programming Challenges, Marina del Rey*. 2011.

[6] Cédric Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". English. In: *Euro-Par 2009 Parallel Processing*. Ed. by Henk Sips, Dick Epema, and Hai-Xiang Lin. Vol. 5704. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 863–874. ISBN: 978-3-642-03868-6. DOI: 10.1007/978-3-642-03869-3_80. URL: http://dx.doi.org/10.1007/978-3-642-03869-3_80.

[7] E. Ayguade et al. "The Design of OpenMP Tasks". In: *Parallel and Distributed Systems, IEEE Transactions on* 20.3 (Mar. 2009), pp. 404–418. ISSN: 1045-9219. DOI: 10.1109/TPDS.2008.105.

[8] Prasanna Balaprakash, R. Gramacy, and S. Wild. *Active-Learning-Based Surrogate Models for Empirical Performance Tuning*. 2013.

[9] Protonu Basu et al. "Towards making autotuning mainstream". In: *International Journal of High Performance Computing Applications* 27.4 (Nov. 1, 2013), pp. 379–393. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342013493644. URL: http://hpc.sagepub.com/content/27/4/379 (visited on 05/01/2014).

[10] Michael Bauer et al. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11. ISBN: 978-1-4673-0804-5. URL: http://dl.acm.org/citation.cfm?id=2388996.2389086.

[11] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. "Periscope: An online-based distributed performance analysis tool". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 1–16. URL: http://link.springer.com/chapter/10.1007/978-3-642-11261-4_1 (visited on 08/05/2015).

[12] Katharina Benkert and Edgar Gabriel. "Empirical Optimization of Collective Communications with ADCL". In: *High Performance Computing on Vector Systems 2010*. Ed. by Michael Resch et al. Springer Berlin Heidelberg, 2010, pp. 37–49. ISBN: 978-3-642-11850-0, 978-3-642-11851-7. URL: http://link.springer.com/chapter/10.1007/978-3-642-11851-7_3 (visited on 05/01/2014).

[13] D. Bohme, F. Wolf, and M. Geimer. "Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications". In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International. May 2012, pp. 2538–2541. DOI: 10.1109/IPDPSW.2012.321.

[14] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2014. URL: http://tools.ietf.org/html/rfc7159.html (visited on 06/26/2015).

[15] Zoran Budimlić et al. "Concurrent collections". In: *Scientific Programming* 18.3-4 (2010), pp. 203–217.

[16] Jong-Ho Byun et al. "Autotuning sparse matrix-vector multiplication for multicore". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-215* (2012). URL: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2012-215.pdf (visited on 07/10/2015).

[17] Rosario Cammarota et al. "Optimizing Program Performance via Similarity, Using a Feature-Agnostic Approach". In: *Advanced Parallel Processing Technologies*. Ed. by Chenggang Wu and Albert Cohen. Lecture Notes in Computer Science 8299. Springer Berlin Heidelberg, 2013, pp. 199–213. ISBN: 978-3-642-45292-5, 978-3-642-45293-2. URL: http://link.springer.com/chapter/10.1007/978-3-642-45293-2_15 (visited on 05/02/2014).

[18] Mohamad Chaarawi et al. "A Tool for Optimizing Runtime Parameters of Open MPI". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. Lecture Notes in Computer Science 5205. Springer Berlin Heidelberg, 2008, pp. 210–217. ISBN: 978-3-540-87474-4, 978-3-540-87475-1. URL: http://link.springer.com/chapter/10.1007/978-3-540-87475-1_30 (visited on 05/01/2014).

[19] Nicholas Chaimov, Boyana Norris, and Allen Davis Malony. "Toward Multi-target Autotuning for Accelerators". In: *International Conference on Parallel and Distributed Systems*. 2014.

[20] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442. eprint: http://hpc.sagepub.com/content/21/3/291.full.pdf+html. URL: http://hpc.sagepub.com/content/21/3/291.abstract.

[21] Kavitha Chandrasekar et al. "Task characterization-driven scheduling of multiple applications in a task-based runtime". In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM. 2015, pp. 52–55.

[22] Philippe Charles et al. "X10: An Object-oriented Approach to Non-uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852. URL: http://doi.acm.org/10.1145/1094811.1094852.

[23] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Utah, 2008.

[24] Guojing Cong et al. "A Systematic Approach toward Automated Performance Analysis and Tuning". In: *IEEE Transactions on Parallel and Distributed Systems* 23.3 (Mar. 2012), pp. 426–435. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.189.

[25] Tomasz S Czajkowski et al. "From OpenCL to high-performance hardware on FPGAs". In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE. 2012, pp. 531–534.

[26] Georges Da Costa et al. "Exascale Machines Require New Programming Paradigms and Runtimes". In: *Supercomputing Frontiers and Innovations* 2 (2015), pp. 6–27.

[27] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492 (visited on 06/17/2015).

[28] Lamia Djoudi et al. "Exploring application performance: a new tool for a static/dynamic approach". In: *Proceedings of the 6th LACSI Symposium*. 2005. URL: http://www.researchgate.net/profile/William_Jalby/publication/239735488_Exploring_Application_Performance_a_New_Tool_For_a_StaticDynamic_Approach/links/00b4952d6f80d5e051000000.pdf (visited on 09/14/2015).

[29] Isaac Dooley. "Intelligent Runtime Tuning of Parallel Applications With Control Points". http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml. PhD thesis. Dept. of Computer Science, University of Illinois, 2010.

[30] Peng Du et al. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[31] Juan Durillo and Thomas Fahringer. "From single-to multi-objective autotuning of programs: Advantages and implications". In: *Scientific Programming* 22.4 (2014), pp. 285–297.

[32] Alexandre Eichenberger et al. *OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging*. Tech. rep. Technical report, 2013.

[33] S. Feki and E. Gabriel. "Incorporating Historic Knowledge into a Communication Library for Self-Optimizing High Performance Computing Applications". In: *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO '08*. Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Oct. 2008, pp. 265–274. DOI: 10.1109/SASO.2008.47.

[34] Leonardo Fialho and James Browne. "Framework and modular infrastructure for automation of architectural adaptation and performance optimization for HPC systems". In: (2014), pp. 261–277. URL: http://link.springer.com/chapter/10.1007/978-3-319-07518-1_17 (visited on 07/10/2015).

[35] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Sept. 2012. URL: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[36] Matteo Frigo, Steven, and G. Johnson. "The design and implementation of FFTW3". In: *Proceedings of the IEEE*. 2005, pp. 216–231.

[37] Karl Fürlinger and Michael Gerndt. "ompP: A profiling tool for OpenMP". In: *OpenMP Shared Memory Parallel Programming*. Springer, 2008, pp. 15–23.

[38] Grigori Fursin. "Collective Mind: cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning". In: *arXiv:1308.2410 [cs, stat]* (Aug. 11, 2013). URL: http://arxiv.org/abs/1308.2410 (visited on 05/01/2014).

[39] Grigori Fursin. "Collective Tuning Initiative: automating and accelerating development and optimization of computing systems". In: GCC Developers' Summit. June 8, 2009. URL: http://hal.inria.fr/inria-00436029 (visited on 05/01/2014).

[40] E. Gabriel and S. Huang. "Runtime Optimization of Application Level Communication Patterns". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370406.

[41] Edgar Gabriel et al. "Towards Performance Portability Through Runtime Adaptation for High-performance Computing Applications". In: *Concurr. Comput. : Pract. Exper.* 22.16 (Nov. 2010), pp. 2230–2246. ISSN: 1532-0626. DOI: 10.1002/cpe.v22:16. URL: http://dx.doi.org/10.1002/cpe.v22:16 (visited on 05/01/2014).

[42] Markus Geimer et al. "A scalable tool architecture for diagnosing wait states in massively parallel applications". In: *Parallel Computing* 35.7 (2009), pp. 375–388. URL: http://www.sciencedirect.com/science/article/pii/S0167819109000398 (visited on 08/10/2015).

[43] Markus Geimer et al. “Scalable parallel trace-based performance analysis”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2006, pp. 303–312. URL: http://link.springer.com/chapter/10.1007/11846802_43 (visited on 08/10/2015).

[44] Michael Gerndt. “Performance analysis tools”. In: *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1515–1522.

[45] Georgios Goumas et al. “Adapt or Become Extinct!: The Case for a Unified Framework for Deployment-time Optimization (Position Paper)”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT ’11. New York, NY, USA: ACM, 2011, pp. 46–51. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000422. URL: http://doi.acm.org/10.1145/2000417.2000422 (visited on 05/01/2014).

[46] Martin Griebl, Christian Lengauer, and Sabine Wetzel. “Code Generation in the Polytope Model”. In: *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[47] OpenACC Working Group. *The OpenACC Application Programming Interface, Version 2.0*. 2013. URL: http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf.

[48] Patricia Grubel et al. “The Performance Implication of Task Size for Applications on the HPX Runtime System”. In: *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE. 2015, pp. 682–689.

[49] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. “Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage”. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Lecture Notes in Computer Science 8632. Springer International Publishing, Aug. 25, 2014, pp. 87–98. ISBN: 978-3-319-09872-2, 978-3-319-09873-9. URL: http://link.springer.com/chapter/10.1007/978-3-319-09873-9_8 (visited on 04/13/2015).

[50] Robert J Hall. “Call path profiling”. In: *Proceedings of the 14th international conference on Software engineering*. ACM. 1992, pp. 296–306.

[51] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. “Annotation-Based Empirical Performance Tuning Using Orio”. In: *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*. Rome, Italy, 2009.

[52] Jeffrey Hollingsworth and Ananta Tiwari. “End-to-End Auto-Tuning with Active Harmony”. In: *Performance Tuning of Scientific Applications*. CRC Press, June 2010. Chap. 10, pp. 217–238. ISBN: 978-1-4398-1569-4. DOI: doi:10.1201/b10509-11. URL: http://dx.doi.org/10.1201/b10509-11.

[53] Brandon Holt et al. “Flat Combining Synchronized Global Data Structures”. In: *7th International Conference on PGAS Programming Models*, p. 76.

[54] Chao Huang, Orion Lawlor, and Laxmikant V Kale. “Adaptive MPI”. In: *Languages and Compilers for Parallel Computing*. Springer, 2004, pp. 306–322.

[55] Chao Huang, Gengbin Zheng, and Laxmikant V Kalé. "Supporting adaptivity in MPI for dynamic parallel applications". In: *Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign* (2007).

[56] Kevin A Huck et al. "An Autonomic Performance Environment for Exascale". In: *Supercomputing frontiers and innovations* 2.3 (2015), pp. 49–66.

[57] Kevin A. Huck et al. "TAUg: Runtime Global Performance Data Access Using MPI". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Bernd Mohr et al. Lecture Notes in Computer Science 4192. Springer Berlin Heidelberg, 2006, pp. 313–321. ISBN: 978-3-540-39110-4, 978-3-540-39112-8. URL: http://link.springer.com/chapter/10.1007/11846802_44 (visited on 05/20/2015).

[58] Intel. *Intel SDK for OpenCL Applications - Performance Debugging*. 2013. URL: https://software.intel.com/en-us/articles/intel-sdk-for-opencl-applications-performance-debugging-intro.

[59] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[60] ISO. *Standard for Programming Language C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 28, 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[61] Changhee Jung et al. "Brainy: Effective Selection of Data Structures". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: ACM, 2011, pp. 86–97. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993509. URL: http://doi.acm.org/10.1145/1993498.1993509 (visited on 05/02/2014).

[62] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications". In: *Proceedings of the 2009 International Conference on Parallel Processing Workshops*. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401. ISBN: 978-0-7695-3803-7. DOI: 10.1109/ICPPW.2009.14. URL: http://dx.doi.org/10.1109/ICPPW.2009.14.

[63] Hartmut Kaiser et al. "Higher-level parallelization for local and distributed asynchronous task-based programming". In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*. ACM. 2015, pp. 29–37.

[64] Hartmut Kaiser et al. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: ACM, 2014, 6:1–6:11. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883. URL: http://doi.acm.org/10.1145/2676870.2676883.

[65] Laxmikant V. Kale and Gengbin Zheng. "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects". In: *Advanced Computational Infrastructures for Parallel and Distributed Applications*. Ed. by M. Parashar. Wiley-Interscience, 2009, pp. 265–282.

[66] Karen L Karavanic and Barton P Miller. "Improving online performance diagnosis by the use of historical performance data". In: *Supercomputing, ACM/IEEE 1999 Conference*. IEEE. 1999, pp. 42–42.

[67] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: 10.1137/S1064827595287997. URL: http://dx.doi.org/10.1137/S1064827595287997.

[68] Andreas Knüpfer et al. "The vampir performance analysis tool-set". In: *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[69] Souad Koliai et al. "A Balanced Approach to Application Performance Tuning". In: *Languages and Compilers for Parallel Computing*. Ed. by Guang R. Gao et al. Lecture Notes in Computer Science 5898. Springer Berlin Heidelberg, 2010, pp. 111–125. ISBN: 978-3-642-13373-2, 978-3-642-13374-9. URL: http://link.springer.com/chapter/10.1007/978-3-642-13374-9_8 (visited on 05/01/2014).

[70] Souad Koliaï et al. "Quantifying Performance Bottleneck Cost Through Differential Analysis". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. New York, NY, USA: ACM, 2013, pp. 263–272. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465440. URL: http://doi.acm.org/10.1145/2464996.2465440 (visited on 05/01/2014).

[71] Vivek Kumar et al. "HabaneroUPC++: A Compiler-free PGAS Library". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. New York, NY, USA: ACM, 2014, 5:1–5:10. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676879. URL: http://doi.acm.org/10.1145/2676870.2676879 (visited on 05/20/2015).

[72] Mahendra Kutare et al. "Monalytics: online monitoring and analytics for managing large scale data centers". In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 141–150.

[73] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 81–91. ISBN: 978-0-7695-3576-0. DOI: 10.1109/CGO.2009.21. URL: http://dx.doi.org/10.1109/CGO.2009.21 (visited on 05/01/2014).

[74] Chunhua Liao et al. "Early experiences with the openmp accelerator model". In: *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[75] Allen D Malony, Sameer Shende, and Alan Morris. "Phase-Based Parallel Performance Profiling." In: *PARCO*. 2005, pp. 203–210.

[76] Allen D Malony et al. "Parallel performance measurement of heterogeneous parallel systems with GPUs". In: *Parallel Processing (ICPP), 2011 International Conference on*. IEEE. 2011, pp. 176–185.

[77] Azamat Mametjanov et al. "Autotuning Stencil-Based Computations on GPUs". In: *Proceedings of IEEE Cluster 2012*. 2012.

[78] Anirban Mandel, Rob Fowler, and Allan Porterfield. "System-wide introspection for accurate attribution of performance bottlenecks". In: *Second International Workshop on High-perfromance Infrastruture for Scalable Tools*. 2012.

[79] T Mattson et al. *OCR: The Open Community Runtime interface version 1.1. 0*. 2015.

[80] Paul E McKenney. "Is parallel programming hard, and, if so, what can you do about it?" In: *Linux Technology Center, IBM Beaverton* (2011).

[81] Abdul Wahid Memon and Grigori Fursin. "Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing". In: PARCO mini-symposium on 'Application Autotuning for HPC (Architectures)'. Sept. 12, 2013. URL: http://hal.inria.fr/hal-00944513 (visited on 05/01/2014).

[82] Jiayuan Meng et al. "GROPHECY: GPU Performance Projection from CPU Code Skeletons". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. New York, NY, USA: ACM, 2011, 14:1–14:11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063402. URL: http://doi.acm.org/10.1145/2063384.2063402 (visited on 05/30/2014).

[83] Renato Miceli et al. "AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications". In: *Applied Parallel and Scientific Computing*. Ed. by Pekka Manninen and Per Öster. Lecture Notes in Computer Science 7782. Springer Berlin Heidelberg, 2013, pp. 328–342. ISBN: 978-3-642-36802-8, 978-3-642-36803-5. URL: http://link.springer.com/chapter/10.1007/978-3-642-36803-5_24 (visited on 05/01/2014).

[84] Barton P Miller et al. "The Paradyn parallel performance measurement tool". In: *Computer* 28.11 (1995), pp. 37–46.

[85] Bernd Mohr. "Scalable parallel performance measurement and analysis tools-state-of-the-art and future challenges". In: *Supercomputing frontiers and innovations* 1.2 (2014), pp. 108–123.

[86] Jacob Nelson et al. "Latency-tolerant Software Distributed Shared Memory". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 291–305. ISBN: 978-1-931971-225. URL: http://dl.acm.org/citation.cfm?id=2813767.2813789.

[87] *NERSC User Survey*. 2013. URL: https://www.nersc.gov/news-publications/publications-reports/user-surveys/2013/.

[88] Chris J Newburn et al. “Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor”. In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1213–1225.

[89] John Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500.

[90] NVIDIA. *CUDA Toolkit 7.5 CUPTI API Specification*. 2015. URL: http://docs.nvidia.com/cuda/cupti/.

[91] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. Nov. 2015. URL: http://www.openmp.org/mp-documents/openmp-4.5.pdf.

[92] S. Pellegrini, R. Prodan, and T. Fahringer. “Tuning MPI Runtime Parameter Setting for High Performance Computing”. In: *2012 IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS)*. 2012 IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS). Sept. 2012, pp. 213–221. DOI: 10.1109/ClusterW.2012.15.

[93] Chuck Pheatt. “Intel&Reg; Threading Building Blocks”. In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), pp. 298–298. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=1352079.1352134.

[94] Markus Püschel et al. “SPIRAL: Code generation for DSP transforms”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[95] Dan Quinlan. “ROSE: Compiler support for object-oriented frameworks”. In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226.

[96] Giridhar Ravipati et al. *Toward the deconstruction of Dyninst*. Tech. rep. Technical Report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp. cs. wisc. edu/paradyn/papers/Ravipati07Symta bAPI. pdf), 2007.

[97] James Reinders. “An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors”. In: (2012). URL: https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf.

[98] Arch D. Robison. “Composable Parallel Patterns with Intel Cilk Plus”. In: *Computing in Science and Engineering* 15.2 (2013), pp. 66–71, 87. DOI: http://dx.doi.org/10.1109/MCSE.2013.21. URL: http://scitation.aip.org/content/aip/journal/cise/15/2/10.1109/MCSE.2013.21.

[99] Philip C Roth and Barton P Miller. “On-line automated performance diagnosis on thousands of processes”. In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM. 2006, pp. 69–80.

[100] Gabe Rudy. “CUDA-CHiLL: A Programming Language Interface for GPGPU Optimization and Code Generation”. MA thesis. University of Utah, 2010.

[101] Santosh Sarangkar and Apan Qasem. "MATS: A Model-driven Adaptive Tuning System for Parallel Workloads". In: *Journal of Parallel and Cloud Computing (JPCC)* 1.2 (2012), pp. 50–64.

[102] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009.

[103] Sameer S Shende and Allen D Malony. "The TAU parallel performance system". In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[104] Jaewook Shin et al. "Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology". In: *Software Automatic Tuning*. Springer, 2010, pp. 353–370.

[105] Jaewook Shin et al. "Speeding up Nek5000 with autotuning and specialization". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM. 2010, pp. 253–262.

[106] Thomas Sterling et al. "SLOWER: A performance model for Exascale computing". In: *Supercomputing frontiers and innovations* 1.2 (2014), pp. 42–57.

[107] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69.

[108] Yanhua Sun, Jonathan Lifflander, and L. V. Kale. "PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications". In: *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*. Munich, Germany, June 2014.

[109] Lingjia Tang, Jason Mars, and Mary Lou Soffa. "Contentiousness vs. Sensitivity: Improving Contention Aware Runtime Systems on Multicore Architectures". In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 12–21. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000419. URL: http://doi.acm.org/10.1145/2000417.2000419 (visited on 05/27/2014).

[110] Valerie Taylor, Xingfu Wu, and Rick Stevens. "Design and implementation of prophesy automatic instrumentation and data entry system". In: *Proceedings of the Parallel and Distributed Computing and Systems Conference (PDCS)*. 2001.

[111] V. Taylor et al. "Prophesy: automating the modeling process". In: *Third Annual International Workshop on Active Middleware Services, 2001*. Third Annual International Workshop on Active Middleware Services, 2001. Aug. 2001, pp. 3–11. DOI: 10.1109/AMS.2001.993715.

[112] Traileka Glacier Team. *What Is OCR?* June 2014. URL: https://xstack.exascale-tech.com/wiki/images/d/d3/What-is-OCR.pptx.

[113] Ananta Tiwari et al. "Auto-tuning full applications: A case study". In: *Int. J. High Perform. Comput. Appl.* 25.3 (Aug. 2011), pp. 286–294. ISSN: 1094-3420. DOI: 10.1177/1094342011414744. URL: http://dx.doi.org/10.1177/1094342011414744.

[114] Antana Tiwari. "Tuning Parallel Applications in Parallel". PhD thesis. University of Maryland, College Park, 2011.

[115] *TOP500 List*. Nov. 2015. URL: http://www.top500.org/lists/2015/11/.

[116] UPC Consortium. *UPC Language and Library Specifications, v1.3*. Tech Report LBNL-59208. Lawrence Berkeley National Lab, 2013. URL: http://upc.lbl.gov/publications/upc-spec-1.3.pdf.

[117] Jeffrey Vetter and Chris Chambreau. "MPIp: Lightweight, scalable MPI profiling". In: (2005).

[118] Richard Vuduc, James W Demmel, and Katherine A Yelick. "OSKI: A library of automatically tuned sparse matrix kernels". In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[119] Michael Wagner, Tobias Hilbrich, and Holger Brunst. "Online Performance Analysis: An Event-Based Workflow Design towards Exascale". In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*. IEEE. 2014, pp. 839–846.

[120] Alexander Wert, Jens Happe, and Lucia Happe. "Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 552–561. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486861 (visited on 05/13/2014).

[121] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project". In: *Parallel Computing* 27 (2000), p. 2001.

[122] R. Clint Whaley and Jack J. Dongarra. "Automatically tuned linear algebra software". In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing '98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X. URL: http://dl.acm.org/citation.cfm?id=509058.509096.

[123] K.B. Wheeler, R.C. Murphy, and D. Thain. "Qthreads: An API for programming with millions of lightweight threads". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Apr. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536359.

[124] M. Wilde et al. "Swift: A Language for Distributed Parallel Scripting". In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[125] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[126] Felix Wolf et al. "Automatic Analysis of Inefficiency Patterns in Parallel Applications: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 19.11 (Aug. 2007), pp. 1481–1496. ISSN: 1532-0626. DOI: 10.1002/cpe.v19:11. URL: http://dx.doi.org/10.1002/cpe.v19:11 (visited on 05/01/2014).

[127] J. M. Wozniak et al. "Turbine: A Distributed-Memory Dataflow Engine for Extreme-Scale Many-Task Applications". In: *Proceedings SWEET 2012*. Scottsdale, AZ, May 2012. URL: https://sites.google.com/site/sweetworkshop2012/.

[128] J.M. Wozniak et al. "Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). May 2013, pp. 95–102. DOI: 10.1109/CCGrid.2013.99.

[129] Xingfu Wu et al. "Design and development of Prophesy Performance Database for distributed scientific applications". In: *10th SIAM Conference on Parallel Processing for Scientific Computing*. 2001.

[130] Asim YarKhan. "Dynamic task execution on shared and distributed memory architectures". PhD thesis. Knoxville, TN, USA: University of Tennessee, 2012. URL: http://trace.tennessee.edu/utk_graddiss/1575/.

[131] Katherine Yelick et al. "Productivity and performance using partitioned global address space languages". In: *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM. 2007, pp. 24–32.

[132] Kamen Yotov et al. "A Comparison of Empirical and Model-driven Optimization". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. New York, NY, USA: ACM, 2003, pp. 63–76. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781140. URL: http://doi.acm.org/10.1145/781131.781140 (visited on 05/01/2014).

[133] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228301 (visited on 06/17/2015).

[134] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10. URL: http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf (visited on 06/17/2015).

[135] Gengbin Zheng. "Achieving high performance on extremely large parallel machines: performance prediction and load balancing". PhD thesis. Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[136] Yili Zheng et al. “UPC++: A PGAS Extension for C++”. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. May 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115.