Cluster Comput (2010) 13: 243–256
DOI 10.1007/s10586-010-0134-7

Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions

Li Yu · Christopher Moretti · Andrew Thrasher · Scott Emrich · Kenneth Judd · Douglas Thain

Received: 9 November 2009 / Accepted: 16 March 2010 / Published online: 23 April 2010

Abstract Both distributed systems and multicore systems are difficult programming environments. Although the expert programmer may be able to carefully tune these systems to achieve high performance, the non-expert may struggle. We argue that high level abstractions are an effective way of making parallel computing accessible to the non-expert. An abstraction is a regularly structured framework into which a user may plug in simple sequential programs to create very large parallel programs. By virtue of a regular structure and declarative specification, abstractions may be materialized on distributed, multicore, and distributed multicore systems with robust performance across a wide range of problem sizes. In previous work, we presented the All-Pairs abstraction for computing on distributed systems of single CPUs. In this paper, we extend All-Pairs to multicore systems, and introduce the Wavefront and Makeflow abstractions, which represent a number of problems in economics and bioinformatics. We demonstrate good scaling of both abstractions up to 32 cores on one machine and hundreds of cores in a distributed system.

Keywords Abstractions · Multicore · Distributed systems · Bioinformatics · Economics

L. Yu (✉) · C. Moretti · A. Thrasher · S. Emrich · D. Thain
Department of Computer Science and Engineering, University of Notre Dame, South Bend, USA
e-mail: lyu2@nd.edu

K. Judd
Hoover Institution, Stanford University, Stanford, USA

1 Introduction

Distributed systems such as clusters, clouds, and grids are very challenging programming environments. (Hereafter, we refer to all of these systems as *clusters*.) A user that wishes to execute a large workload with some inherent parallelism is confronted with a dizzying array of choices. How should the workload be broken up into jobs? How should the data be distributed to each node? How many nodes should be used? Will the network be a bottleneck? Often, the answers to these questions depend heavily on the properties of the system and workload in use. Changing one parameter, such as the size of a file or the runtime of a job, may require a completely different strategy.

Multicore systems present many of the same challenges. The orders of magnitude change, but the questions are similar. How should work be divided among threads? Should we use message passing or shared memory? How many CPUs should be used? Will memory access present a bottleneck? When we consider clusters of multicore computers, then the problems become more complex.

We argue that *abstractions* are an effective way of enabling non-expert users to harness clusters, multicore computers, and clusters of multicore computers. An abstraction is a declarative structure that joins simple data structures and small sequential programs into parallel graphs that can be scaled to very large sizes. Because an abstraction is specialized to a restricted class of workloads, it is possible to create an efficient, robust, scalable, and fault tolerant implementation. In previous work, we introduced the All-Pairs [12] and Classify [13] abstractions, and described how they can be used to solve data intensive problems in the fields of biometrics, bioinformatics, and data mining. Our implementations allow non-experts to harness hundreds of processors on problems that run for hours or days using the Condor [27] distributed batch system.

In this paper, we extend the concept of abstractions to multicore computers and clusters of multicore computers. In Sect. 2, we present the concept of abstractions, and formally describe All-Pairs, Wavefront and Makeflow. In Sect. 3, we describe a general architecture for implementing abstractions on multicore clusters. In Sect. 4, we describe the technical challenges particular to All-Pairs, Wavefront, and Makeflow. In Sect. 5, we demonstrate weak scaling of each abstractions to large numbers of cores and nodes under controlled conditions. In Sect. 6, we discuss the advantages of a suite of specific abstractions. In Sect. 7, we demonstrate applications in bioinformatics and economics robustly running on hundreds of cores in an unreliable distributed system. We conclude with a review of related work and open avenues for research.

2 Abstractions

An *abstraction* is a declarative framework that joins together sequential processes and data structures into a regularly structured parallel graph. An abstraction *engine* is a particular implementation that materializes that abstraction on a system, whether it be a sequential computer, a multicore computer, or a distributed system. Figure 1 shows three examples of abstractions: All-Pairs, Wavefront and Makeflow.

All-Pairs (A[i], B[j], F(x, y))
returns matrix M
where M[i, j] = F(A[i], B[j])

The All-Pairs abstraction computes the Cartesian product of two sets, generating a matrix where each cell M[i, j] contains the output of the function F on objects A[i] and B[j]. This sort of problem is found in many different fields. In bioinformatics, one might compute All-Pairs on a set of gene sequences as the first step of building a phylogenetic tree. In biometrics, one might compute All-Pairs to determine the accuracy of a matching algorithm on a collection of faces. In data mining applications, one might compute All-Pairs on a set of documents to generate a graph of relationships.

Wavefront (R[i, j], F(x, y, d))
returns matrix R
where R[i, j] = F(R[$i-1$, j], R[i, $j-1$], R[$i-1$, $j-1$])

The Wavefront abstraction computes a recurrence relationship in two dimensions. Each cell in the output matrix is generated by a function F where the arguments are the values in the cells immediately to the left, below, and diagonally left and below. Once a value has been computed at position (1,1), then values at positions (2,1) and (1,2) may be computed, and so forth, until the entire matrix is complete. The problem can be generalized to an arbitrary number of dimensions. Wavefront represents a number of simulation problems in economics and game theory, where the initial states represent ending states of a game, and the recurrence is used to work backwards in order to discover the effect of decisions at each state. Wavefront also represents the problem of sequence alignment via dynamic programming in genomics.

Makeflow (R[n])
where each rule R[i] is:
input files : output files : command returns output files from all R[i]

The Makeflow abstraction expresses any arbitrary directed acyclic graph (DAG). Whereas All-Pairs and Wavefront are problems that can be decomposed into thousands or millions of instances of the same function to be run with near-identical requirements, a DAG workload may be structurally heterogeneous and consist of programs and files of highly variable runtime and size. Many such problems are

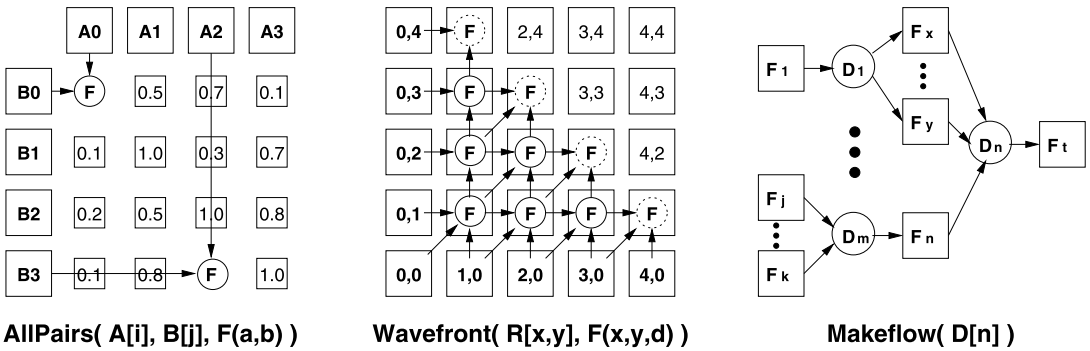


Fig. 1 Three examples of abstractions. All-Pairs, Wavefront and Makeflow are examples of abstractions. All-Pairs computes the Cartesian product of two sets A and B using a custom function F. Wavefront computes a two-dimensional recurrence relation using boundary conditions and a custom function F as an input. Makeflow takes an array of dependencies, which could be visualized as a directed acyclic graph structured workload, computes according to the workflow and produces a target file. Using different techniques, each can be executed efficiently on multicore clusters

found in bioinformatics, where users chain together multiple independent tools to solve a larger problem. Below, we will show Makeflow applied to a genomics problem.

On very small problems, these abstractions are easy to implement. For example, a small All-Pairs can be achieved by just iterating over the output matrix. However, many users have *very large* examples of these problems, which are not easy to implement. For example, a common All-Pairs problem in biometrics compares 4000 images of 1 MB to each other using a function that runs for one second, requiring 185 CPU-days of sequential computation. A sample Wavefront problem in economics requires evaluating a 500 by 500 matrix, where each function requires 7 s of computation, requiring 22 CPU-days of sequential computation. To solve these problems in reasonable time, we must harness hundreds of CPUs. However, scaling up to hundreds of CPUs forces us to confront these challenges:

- *Data Bottlenecks.* Often, I/O patterns that can be overlooked on one processor may be disastrous in a scalable system. One process loading one gigabyte from a local disk will be measured in seconds. But, hundreds of processes loading a gigabyte from a single disk over a shared network will encounter several different kinds of contention that do not scale linearly. An abstraction must take appropriate steps to carefully manage data transfer within the workload.
- *Latency vs Concurrency.* Dispatching sub-problems to a remote CPU can have a significant cost in a large distributed system. To overcome this cost, the system may increase the granularity of the sub-problems, but this decreases the available concurrency. To tune the system appropriately, the implementation must acquire knowledge of all the relevant factors.
- *Fault Tolerance.* The larger a system becomes, the higher the probability the user will encounter hardware failures, network partitions, adverse policy decisions, or unexpected slowdowns. To run robustly on hundreds of CPUs, our model must accept failures as a normal operating condition.
- *Ease of Use.* Most importantly, each of these problems must be addressed without placing additional burden on the end user. The system must operate robustly on problems ranging across several orders of magnitude by exploring, measuring, and adapting without assistance from the end user.

Examples of abstractions beyond the three mentioned above include Bag-of-Tasks [2, 24], Bulk Synchronous Parallel [3], and Map-Reduce [4]. None of these models is a universal programming language, but each is capable of representing a certain class of computations very efficiently. In that sense, programming abstractions are similar to the idea of systolic arrays [11], which are machines specialized for very specific, highly parallel tasks. Abstractions like All-Pairs and Wavefront are obviously than general purpose workflow languages such as DAGMan [27], Pegasus [5], Swift [30], and Dryad [10]. But, precisely because abstractions are regularly structured and less expressive, it is more tractable to provide robust and predictable implementations of large workloads. Once experience has been gained with specific abstractions, future work may evaluate whether more general languages can apply the same techniques.

3 Architecture

Figure 2 shows a general strategy for implementing abstractions on distributed multicore systems. The user invokes the abstraction by passing the input data and function to a *distributed master*. This process examines the size of the input data, the runtime of the function, consults a *resource*

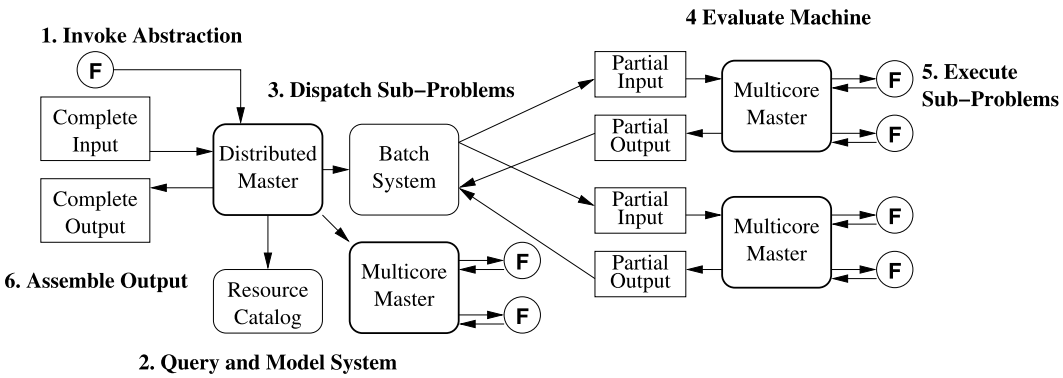


Fig. 2 Distributed multicore implementation. All-Pairs, Wavefront, and other abstractions can be executed on multicore clusters with a hierarchical technique. The user first invokes the abstraction, stating the input data sets and the desired function. The distributed master process measures the inputs, models the system, and submits sub-jobs to the distributed system. Each sub-job is executed by a multicore master, which dispatches functions, and returns results to the distributed master, which collects them in final form for the user

catalog to determine the available machines, and models the expected runtime of the workload in various configurations. After choosing a parallelization strategy, the distributed master submits sub-problems to the local *batch system*, which dispatches them to available CPUs. Each job consists of a *multicore master* which examines the executing machine, chooses a parallelization strategy, executes the sub-problem, and returns a partial result to the distributed master. As results are returned, the distributed master may dispatch more jobs and assembles the output into a compact final form.

For ease of use and implementation, both the distributed and multicore masters are contained in a single executable and invoked in the same way. Both All-Pairs and Wavefront are invoked by stating directories containing the input data and the name of the executable that implements the function:

```
allpairs function.exe Adir Bdir
wavefront function.exe Rdir
```

Without arguments, the distributed master will automatically choose how to partition the problem. When dispatching a sub-problem to a CPU, the distributed master simply invokes the same executable with options to select multicore mode on a given sub-problem, for example:

```
wavefront -M -X 15 -Y 20 -W 5 -H 5
function.exe Rdir
```

Of course, this assumes that the necessary files are available on the executing machine. The distributed master is responsible for setting this up via direct file transfer, or specification through the batch system. Note that this architecture allows for more than two levels of hierarchy—a global master could invoke distributed masters on multiple clusters—but we have not explored this idea yet.

The user may specify the *function* in several different ways. The function is usually a single executable program, in which case the input data is passed through files named on the command line, and the output is written to the standard output. This allows the end user to choose whatever programming language and environment they are most comfortable with, or even use an existing commercial binary. For example, the All-Pairs and Wavefront functions are invoked like this:

```
allpairs_func.exe Aitem Bitem > Output
wavefront_func.exe Xitem Yitem Ditem > Output
```

Invoking an external program might have unacceptable overhead if the execution time is relatively short. To overcome this, the user may also compile the function into a threaded shared library with interfaces like this:

```
void * allpairs_function(
const void *a, int alength,
const void *b, int blength );

void * wavefront_function(
const void *x, int xlength,
const void *y, int ylength,
const void *d, int dlength );
```

Regardless of how the code is provided, we use the term *function* in the logical sense: a discrete, self-contained piece of code with no side effects. This property is critical to achieving a robust, usable system. The distributed master relies on its knowledge of the function inputs to provide the necessary data to each node. If the function were to read or write unexpected data, the system would not function.

As the results are returned from each multicore master, the distributed master assembles them into a suitable external form. In the case of Wavefront, it is not realistic to leave each output in a separate file (although the batch system may deposit them that way), because the result would be millions of small files. Instead, the distributed master stores the results in an external sparse matrix. This provides efficient storage as well as checkpointing: after a crash, the master reads the matrix and continues where it left off.

The distributed master does not depend on the features of any particular batch system, apart from the ability to submit, track, and remove jobs. Our current implementation interfaces with both Condor [27] and Sun Grid Engine (SGE) [8], and expanding to other systems is straightforward. The distributed master also interfaces with a custom distributed system called Work Queue, which we will motivate and describe later.

To use Makeflow, a user needs to create a Makeflow script that describes the workflow of his workload. This language is very similar to traditional Make [31]: each rule states a program to run, along with the input files needed and the output files produced. Here is a very simple example:

```
part1 part2: input.data split.py
./split.py input.data

out1: part1 mysim.exe
./mysim.exe part1 >out1

out2: part2 mysim.exe
./mysim.exe part2 >out2
```

Like All-Pairs and Wavefront, Makeflow can run an entire workload on a local multicore machine, or submit jobs to Condor, SGE, or Work Queue. However, it does not have a hierarchical implementation: only single jobs are dispatched to remote machines. This is because graph partitioning is algorithmically complex, and impractical for heterogeneous workloads where runtime prediction is unreliable. Put simply, Makeflow has greater generality, but this comes at the cost of implementation efficiency, as we will emphasize below.

Table 1 Time to dispatch a task

Method	Time (μs)
pthread	6.3
fork	253
exec	830
popen	2500+
system	2500+

4 Building blocks

Our overall argument is that highly restricted abstractions are an effective way of constructing very large problems that are easily composed, robustly executed, and highly scalable. To evaluate this argument, we will begin by examining several questions about each abstraction at the level of microbenchmarks, then evaluate the system has a whole.

4.1 Threads and processes

It is often assumed that multicore machines should be programmed via multithreaded libraries or compilers. Our technique instead employs *processes*, because they are more easily adapted to distributed systems. How does this decision affect performance at the level of a single machine?

As a starting point, we constructed simple benchmarks to measure the time to dispatch a null task using various techniques. Each measurement is repeated one thousand times, and the average is shown. (Unless otherwise noted, the benchmark machine is a 1 GHz dual core AMD Opteron model 1210 with 2 GB RAM running Linux 2.6.9.) Table 1 shows the results. `pthread` creates and joins a standard POSIX thread on an empty function, `fork` creates and works for a process which simply calls `exit`, `exec` forks and executes an external program, and `popen` and `system` create new sub-processes invoked through the shell.

It is no surprise that creating a thread is several orders of magnitude faster than creating a process. However, it is not so obvious that `popen` and `system` are considerably more expensive than `exec`, and often vary in cost from user to user. This is because these methods invoke the user's shell along with their complex startup scripts, which can have unbounded execution time and create troubleshooting problems. If we are careful to avoid these methods, then executing an external program can be made reasonably fast. Moreover, it is only necessary for the execution time to dominate the invocation time: a task in an abstraction running for a second or more is sufficient.

4.2 Concurrency and data in All-Pairs

Of course, within a real program, we must weigh invocation time against more complex issues such as synchronization, caching, and access to data. To explore the boundaries of these issues, we studied the All-Pairs multicore master running in sequential mode on a single machine, comparing 1 MB randomly generated files. A simple comparison function counts the number of bytes different in each object. From a systems perspective, this is similar to a biometrics problem, and provides a high ratio of data to computation. Any realistic comparison function would be more CPU intensive, so these tests explore the worst case.

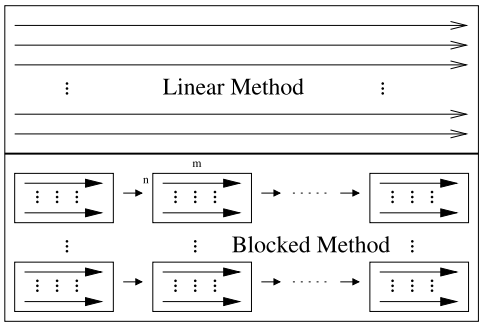


Fig. 3 Linear method vs blocked method. The linear method evaluates cells in the matrix line by line. The blocked method evaluates cells block by block with a width chosen to fit in the file system buffer cache

In this scenario, we vary several factors. First, we vary the invocation method of the function: create a thread to run an internal function (*thread*) or create a process to execute an external program (*process*). The author of a function is free to choose their own I/O technique, so we also compare buffered I/O byte-by-byte (*fgetc*), block-by-block (*fread*), and memory-mapped I/O (*mmap*). A naive implementation would simply iterate over the output matrix in order, causing cache misses at all levels on every access. A more effective method shown in Fig. 3 is to choose a smaller block of cells and iterate over those completely before proceeding to the next block. The width of the block is called the *block size*. (This technique is sufficient for our purposes, but see Frigo et al. [7] for more clever methods.)

Figure 4 shows the relative weight of all these issues. Each curve shows the runtime of a 1000×10 comparison over various block sizes. The two slowest curves are *thread* and *process*, both using *fgetc*. The two middle curves are *process* using *fread* and *mmap*, and the fastest is *thread* with *mmap*. All curves show significant slowdown when the block size exceeds physical memory.

Clearly, threads with *mmap* execute twice as fast as the next best configuration. If the user is willing to write a thread-safe function for use with the abstraction, they should do so. However, the use of processes is only twice as slow *in this artificial worst case* and will not fare as poorly with a more CPU-intensive function. Moreover, the appropriate use of virtual memory by the abstraction and the I/O technique chosen by the function are *much more significant factors* than the difference between threads and processes. We

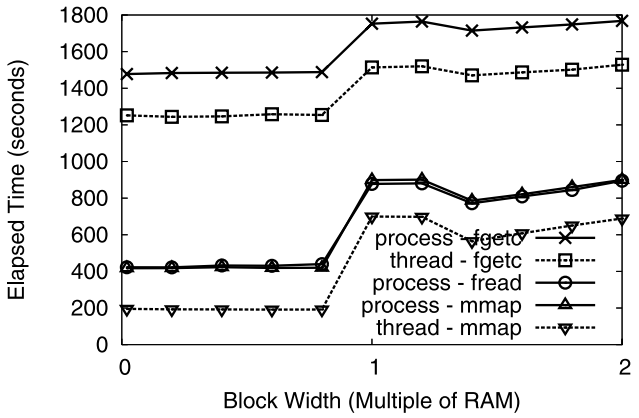


Fig. 4 Threads, processes, and I/O techniques. The performance of a data intensive 1000 × 10 All-Pairs in sequential mode using threads and processes with various I/O techniques. While threads provide the best performance, processes are a reasonable method even on this worst case

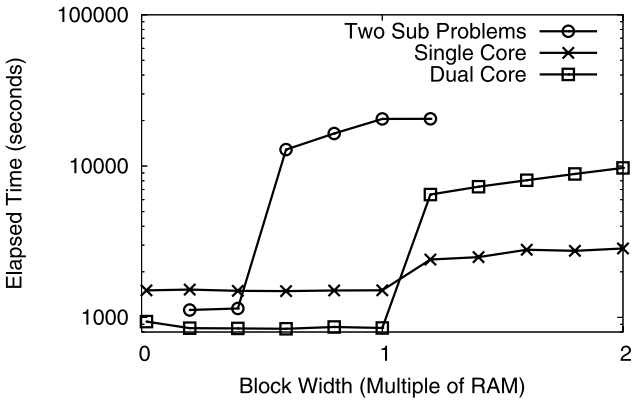


Fig. 5 Multicore vs sub-problems. The performance of an 1000 × 10 All-Pairs in sequential mode, in dual-core mode, and as two independent sequential sub-problems, using various block sizes. This demonstrates the importance of an explicit multicore strategy

conclude that using processes to exploit parallelism is a reasonable tradeoff if it improves the usability of the system.

(*We re-emphasize that each abstraction can accept either an external program or a threaded internal function. So far, none of our users has chosen to use threads.*)

Next we consider how to carry out All-Pairs on a multicore machine. Although there are many possible ways, we may consider two basic strategies. One is to generate N contiguous sub-problems, and allow each core to run independently. The other is to write an explicit multicore master that proceeds through the entire problem coherently, dispatching individual functions to each core. Figure 5 compares both of these against a simple sequential approach. As can be seen, the sub-problem approach performs far worse, because it does not coordinate access to data, and caches at all levels are quickly overwhelmed. Thus, we have shown it is necessary to have a deliberate multicore implementation, rather than treating each core as a separate node.

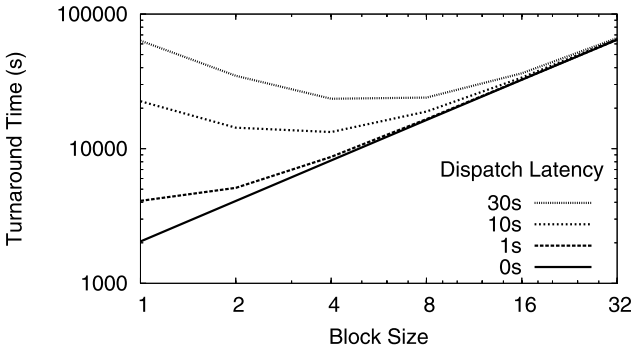


Fig. 6 The effect of latency on wavefront. The modeled runtime of a 1000 × 1000 Wavefront where each function takes one second to complete, with varying block size and dispatch latency. As dispatch time increases, the system must increase block size to overcome the idle time

4.3 Control flow in Wavefront

As we have shown, the primary problem in efficient All-Pairs is managing data access. However, in Wavefront the problem is almost entirely control flow. The first task of the problem is sequential. Once completed, two tasks may run in parallel, then three, and so forth. If there is any delay in dispatching or completing a task, this will have a cascading effect on dependent adjacent tasks. We will consider two control flow problems: dispatch latency and run-time variance.

Figure 6 models the effect of latency on a Wavefront problem. This simple model assumes a 1000 × 1000 problem where each task takes one second to complete. On the *x* axis, *block size* indicates the size of sub problem dispatched to a processor. Each curve shows the runtime achieved for a system with dispatch latency ranging from zero (e.g. a multicore machine) to 30 s (e.g. a wide area computing grid).

As block size increases, the sub-problem runtime increases relative to the dispatch latency, but less parallelism is available because the distributed master must wait for an entire sub-problem to complete before dispatching its neighbors. The result is that for very high dispatch times, a modest block size improves performance, but cannot compete with a system that has lower dispatch latency. So, the key to the problem is to minimize dispatch latency.

Although Wavefront can submit jobs to Condor and SGE batch systems directly, the dispatch latency of these systems when idle is anywhere from ten to sixty seconds, depending on the local configuration. For short-running functions, this will not result in acceptable performance, even if we choose a large block size. (This is not an implementation error in ei-

ther system, rather it is a natural result of the need to service many different users within complex policy constraints.)

To address this, we borrowed the idea of a *fast dispatch execution system* as in Falkon [17]. We built a simple system called Work Queue that uses lightweight worker processes that can be submitted to a batch system. Each contacts the distributed master, and provides the ability to upload and execute files. This allows for task dispatch times measured in milliseconds instead of seconds. Workers may be added or removed from the system at any time, and the master will compensate by assigning new tasks, or reassigning failed tasks.

However, even if we solve the problem of fixed dispatch latency, we must still deal with the unexpected delays that occur in distributed systems. When Work Queue runs on a Condor pool, a running task may still be arbitrarily delayed in execution. It may be evicted by system policy, stalled due to competition for local resources, or simply caught on a very slow machine. To address these problems, the Work Queue scheduler keeps statistics on the average execution time of successful jobs and the success rate of individual workers. It makes assignments preferentially to machines with the fastest history, and proactively aborts and re-assigns tasks that have run longer than three standard deviations past the average. These techniques are collectively called Fast Abort.

Figure 7 shows the impact of Fast Abort on starting up a 1000×1000 Wavefront on 180 CPUs. Without Fast Abort, stuck jobs cause the workload to however around twenty tasks running at once. With Fast Abort, the stragglers are systematically resolved and the concurrency increases linearly until all CPUs are in use. Figure 8 shows this behavior from another perspective. The distributed master periodically produces a bitmap showing the progress of the run. Colors indicate the state of each cell: red is incomplete, green is running, and blue is complete. Due to the heterogeneity of the underlying machines, the wave proceeds irregularly. Although an $N \times N$ problem should use N CPUs at maximum, this perfect diagonal is rarely seen.

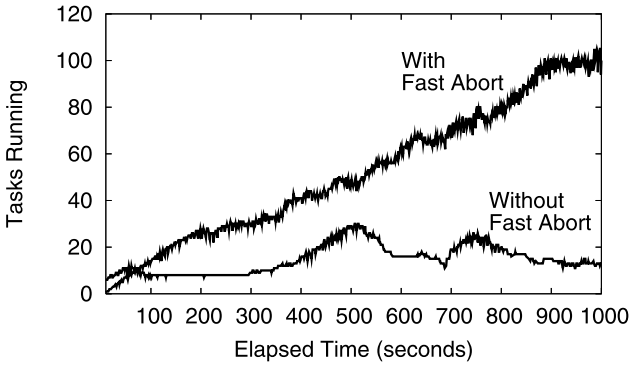


Fig. 7 The effect of fast abort on wavefront. The startup behavior of a 500×500 Wavefront with and without Fast Abort. Without Fast Abort, every delayed result impedes the increase in parallelism, which stabilizes around 20. With Fast Abort, delays are avoided and parallelism increases steadily

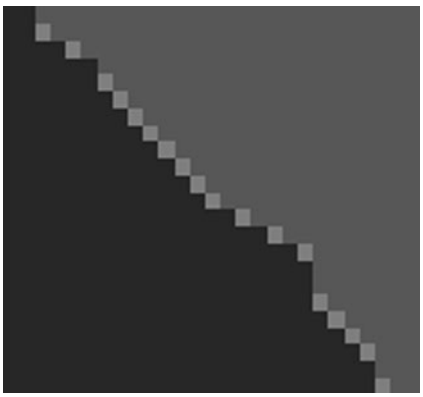


Fig. 8 Asynchronous progress in wavefront. A progress display from a Wavefront problem. Each cell shows the current state of a portion of the computation: the *darkest gray* in the lower left corner indicates incomplete, the *lighter gray* in the upper right indicates complete, and the *light cells* in between are currently running. The irregular progress is due to heterogeneity and asynchrony in the system

4.4 Greater generality with Makeflow

Makeflow provides a different type of building block for large multicore workflows with abstractions. Makeflow combines many functions together (instead of many instances of the same function) to express more complex series of operations.

Makeflow uses a syntax very similar to traditional Make, but it differs in one critical way: each rule of a Makeflow must exactly state *all of the files* consumed or created by the rule. (In traditional Make, one can often omit files, or add dummy rules as needed to affect the control flow.) Makeflow is more strict, but this allows it to accurately generate batch jobs, exploit common patterns of work, and schedule jobs to where their data is located. This allows Makeflow to run correctly on both local multicore machines as well as a distributed system.

The Makeflow abstraction can be configured to use different numbers of cores. Figure 9 shows the turnaround times varying the number of cores used with two different options for executing a genomics workload on 1–24 cores. The top curve ("cluster") presents Makeflow using Work Queue, with workers submitted to remote machines as Condor jobs. The bottom curve ("multicore") executes all work as Makeflow-controlled local processes, in which Makeflow automatically takes advantage of multiple cores on the submitting machine. Makeflow jobs running locally outperform jobs tasked to remote workers and scale well up to the number of available cores.

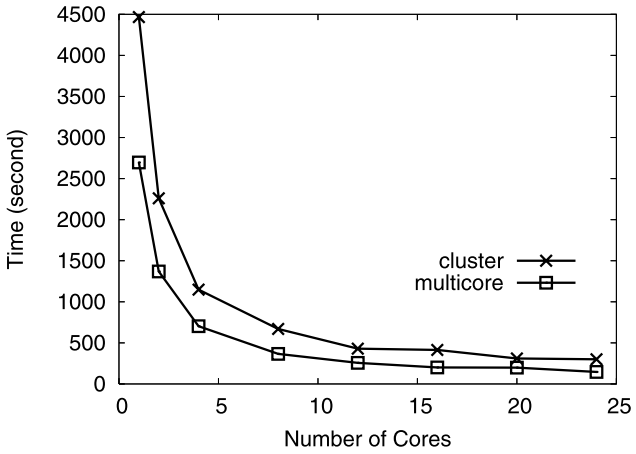


Fig. 9 Makeflow on multicore and cluster. The performance of a genomics application run through Makeflow, using 1–24 cores on both a multicore machine and a cluster using Work Queue

5 Putting it all together

In a well-defined dedicated environment in which the distributed master knows exactly which resources will be used, a model can partition work to the resources in such a way as to optimize the workload [28]. This applies to multicore environments as well—the distributed master could build multicore assumptions into the model to optimize a workload. However, this finely-tuned partitioning does not adapt well to heterogeneous environments or resource unavailability. Previous work derived a more realistic solution for modeling the turnaround time of an All-Pairs workload in a cluster [12]. Is it possible to use the multicore version of the All-Pairs abstraction transparently beneath the cluster abstraction?

If the abstraction is to use the multicore master transparently, then it must continue to exclude considerations of the number of cores per node from the model. If the workload is benchmarked on a single-core system or with a single-threaded executor, then the model will choose appropriate resources to run the workload efficiently assuming single-threaded operation. Adding multicore execution to this workload, then, will only serve to make the batch jobs complete faster on the multicore resources. It does not change the overall workload any more than having benchmarked on a slow node would: the success of the model in avoiding disastrous cases is maintained, the faster resources (in this case multicore nodes) will account for a greater portion of the batch jobs than their "fair share", and any long-tail from slow nodes would extend out at most to the same duration as without any multicore nodes.

So it is possible, but this is little solace if there is a clearly better solution for modeling a distributed All-Pairs workload using multicore resources. Another option is to integrate the multicore master (instead of the original single-threaded executor) into the benchmarking process for the model. If the function runtime is benchmarked using the multicore master, then the function execution time (computed as the average time per function over a small set of executions) will be comparable to the expected execution of batch jobs on the same number of cores. This is a good approach for submitting to homogeneous clusters of resources in which the same number of cores are available for every batch job. In a heterogeneous environment, however, this only serves to exacerbate the model's assumption that the benchmark node reflects the cluster's resources. Whereas the original model conceded that individual resources might be perhaps a generation newer (faster) or older (slower) than the benchmark node, the inclusion of multicore uncertainty into the benchmarking increases the potential range of resource capabilities and thus the potential for long-tail effects in a workload.

Another option would be to include a coefficient of the average number of cores within the model. Because the model includes a component for the time to complete a single batch job, an adjustment for the number of cores could be made by dividing the batch job execution time in the model by this average. This retains the same prerequisite measurements (plus the calculation of the average number of cores), however it has several limitations. First, the pool of resources must be well-defined so that the average number of cores may be determined; but because the model is used to select the appropriate number of resources, the exact set of hosts is not known a priori. Thus, the average number of cores available for each host is a pool average rather than one specific to the actual resources used. Further, contention for resources means that not all hosts will be utilized equally or predictably, which presents the same problem in trying to include a factor of the number of cores in the turnaround time model. This is especially problematic as we move beyond workstations with at most a few cores: unavailability of a machine with dozens of cores significantly changes the average number of cores of the available machines.

With that said, can we accurately model the performance of our abstractions? Figure 10 shows the modeled performance of All-Pairs workloads of varying sizes running on an 8-core machine and a 64-core cluster. Figure 11 shows the modeled performance of Wavefront workloads running on a 32-core machine and a 180-core cluster. In both cases, the multicore model is highly accurate, due to a lack of competing users and other complications of distributed systems. Both models are sufficiently accurate that we may use them to choose the appropriate implementation at runtime based on the properties given to the abstraction. Figure 12 shows the modeled performance of Makeflow workloads running on a 24-core machine and a 60-core cluster. Figure 13 compares the multicore and cluster models for the previous All-Pairs and Wavefront examples, and demonstrates the actual

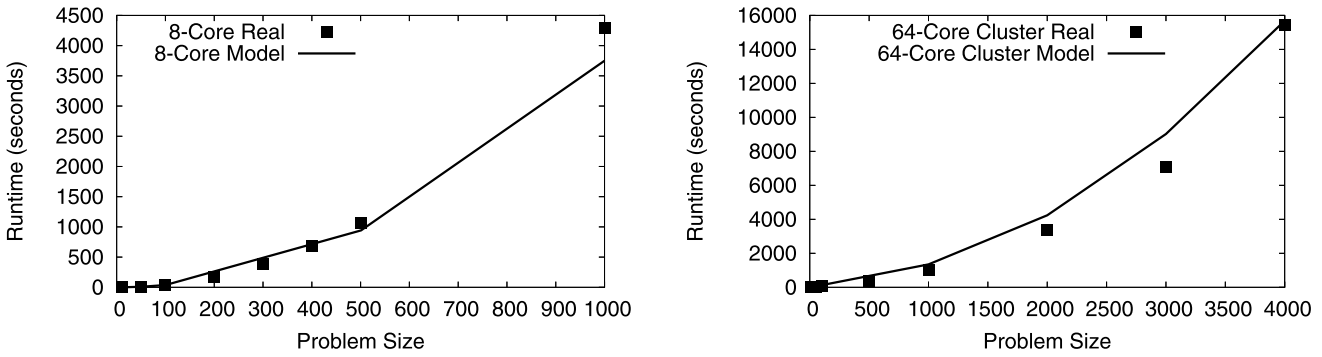


Fig. 10 Accuracy of the all-pairs model on multicore and cluster. The real and modeled performance of an All-Pairs benchmark of varying sizes on a 8-core machine (*left*) and an 64-core cluster (*right*)

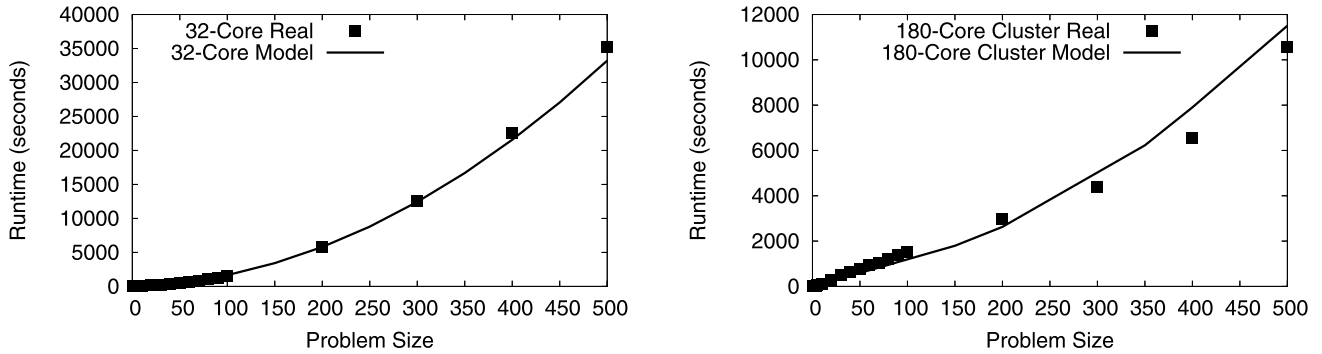


Fig. 11 Accuracy of the wavefront model on multicore and cluster. The real and modeled performance of a Wavefront benchmark of varying sizes on a 32-core machine (*left*) and an 180-core cluster (*right*)

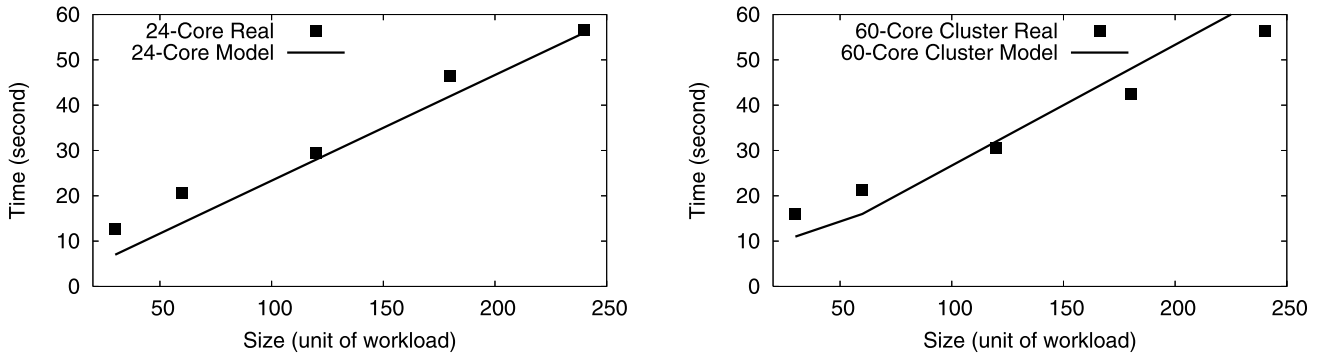


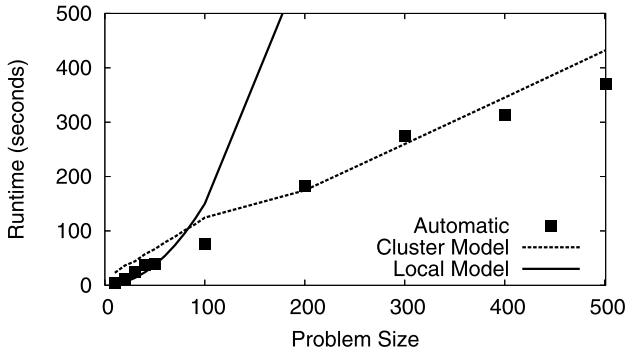
Fig. 12 Accuracy of the makeflow model on multicore and cluster. The real and modeled performance of a Makeflow benchmark of varying sizes on a 24-core machine (*left*) and a 60-core cluster (*right*)

performance achieved when selecting the implementation at runtime.

6 Why multiple abstractions?

With the Makeflow abstraction for arbitrary DAG workflows, could we choose to use it as a general tool instead of implementations of the specific abstractions mentioned above? In our experience, the answer is that we could, but in doing so we lose many of the problem-specific advantages given by the less general abstractions. We carry out All-Pairs on a 24-core machine using both the all-pairs multicore abstraction and the Makeflow abstraction.

We vary the size of the workloads from creating a 10×10 matrix to creating a 1000×1000 matrix. Each matrix cell is computed by comparing two 20 KB files. With the Make-



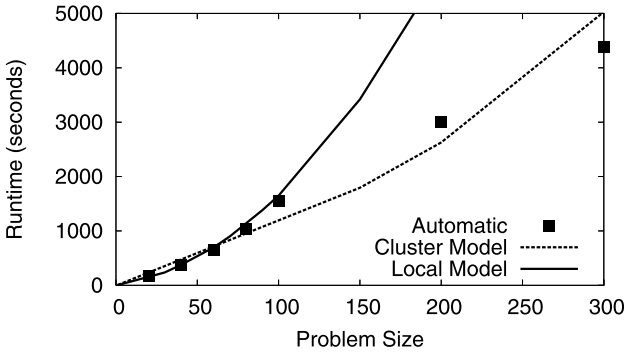


Fig. 13 Selecting an implementation based on the model. These graphs overlay the modeled multicore and cluster performance on problems of various sizes for All-Pairs (*left*) and Wavefront (*right*). The *dots* indicate actual performance for the selected problem size. As can be seen, the modeled performance is not perfect, but it is sufficient to choose the right implementation

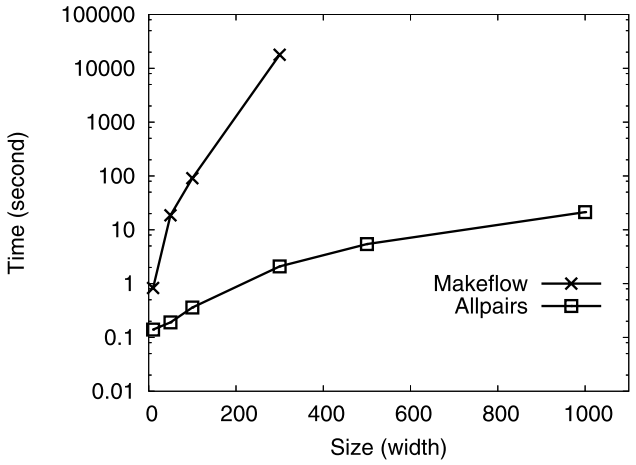


Fig. 14 Solving All-Pairs with Makeflow and All-Pairs. This figure shows the time to complete an All-Pairs problem of various sizes using the general Makeflow tool and the specific All-Pairs tool. The more general tool is considerably more expensive, because it uses files for output storage, and is unable to dispatch sub-problems to multicore processors

flow abstraction, each cell value depends on a comparison and the cell value is stored in a file after it is computed. And we have to write an additional program, which depends on all the cell value files, to extract all cell values from generated files and put them into the target matrix. The running time of both abstractions on different workloads are shown in Fig. 14. It is easy to see that the All-Pairs multicore abstraction scales almost linearly as the workload increases. However, the Makeflow abstraction is several orders of magnitude slower at this problem, because it uses files for output storage, and is unable to manage work in organized blocks.

The increased generality of Makeflow has a significant price, so we conclude that there is a great benefit to retaining specific abstractions such as All-Pairs and Wavefront for specialized problems.

7 Applications

In a previous paper [12], we demonstrated a number of applications of All-Pairs, so here we will focus on applications of Wavefront and Makeflow.

Our example applications run on an open Condor pool of approximately 600 CPUs, consisting of heterogeneous desktop machines, homogeneous research clusters, and multicore servers ranging from 4–32 CPUs. Clock speeds range from 500 MHz to 4 GHz. The number of CPUs changes unpredictably as desktop users come and go, and other workloads enter and leave the system. Thus, these results show that our system achieves reasonable performance under adverse conditions on real applications.

We use the term *speedup* with the usual definition: sequential runtime divided by actual runtime. However, note that the maximum possible speedup is *not* the number of processors. Consider an idealized machine with P processors running an $N \times N$ problem with $P \geq N$. In the first timestep, one task will run, then two, and so forth, until the diagonal is reached in N steps. Then, $N-1$ tasks run simultaneously and so on down to one task, giving a parallel runtime of $(2N-1)$ timesteps. The sequential runtime is N^2, so the best possible speedup is $N^2/(2N-1)$ or $N/2$. But, if $P \ll N$, then the system will quickly reach a steady state of P tasks running, and the runtime will approach N^2/P with a speedup of P. We will see both cases below.

7.1 Bioinformatics

Sequence alignment is one of the most important tasks in bioinformatics and is used in a variety of applications. Common variants of pairwise sequence alignment can be solved using dynamic programming [14] and each requires time proportional to the product of the two sequences considered. Prior parallel implementations have been motivated by either the need to compare a single pair of large sequences [18]

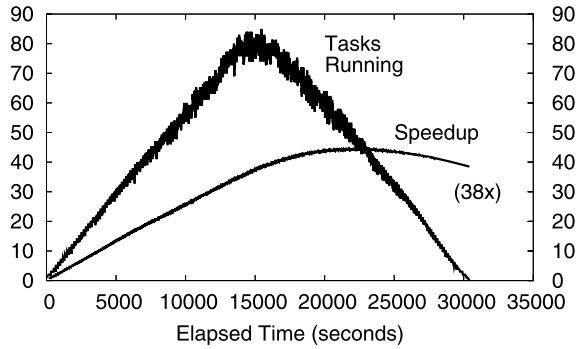


Fig. 15 100 × 100 Wavefront in bioinformatics. A timeline of a 100 × 100 Wavefront problem implementing sequence alignment running on non-dedicated multicore Condor pool. 80 cores were available at the peak of the execution. An overall speedup of 38x is achieved, the maximum possible is 50x

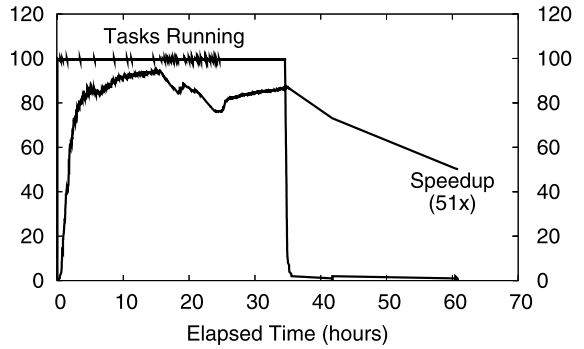


Fig. 16 Makeflow without Fast Abort. A timeline of SSAHA execution on 100 simultaneous workers without Fast Abort. As can be seen, the long tail is almost as long as the peak computation period

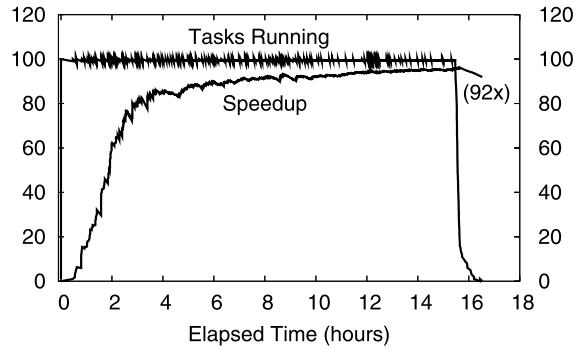


Fig. 17 Makeflow with Fast Abort. A timeline of SSAHA execution on 100 simultaneous workers with Fast Abort. Compared to the above figure, the tail is mostly eliminated

or the need to compare many small sequences [15] for tasks such as phylogenetic inference and genome assembly. Previous algorithms have implemented the wavefront problem on dedicated clusters and parallel architectures such as the Cell [23]. Our implementation achieves similar speedups, but requires only sequential coding, and can execute on unreliable, loosely coupled machines.

In less than a day, we wrote a single process function in 156 lines of C++ that performed alignment on a substring and propagated the required data for later steps. Distributed sequence alignment was then tested on two large bacteria genomes using wavefront: a non-virulent lab strain of Anthrax (*Bacillus anthracis* str. Ames; Genbank NC_003997) and its virulent ancestor strain (*Bacillus anthracis* str. 'Ames Ancestor'; Genbank NC_007530). Each genome is approximately 5.3 million characters long, and the score of an optimal suffix-prefix alignment was computed using only linear-space. An actual alignment (i.e., the path through the dynamic programming matrix) is also attainable based on the divide-and-conquer Hirschberg technique [23], which requires twice as much computation and a more complicated strategy.

Figure 15 shows a timeline of this alignment running using a 100 × 100 partition of the problem. Each task takes about 117 s to run on a 1 GHz CPU. On the Condor pool, a maximum of 80 tasks running simultaneously was achieved. The overall runtime was reduced from 13 days sequential to 8.3 hours with a speedup of 38x out of the maximum possible 50x.

We also explored the application of a heuristic for bioinformatics problems similar to sequence alignment. SSAHA (Sequence Search and Alignment by Hashing Algorithm) [29] is a bioinformatics tool designed to map one set of genetic data onto another set of data. SSAHA is very similar to the popular bioinformatics tool BLAST [1] because it creates a hash table for a set of subject sequences to speed up the search of query sequences for matches. Unlike BLAST, SSAHA computes the complete mapping and therefore can be used to discover detailed differences between sequences and individuals [29]. SSAHA is a publicly available sequential application. Our implementation involves running the sequential application many times in parallel using the Makeflow and Work Queue abstractions. This allows us to harness the Condor pool to complete our computation in a reasonable time.

Our implementation mapped 11.5 million sequences consisting of 11 billion bases onto the genome *Sorghum bicolor* [16] (738.5 million bases). This is a large bioinformatics workload with the majority of execution time for each job dedicated to mapping the queries and a small portion dedicated to generating hash tables. The abstraction split a large sequential execution into nearly 2300 smaller sequential computations that were run in parallel on workers submitted to our Condor pool. Figure 16 shows the execution of this job on a maximum of 100 simultaneous workers without Fast Abort. There is an extremely prominent long-tail effect that nearly doubles the total execution time. Figure 17 shows the same workload run with fast abort enabled, which nearly eliminated the long-tail effect and more than halved our to-

tal run time. The implementation using Fast Abort required 16 hours of runtime compared to the sequential runtime of 65 days with a total speedup of 92x.

7.2 Economics

The wavefront abstraction can represent a number of dynamic economic problems. Consider, for example, the competition between two microprocessor vendors. Each firm produces microprocessors and engages in R&D to improve the clock speed. That game ends when they reach limits imposed by physics. Economic models examining such dynamic games would discretize the problem by assuming that there are N possible efficiencies and each firm begins with efficiency level 1. The state of a two-player game is denoted by the vector of efficiencies (i, j). At each such state, each firm competes for sales of the chips of those efficiencies but each firm also wants to improve its efficiency. When the game reaches the state (N, N) the dynamics are done and we have reached a static situation which can be computed directly. If the state of the game is $(N-1, N)$ then firm 1 still works to improve its efficiency and its incentives to work on R&D are affected by the anticipated profits it receives when the game goes to (N, N). This is also true for player 2 in the state $(N, N-1)$. Hence, the solution at (N, N) allows us to solve $(N-1, N)$ and $(N, N-1)$. Similarly, those solutions allow us to solve $(N-2, N)$, $(N-1, N-1)$ and $(N, N-2)$. The wavefront abstraction sweeps through the states until we have solved the dynamic game at all states (i, j), $1 \leq i, j \leq N$.

This kind of game arises in many dynamic economic problems. See [9, 25, 26] for original papers on the learning curve, [6, 19–21] for examples of dynamic R&D races, and [22] for an example from the exhaustible resources literature. All of these results are limited in scope because a sequential implementation dramatically limits the number of parameters. For example, the learning and R&D papers assume only two firms and a small number of steps. This is an unreasonable assumption since there are many firms in each industry, particularly at the early stages where innovation is rapid and many firms are competing to be one of the few survivors. These models are essential for a serious examination of antitrust policies that limit how fiercely firms may compete and tax policies that are supposedly designed to encourage innovation

Using the wavefront abstraction, we can easily carry out problems many orders of magnitude larger than have been attempted before. With less than a day of coding, we ported a Nash equilibrium function for two players with four parameters from Mathematica into a 77-line C program usable with Wavefront. On a single input, this function requires about 7.6 s to complete on a 1 GHz CPU.

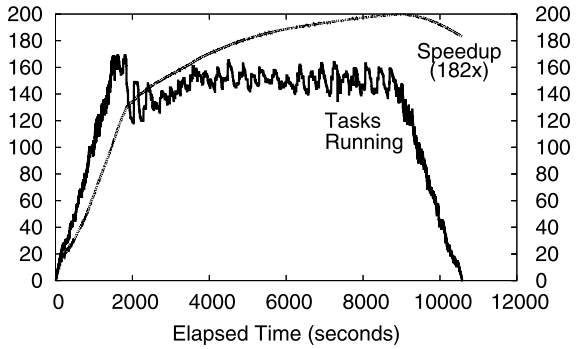


Fig. 18 500×500 Wavefront in economics. A timeline of a 500×500 Wavefront problem in economics running on non-dedicated multicore Condor pool. Because many of the remote CPUs were faster than the submitting CPU, the overall speedup of 180x is greater than the number of CPUs

Figure 18 shows a timeline of this workload running on the Condor pool. The workload quickly reached the maximum available parallelism of between 120 and 160 CPUs. An overall speedup of 182x was achieved, reducing the sequential runtime from 22 days to 2.9 hours. The speedup achieved was faster than ideal because many of the remote CPUs were faster than the submitting machine on which the function was benchmarked.

8 Conclusion

We have demonstrated how simple high level abstractions can be used to scale regularly structured problems up to clusters of multicore computers. We have made the following key observations:

- Processes are a realistic alternative to threads for programming multicore systems, even on I/O intensive tasks.
- It is feasible to accurately model the performance of large scale abstractions across a wide range of configurations, allowing for the rational selection of appropriate resources.
- Abstractions are easy for non-experts to program, provided there is a good match between the application structure and the application.
- The All-Pairs and Wavefront abstractions can be scaled up to hundreds of cores, achieving good performance even under adverse conditions.
- General abstractions, like Makeflow, are able to deal with more kinds of application structures; however, they might not achieve the same performance as specific abstractions.

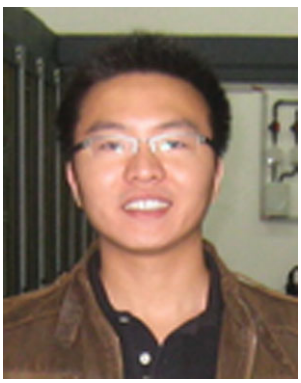
There are many avenues of future work. We have outlined a two-level hierarchy of implementations for abstractions, but the system could be generalized to support solving very large problems across the wide area with deeper

nesting. Additional implementations of abstractions on specialized architectures such as the Cell or FPGAs might be effective ways of transparently adding such devices to large computations.

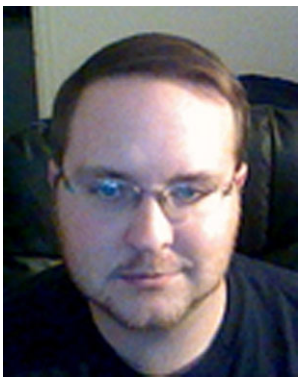
Acknowledgements This work was supported in part by the National Science Foundation under grants CCF-06-43229 and CNS-06-21434, and the National Institutes of Health NIAID contract HHSN272200900039C.

References

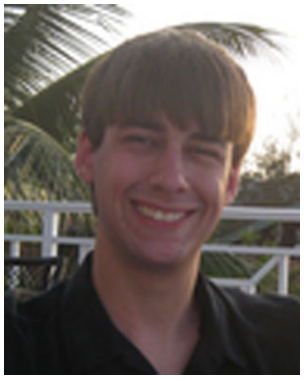
1. Altschul, S.F., et al.: Basic local alignment search tool. J. Mol. Biol. **3**, 403–410 (1990)
2. Bakken, D., Schlichting, R.: Tolerating failures in the bag-of-tasks programming paradigm. In: IEEE International Symposium on Fault Tolerant Computing, June 1991
3. Cheatham, T., Fahmy, A., Siefanescu, D., Valiani, L.: Bulk synchronous parallel computing—a paradigm for transportable software. In: Hawaii International Conference on Systems Sciences (2005)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large cluster. In: Operating Systems Design and Implementation (2004)
5. Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, B., Good, J., Laity, A., Jacob, J., Katz, D.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci. Program. J. **13**(3), 219–237 (2005)
6. Doraszelski, U.: An R&D race with knowledge accumulation. Bell J. Econ. **34**, 19–41 (2003)
7. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache oblivious algorithms. In: Foundations of Computer Science (FOCS) (1999)
8. Gentzsch, W.: Sun grid engine: towards creating a compute power grid. In: CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid (2001)
9. Ghemawat, P., Spence, A.M.: Learning curve spillovers and market performance. Q. J. Econ. **100**, 839–852 (1985)
10. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data parallel programs from sequential building blocks. In: Proceedings of EuroSys, March 2007
11. Kung, H.T.: Why systolic architectures? IEEE Comput. **15**, 37–46 (1982)
12. Moretti, C., Bulosan, J., Flynn, P., Thain, D.: All-pairs: An abstraction for data intensive cloud computing. In: International Parallel and Distributed Processing Symposium (IPDPS) (2008)
13. Moretti, C., Steinhaeuser, K., Thain, D., Chawla, N.V.: Scaling up classifiers to cloud computers. In: International Conference on Data Mining (ICDM) (2008)
14. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in amino acid sequence of two proteins. J. Mol. Biol. **48**, 443–453 (1970)
15. Oliver, T., Schmidt, B., Nathan, D., Clemens, R., Maskell, D.: Using reconfigurable hardware to accelerate multiple sequence alignment with clustalw. Bioinformatics **21**, 3431–3432 (2005)
16. Paterson, A.H., et al.: The *Sorghum bicolor* genome and the diversification of grasses. Nature **457**, 551–556 (2009)
17. Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M.: Falkon: a fast and light-weight task execution framework. In: IEEE/ACM Supercomputing (2007)
18. Rajko, S., Aluru, S.: Space and time optimal parallel sequence alignments. IEEE Trans. Parallel Distrib. Syst. **15**(12), 1070–1081 (2004)
19. Reinganum, J.: Dynamic games of innovation. J. Econ. Theory **25**, 21–41 (1981)
20. Reinganum, J.: A dynamic game of R&D: patent protection and competitive behavior. Econometrica **50**, 671–688 (1982)
21. Reinganum, J.: Corrigendum. J. Econ. Theory **35**, 196–197 (1985)
22. Reinganum, J., Stokey, N.: Oligopoly extraction of a common property natural resource: the importance of the period of commitment in dynamic games. Int. Econ. Rev. **26**, 161–174 (1985)
23. Sarje, A., Aluru, S.: Parallel biological sequence alignments on the cell broadband engine. In: International Parallel and Distributed Processing Symposium (IPDPS) (2008)
24. da Silva, D., Cirne, W., Brasilero, F.: Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In: Euro-Par (2003)
25. Spence, A.M.: The learning curve and competition. Bell J. Econ. **12**, 49–70 (1981)
26. Spence, A.M.: Cost reduction, competition, and industry performance. Econometrica **52**, 101–121 (1984)
27. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In: Berman, F., Fox, G., Hey, T. (eds.) Grid Computing: Making the Global Infrastructure a Reality. Wiley, New York (2003)
28. Theobald, K.B., Gao, G.R.: An efficient parallel algorithm for all pairs examination. In: Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp. 742–753. ACM, New York (1991)
29. Ning, Z., Cox, A.J., Mullikin, J.C.: SSAHA: a fast search method for large DNA databases. Genome Res. **10**, 1725–1729 (2001)
30. Wilde, M.: Parallel scripting for applications at the petascale. IEEE Computer, November 2009
31. Feldman, S.: Make—a program for maintaining computer programs. Softw. Pract. Exp. **9**, 255–265 (1978)



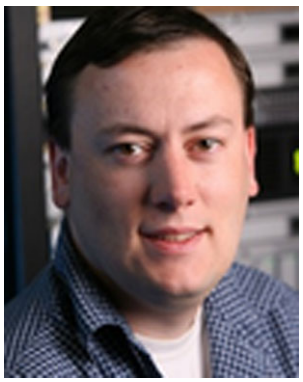
Li Yu graduated from Huazhong University of Science and Technology in 2006 with a B.S. in Computer Science and in 2008 with a M.S. in Computer Science. He is currently a Ph.D. student at the University of Notre Dame, researching distributed computing.



Christopher Moretti graduated from the College of William and Mary in 2004 with a B.S. in Computer Science. He received a Master of Science in Computer Science and Engineering from the University of Notre Dame in 2007. He is currently a Ph.D. candidate at Notre Dame, researching distributed computing abstractions for scientific computing.



Andrew Thrasher graduated from Anderson University in Indiana in 2009 with a B.A. in Computer Science, Mathematics and Physics. He is currently working towards a Master's degree in Computer Science at the University of Notre Dame.



Scott Emrich received the B.S. in Biology and Computer Science from Loyola College in Maryland and the Ph.D. in Bioinformatics and Computational Biology from Iowa State University. His research interests include computational biology, bioinformatics and parallel computing, including arthropod genome analysis with applications to global health and ecology.

Kenneth Judd is the Paul H. Bauer Senior Fellow at the Hoover Institution at Stanford University. He graduated Phi Beta Kappa from the University of Wisconsin (1975) with undergraduate degrees in mathematics and computer sciences. Judd received an M.A. from the University of Wisconsin in mathematics in 1977 and an M.A. in economics in 1980. He was awarded a Ph.D. in economics in 1981 from the University of Wisconsin. His current research focuses on tax policy and antitrust issues, as well as developing computational methods for economic modeling.

Douglas Thain received the B.S. in Physics in 1997 from the University of Minnesota and the M.S. and Ph.D. in Computer Sciences in 1999 and 2004 from the University of Wisconsin. He is currently an Assistant Professor of Computer Science and Engineering at the University of Notre Dame, where his research focuses on scientific applications of distributed computing systems.