Parallel Kernels: An Architecture for Distributed Parallel Computing

P. A. Kienzle (pkienzle@nist.gov) – *NIST Center for Neutron Research, National Institute of Standards and Technology, Gaithersburg, Maryland 20899* USA
N. Patel (npatel17@umd.edu) – *Department of Materials Science and Engineering, University of Maryland, College Park, Maryland 20742* USA
M. McKerns (mmckerns@caltech.edu) – *Materials Science, California Institute of Technology, Pasadena, California 91125* USA

Global optimization problems can involve huge computational resources. The need to prepare, schedule and monitor hundreds of runs and interactively explore and analyze data is a challenging problem. Managing such a complex computational environment requires a sophisticated software framework which can distribute the computation on remote nodes hiding the complexity of the communication in such a way that scientist can concentrate on the details of computation. We present PARK, the computational job management framework being developed as a part of DANSE project, which will offer a simple, efficient and consistent user experience in a variety of heterogeneous environments from multicore workstations to global Grid systems. PARK will provide a single environment for developing and testing algorithms locally and executing them on remote clusters, while providing user full access to their job history including their configuration and input/output. This paper will introduce the PARK philosophy, the PARK architecture and current and future strategy in the context of global optimization algorithms.

Introduction

In this paper we present PARK, a flexible job management and parallel computation framework which is being developed as a part of DANSE project [Ful09]. PARK is a high level tool written in Python to provide the necessary tools for parallel and distributed computing [Bal89]. The heart of the system is yet another parallel mapping kernel, hiding the details of communication and freeing the end-user scientist to concentrate on the computational algorithm. The success of environments like Matlab and NumPy show that using an abstract interface using high level primitives such as vector mathematics, slow interpreted languages can achieve high performance on numerical codes. For parallel computations, Google introduced the Map-Reduce algorithm [Dea04], a robust implementation of master-slave parallelism where nodes could enter into and out of the computation. Individual map-reduce programs do not have to deal with the complexities of managing a reliable cluster environment, but can achieve fast robust performance on distributed database applications. The powerful map construct can be used equally well in many scientific computing problems.

A number of Python packages are addressing parts of this problem. PaPy[Cie09] is based on map constructs and directed acyclic graphs (DAGs) for scalable workflows to process data. Parallel Python[Von09] allows users to run functions remotely or locally. They provide code movement facilities to ease the installation and maintenance of parallel systems. IPython Parallel[Per09] gives users direct control of remote nodes through an interactive python console. They provide a load balancing map capability as well as supporting direct communication between nodes. Pyro [Jon09] is a python remote object implementation with a name service to find remote processes. It includes a publish-subscribe messaging system. PARK uses ideas from all of these systems.

Concept and architecture

PARK is a user-friendly job management tool that allows easy interaction with heterogeneous computational environments.
PARK uses a front end for submitting complex computational jobs to variety of computing environments. The backend computational service can use a dedicated cluster, or a shared cluster controlled by a batch queue. The front end allows the user to submit, kill, resubmit, copy and delete jobs. Like PaPy, jobs will be able to be composed as workflows. Unlike traditional batch processing systems, applications can maintain contact with the running service, receiving notification of progress and sending signals to change the flow of the computation. Communication from services to the client is handled through a publish-subscribe event queue, allowing multiple clients to monitor the same job. The client can signal changes to running jobs by adding messages to their own queues. Clients can choose to monitor the job continuously or to poll status periodically. The emphasis of the system is on early feedback and high throughput rather than maximum processing speed of any one job.
Remote jobs run as service components. A service component contains the full description of a computational task, including code to execute, input data for processing, environment set-up specification, post-processing tasks, output data produced by the application and meta-data for bookkeeping. The purpose of PARK can then be seen as making it easy for end-user scientist to create, submit and monitor the progress of services. PARK keeps a record of the services created

and submitted by the user in a persistent job repository.

The PARK API is designed for both the needs of interactive Graphical User Interfaces (GUI) and for scripted or Command Line Interfaces (CLI). A service in PARK is constructed from a set of components. All services are required to have an application component and a backend component, which define respectively the software to be run and the computational resources to be used. A client can connect to multiple backends at the same time, one of which will be the client machine. Services have input and output data components which are used during service execution. The overall PARK architecture is illustrated in figure 1. PARK monitors the evolution of submitted services and categorizes them into submitted, waiting, scheduled, running, completed, failed or killed states. All service objects are stored in a job repository database and the input and output files associated with the services are stored in file workspace. Both job repository and the file workspace may be in a local file system or on a remote server.

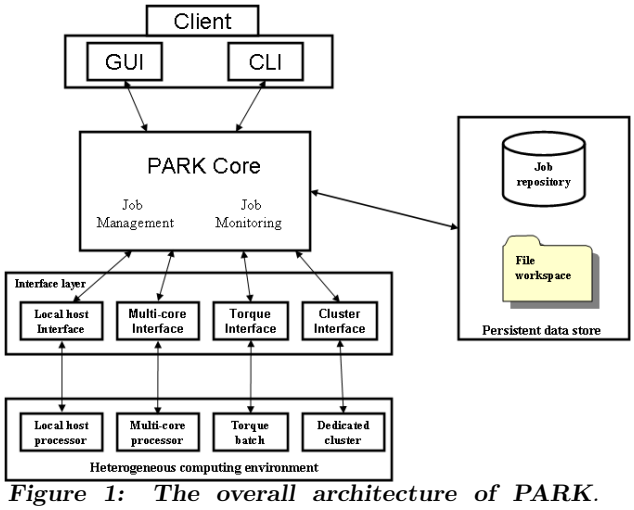

Figure 1: The overall architecture of PARK. *The client interacts with PARK via the Graphical User Interface (GUI) or via Command Line Interface (CLI). All jobs are stored in the persistent data store.*

PARK interface

The client initiates the connection to PARK through a connect object, with the URL of the resource as a parameter. Different resources may use different connection and authentication and transport protocols, but the returned connection object provides a unified interface. The connection may be to the local machine or a remote cluster. Client can submit their jobs through this connect object to respective computational resources and also retrieve the results of jobs. PARK provides functions for executing jobs, retrieving the results of finished jobs, checking the status of the jobs and controlling the jobs. The functions provided by PARK are independent of various computational resources so that a client may write a single program that will run on any platform with minor changes in interface.

The connect function creates a new connection object associated with specified computational resources. If no connection is established, then the job will be run on the client machine. The example below connects to the cluster at compufans.ncnr.nist.gov:

```
compufans = park.connect('compufans.ncnr.nist.gov')
```

The submit function creates a job and submits it to the compufans job queue. This function returns immediately. It returns job id back to the client with which client can control the job. The executable can be a script or a function. If executable is a function, the modules argument specify what modules are required to execute the function:

```
job_id = compufans.submit(executable,
                  input_data=None, modules=())
```

The get_result_id function returns the result of the submitted job after it is completed, or None if it is not completed:

```
result = compufans.get_result_id(job_id=None)
```

The get_status function returns a python dictionary with keys specifying the current status of all the three queues (waiting, running and finished) and values as number of jobs in each queues:

```
status_dict = compufans.get_status(tag)
```

Job management architecture

The job management components include job creator, job scheduler and job launcher. Its architecture is shown on figure 2.

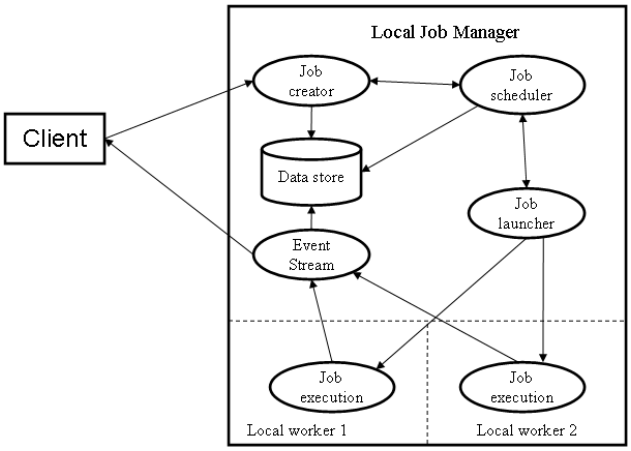

Figure 2: Job management architecture. *In this case client connects to a local system and PARK automatically identifies the number of cores present in the system and spawns that many workers to run job. The job communicates back to the client through a publish-subscribe event queue.*

The job creator receives information on the jobs from the client either through GUI or through CLI. The

job creator assigns an job identifier to every job and returns this to the client. The combination of job manager URL and job id is unique.

The job scheduler is the core scheduling component which decides which jobs to run when. The scheduler is responsible for maintaining enough information about the state of the jobs to make good decisions about job placement. All the resources are allocated to the jobs by job priority. This ensures that high-priority jobs are added at the front of the queue. If jobs have equal priority, resources are allocated to the job that was submitted first. The job scheduler also supports backfill. This ensures that a resource-intensive application will not delay other applications that are ready to run. The job scheduler will schedule a lower-priority job if a higher-priority job is waiting for resources to become available and the lower-priority job can be finished with the available resources without delaying the start time of the higher-priority job.

The job launcher's function is to accept the jobs and launch them. Job launcher contains the logic required to initiate execution on the selected platform for the selected network configuration. The job launcher receives messages sent by the job scheduler. When receiving an execution request, it will create a new thread in order to allow asynchronous job execution. After getting a job identifier (i.e. the handle to the job at the server side) as the response, the job launcher will send it to the corresponding thread of the job scheduler. The job handle will be used to monitor the job status and control the job.

The job monitor is used to monitor the submitted jobs. Information about the remote execution site, queue status and successful termination are collected. The job manager maintains a connection to the running job so that the client can steer the job if necessary. For example, a fitting service may allow the user to change the range of values for a fit parameter. This level of control will not be possible on some architectures, such as TeraGRID nodes that do not allow connections back to the event server.

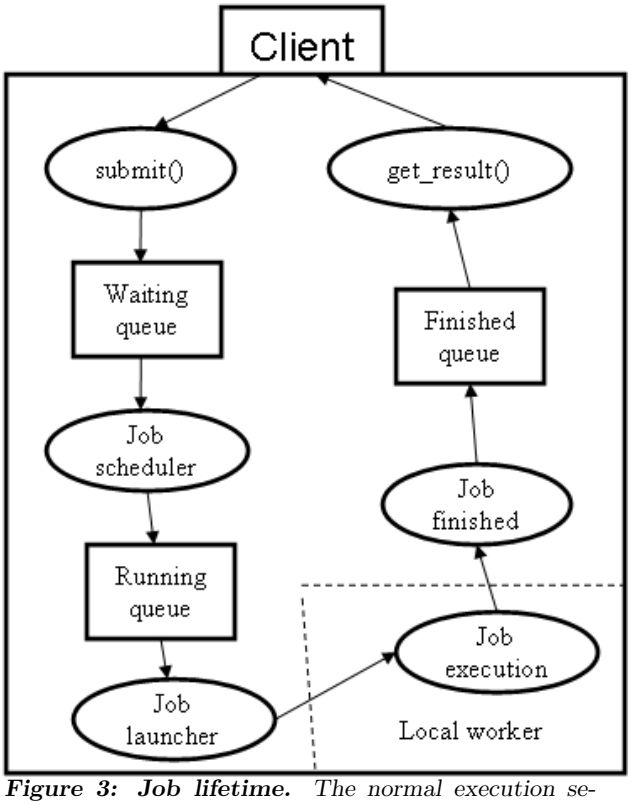

Figure 3: Job lifetime. *The normal execution sequence complete separates the client from the computational resources, allowing users to disconnect when the job is started and reconnect to get the results.*

Job execution workflow

In order to simplify the complexity of job management, various operations of job management are organized as a workflow. Within the workflow the job can be deleted or modified at any time. PARK manages three job queues namely the waiting, running and finished queues and jobs are stored in one of the three queues based on their status. The waiting queue keeps the un-scheduled jobs. The running queue keeps running jobs and the finished queue keeps finished jobs which are either completed successfully or failed. After the job is created, it is put on the waiting queue. When all dependencies are met the job description is submitted to the job launcher and the job is moved to the running queue. On job completion, a completion event is published on the job event queue. The job monitor subscribes to this queue, and moves the job to the finished queue when it receives the completion event. After the job reaches the finished queue, the results are ready to be retrieved by the client (see figure 3). All the logging and error messages are automatically saved. At the client side, multiple jobs can be defined at the same time, so there may exist more than one job workflow in the system. All jobs are independent of each other and identified by either job ID or client tags. Future versions will support workflow conditions, only launching a new job when the dependent jobs are complete.

Parallel mapper

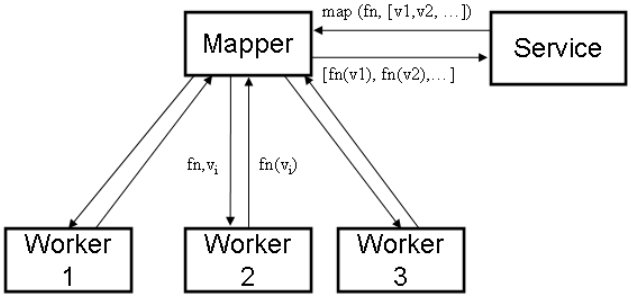

Figure 4: Mapper workflow architecture. *fn is the function to be mapped and v_i are the inputs. The function should be stateless, with output depending only on the current input v_i, not on the previous inputs.*

A running service has many facilities available. It can use a data store or file system for data retrieval. A service can use Mapper which applies a function to a set of inputs. Mapper acts like the built-in 'map' function of Python, but the work is completed in parallel across the available computing nodes. Mapper keeps tracks of all map requests from various running service and handles any issues like node failure or nested maps and infinite loops in a functions. Mapper can be excellent resource for data parallelism where same code runs concurrently on different data elements. Mapper workflow is described in figure 4, in which running service sends map request to Mapper via service handler. Mapper queues individual function-value pairs. Workers pull from this queue, returning results to Mapper. After all results are complete, Mapper orders them and sends them back to the service.

Global optimization

We are using PARK to construct a global optimization framework. The basic algorithm for optimization is as follows:

```
optimizer.configure()
population = optimizer.start(function.parameters)
while True:
    cost = map(function, population)
    optimizer.update(population)
    if converged():
        break
    post_progress_event()
    population = optimizer.next()
```

This algorithm applies equally to pure stochastic optimizers like differential evolution and genetic algorithms as well as deterministic methods such as branch and bound, and even quasi-Newton methods where multiple evaluations are required to estimate the descent direction.

In order to implement this algorithm in PARK, we need to define a Fit service which accepts an optimizer, some convergence conditions and the function to be optimized. The Fit service first registers the function with Mapper. Workers who participate in the map evaluation are first configured with the function, and then called repeatedly with members of the population. Since the configuration cost is incurred once per worker, we can handle problems with a significant configuration cost. For example a fit to a 100Mb dataset requires the file to be transferred to each worker node once at the beginning rather than each time the fitness function is evaluated. This mechanism also supports a dynamic worker pool, since new workers can ask Mapper for the function when they join the pool.

We have generalized convergence conditions. Rather than building convergence into each optimizer, the Fit service keeps track of the values of the population, the best fit, the number of iterations, the number of function calls and similar parameters that are used to control the optimizer. To write a new optimizer for PARK, users will need to subclass from Optimizer as follows:

```
class Optimizer:
    def configure(self):
        "prepare the optimizer"
    def start(self, parameters):
        "generate the initial population"
    def update(self, population):
        "record results of last population"
    def next(self):
        "generate the next population"
```

Each cycle the optimizer has the option of informing the client of the progress of the fit. If the fit value has improved or if a given percentage of the maximum number of iterations is reached then a message will be posted to the event stream associated with the fit job. If the cluster cannot connect to the event queue (which will be the case when running on TeraGRID machines), or if the client is not connected then this event will be ignored.

The assumption we are making is that the cost of function evaluations is large compared to the cost of generating the next population and sending it to the workers. These assumptions are not implicit to the global optimization problem; in some cases the cost of evaluating the population members is cheap compared to processing the population. In a dedicated cluster we can hide this problem by allowing several optimization problems to be run at the same time on the same workers, thus when one optimizer is busy generating the next population, the workers can be evaluating the populations from the remaining optimizers. A solution which can exploit all available parallelism will require dedicated code for each optimizer and more communication mechanisms than PARK currently provides.

Conclusions and future work

A robust job management and optimization system is essential for harnessing computational potential of various heterogeneous computing resources. In this paper we presented an overview of the architecture of PARK distributed parallel framework and its various features which makes running single job across various heterogeneous platform almost painless.

The PARK framework and the global optimizer are still under active development and not yet ready for general use. We are preparing for an initial public release in the next year.

In future we would like to implement new scheduling scheme for optimizing resource sharing. We will add tools to ease the monitoring of jobs and administering a cluster. We are planning various forms of fault tolerance features to PARK, making it robust against failure of any nodes, including the master node via checkpoint and restore. Security is also an important issue, further work being needed to improve the existing security features. Using the high level map primitive, ambitious users can implement PARK applications on new architectures such as cloud computing, BOINC[And04], and TeraGrid by specializing the map primitive for their system. The application code will remain the same.

Acknowledgments

This work is supported by National Science Foundation as a part of the DANSE project, under grant DMR-0520547.

References

[And04] D. P. Anderson. *BOINC: A system for public-resource computing and storage*, in R. Buyya, editor, 5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings, pages 4-10. IEEE Computer Society, 2004.

[Bal89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. *Programming languages for distributed computing systems* ACM computing Surveys, 21(3):261-322, September 1989.

[Cie09] M. Cieslik, *PaPy: Parallel and distributed data-processing pipelines in Python*, in Proceedings of the 8th Python in Science Conference (SciPy 2009)

[Dea04] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[Ful09] B. Fultz, et al.: *DANSE: Distributed data analysis for neutron scattering experiments*, http://danse.us/ (Accessed Aug. 2009)

[Jon09] I. de Jong: *Pyro - Python Remote Objects*, http://pyro.sourceforge.net/ (Accessed Aug. 2009)

[Per09] F. Perez and B. Granger: *IPython: a system for interactive scientific computing*, Computing in Science & Engineering 9(3) 21-29, 2007

[Van09] V. Vanovschi: *Parallel Python*, http://www.parallelpython.com/ (Accessed Aug. 2009)

