Towards Jungle Computing with Ibis/Constellation

Jason Maassen, Niels Drost, Henri E. Bal, Frank J. Seinstra
Department of Computer Science, VU University
Amsterdam, The Netherlands
{jason,niels,bal,fjseins}@cs.vu.nl

ABSTRACT

The scientific computing landscape is becoming more and more complex. Besides traditional supercomputers and clusters, scientists can also apply grid and cloud infrastructures. Moreover, the current integration of many-core technologies such as GPUs with such infrastructures adds to the complexity. To make matters worse, data distribution, hardware availability, software heterogeneity, and increasing data sizes, commonly force scientists to use *multiple* computing platforms *simultaneously*: a true *computing jungle*.

In this paper we introduce Ibis/Constellation, a software platform specifically designed for distributed, heterogeneous and hierarchical computing environments. In Ibis/Constellation we assume that applications consist of several distinct (but somehow related) activities. These activities can be implemented independently using existing, well understood tools (e.g. MPI, CUDA, etc.). Ibis/Constellation is then used to construct the overall application by coupling the distinct activities. Using *application defined labels* in combination with *context-aware work stealing*, Ibis/Constellation provides a simple and efficient mechanism for automatically mapping the activities to the appropriate resources, taking data locality and heterogeneity into account.

We show that an existing supernova detection application can be ported to Ibis/Constellation with little effort. By making small changes to the application defined labels, this example application can run efficiently in three very different HPC computing environments: a distributed set of clusters, a large 48-core machine, and a GPU cluster.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent Programming—*Distributed Programming, Parallel Programming*

General Terms

Design, Experimentation, Performance

3DAPAS'11, June 8, 2011, San Jose, California, USA.

1. INTRODUCTION

High-performance computing (HPC) is used in more and more disciplines. Although HPC has long been a trusted workhorse for physicists and chemists, many other scientists are only just discovering its potential. Recently, HPC has been applied to a wide range of domains, such as multimedia analysis [22], remote sensing [16], medical image processing [11], and semantic web reasoning [26].

Unfortunately, the computing landscape is becoming more complex. Next to traditional supercomputers and clusters, scientists can also apply grid and cloud infrastructures. The current integration of many-core technologies (e.g., GPUs [5]) with these infrastructures adds to the complexity.

While each of these systems in itself is reasonably straightforward to program (by using platform specific tools), creating applications that run on a combination of such systems is notoriously difficult. Traditional tightly-coupled HPC tools, such as MPI [10], are not particularly suited for distributed, heterogeneous and hierarchical environments. To make matters worse, hardware availability, data distribution, software heterogeneity, and the need for scalability, commonly force scientists to use *multiple* computing platforms *simultaneously*: a true *computing jungle* [23].

In this paper we introduce Ibis/Constellation (from here on referred to as Constellation), a lightweight software platform that is specifically designed to support such *Jungle Computing*. Constellation aims to efficiently run applications on complex combinations of distributed, heterogeneous and hierarchical computing hardware. In addition, Constellation makes the re-targeting of applications to completely different computing environments very straightforward.

The programming paradigm offered by Constellation is similar to that of scientific workflows [7] and many task computing [19] (MTC). Constellation assumes that applications consists of multiple *distinct activities* with certain dependencies between them. These activities can be implemented independently using the tools, and targeted at the (HPC) architecture, that suit them best. Multiple implementations of an activity may be created to support different hardware architectures or problem instances. Existing legacy codes can also be used.

This approach to application development vastly reduces the programming complexity. Instead of having to create a single application capable of running in a complex distributed and heterogeneous environment, it is sufficient to create (or reuse) several independent activities targeted at smaller and simpler homogeneous environments. Traditional

HPC tools and libraries such as MPI [10] or CUDA [5] can be used to create each of the separate activities.

Constellation offers a simple API to define activities and the dependencies between them. In functionality it is similar to a many task computing platform, which are often regarded as HPC backends for workflow systems [15].

The focus of our current work is on creating a Constellation run time system (RTS) that can efficiently execute applications on a wide range of distributed, heterogeneous and hierarchical systems, and combinations of such systems. Unlike other many task computing platforms, Constellation offers a simple mechanism to express the heterogeneity present in such applications and hardware.

Using application defined *labels*, Constellation allows an application to tag different types of activities and hardware. Using these labels, one can express that certain activities need to be run in a specific *context*. For example, an activity may need to be run in a specific location due to data dependencies, or require a specific type of hardware (e.g. a GPU). Using *context-aware work stealing* (a simple and efficient matchmaking and load balancing mechanism), the Constellation RTS ensures that each activity is executed in a suitable computing environment.

Constellation is developed as part of the Ibis project [2]. The goal of Constellation is to provide a basic platform that allows us to experiment with Jungle Computing, i.e, applications that run in complex distributed, heterogeneous and hierarchical environments. In addition, we aim to use Constellation as a basis for further development of a wide range of high-level programming models (e.g., workflow, domain-specific languages, divide-and-conquer).

In this paper we present the first results of our efforts. We will first take a closer look at the Constellation API and RTS. Next, we will show how an existing supernova detection application can be ported to Constellation with relatively little effort. By simply re-labeling the activities of this application, it can be run efficiently in three very different hardware environments: a distributed set of clusters, a single 48-core machine, and a GPU cluster.

2. RELATED WORK

Constellation is closely related to workflow systems [7] and many-task computing [19] (MTC). Although the distinction between these two concepts is vague, Ogasawara et al [15] argues that workflow systems are often more focused on giving support in workflow design, reuse, version control and provenance, while many-task computing systems specialized in parallelizing workflow activities in large HPC environments. As such, many task computing platforms can be regarded as HPC backends for workflow systems. We regard Constellation as a many task computing platform.

Research in the MTC field has mainly focused on very large scale data intensive applications executing on very large systems [20]. However, we believe that this model is equally suitable for Jungle Computing. Jungle Computing was first introduced in Seinstra et al. [23] where it is defined as *simultaneously* using a combination of *heterogeneous*, *hierarchical*, and *distributed* computing resources.

MUSE [17] is an example of a Jungle Computing system. MUSE is a framework for large-scale simulations of dense stellar systems. It couples existing codes for dynamics, stellar evolution, and hydrodynamics. These codes require a variety of hardware ranging from traditional clusters and supercomputers to GPU and GRAPE [14] machines. Unlike Constellation, MUSE is designed for a single domain. Its successor, AMUSE[1], is currently under active development.

The DIRAC [25] high-throughput system can run applications in a distributed and heterogeneous environment. It assumes that each task in the application is run on a single worker node. It uses a single centralized queue and heavyweight *matchmaking*. Constellation also allows parallel tasks, supports multiple queues in different locations, and uses very simple and efficient matchmaking mechanism.

Matchmaking [21], is one of the main problems in heterogeneous applications, i.e., getting the right activities to the right resource. Although matchmaking is also performed by traditional schedulers [3, 9, 24, 32], they are generally optimized for long running tasks. As shown in Raicu et al. [18], they therefore lack performance when scheduling the many small tasks often present in MTC applications. Falkon [18] has therefore deliberately dropped matchmaking in favor of faster task scheduling. A multi-level *glide-in* approach is used, as introduced in Condor [24]. With glide-in, the batch queues are only used to start lightweight processes on the resources. These processes are then used to directly start application tasks on these resources, thereby completely circumventing the batch queues and significantly improving both throughput and scalability.

Constellation uses a similar glide-in approach. Unlike Falkon, however, Constellation still supports matchmaking, as this is essential when running in heterogeneous environments. Unlike traditional schedulers, the matchmaking in Constellation is intentionally kept simple to improve the efficiency. An additional advantage of the glide-in approach is that it allows us to combine multiple unrelated resources into a single compute pool for our applications. A similar approach is used by Condor [24] and Walker et al. [30].

Pegasus [6] has been designed to map scientific workflows onto grid systems. Pegasus does not uses a glide-in approach. Instead, tasks are submitted to the queuing systems of the grid site that are used. Therefore, the applications must be relatively course-grained to run efficiently. Similarly, Nephele [31] maps scientific workflows onto clouds, exploiting their dynamic nature to optimize for cost. Constellation currently does not take cost into account.

StarPU [1] is a runtime system designed for heterogeneous multi core architectures. Unlike Constellation, however, it is designed to schedule heterogeneous tasks within a single system (e.g., a multi core machine with a GPU).

In the Ibis project [2], we have investigated many of the problems that arise when developing HPC applications for distributed environments. Ibis offers a rich software stack that provides various libraries for application deployment [8, 27], support for communication in restricted networks [13] and many programming models [4,28,29]. Constellation uses many of the software components developed in Ibis, but is the first programming model in Ibis specifically designed to run Jungle Computing applications.

3. CONSTELLATION

Constellation is a lightweight platform that is specifically designed for distributed, heterogeneous and hierarchical computing environments. In the following sections, we will give an overview of the programming model offered by Constella-

[1] http://amusecode.org

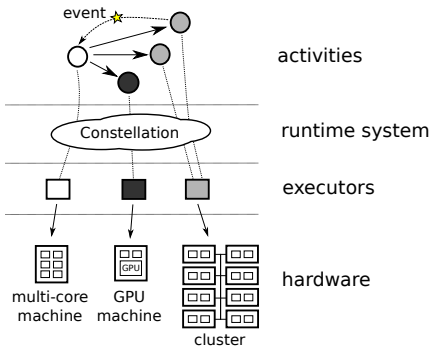

Figure 1: Example of a Constellation application.

tion. In addition, we briefly look at the programming API, and describe the implementation of matchmaking and load balancing mechanism based on context-aware work stealing.

3.1 Overview

In Constellation, a program consists of a set of loosely coupled *Activities* that communicate using *Events*. The complexity of a program may vary from a simple bag-of-tasks to a complex workflow comprised of multiple interdependent activities. Figure 1 (top) shows a simple example using four activities. There, an initial activity (white) is started that submits three additional activities (gray and black).

Each activity represents a distinct action that needs to be performed by the application, e.g., process a piece of data or run a simulation. As such, an activity usually represents a combination of code, parameters, and data. Each activity may consist of a script, sequential C, CUDA, a parallel application using OpenMP or MPI, etc.

Constellation uses *Executors* to represent hardware capable of running activities. An executor may represent a single core of a machine, a single machine with multiple cores, a specialized piece of hardware (e.g., a GPU), an entire cluster, etc. The application is free to determine how the executors should represent the hardware.

Constellation assumes a *glide-in* [18, 24, 30] approach is used to start the executors on the available hardware, e.g. by using IbisDeploy [2] or Zorilla [8]. In this paper we will not describe how the necessary resources are obtained. Instead, we assume that a heterogeneous set of resources capable of running the necessary executors is available.

In Figure 1 (bottom), three executors are shown, one representing a multi-core machine, one representing a machine containing a GPU, and one representing a small cluster of 8 machines. Obviously, when such an heterogeneous set of executors is used, not every activity will be able to run on every executor. An activity consisting of GPU code will not run on an MPI cluster, nor will an MPI application run on a GPU. Therefore, it is essential for running the application that the activities end up on a suitable executor. For this purpose, Constellation uses the concept of *context*.

A context is an application defined label (or set of labels) that can be attached to both activities and executors. A label can describe data dependencies (`"dataset X"`), hardware capabilities (`"GPU"`), or problem and resource sizes (`"large"`). When an activity and executor use the same label, they *match*, i.e., the executor is assumed to offer the right context for running the activity. Constellation plays the role of matchmaker to ensure that each activity is forwarded to a suitable executor. In Figure 1 this is illustrated by the different shades used for the activities and executors.

Although the concept of context matching is similar to the resource requirement matching used in traditional resource managers [3, 24], there is an important difference. Resource managers often use a pre-defined list of attributes describing both the hardware and software (e.g., libraries) that are available on a machine. A feature can only be used for matching if it was included in the list to begin with. In addition, complex combinations of attributes often are needed to describe a task's requirements, thereby making the matching procedure complicated and expensive.

In contrast, Constellation uses simple *application defined* labels to describe what context an activity requires and an executor has to offer. By using application specific knowledge, the distinctions between the different activities and executors are often easy to make. For example, for many applications, simple labels such as `"GPU"` or `"dataset X"` are sufficient to classify the executors. This makes context matching simple and fast, allowing fine-grained applications to be scheduled efficiently.

Events can be used for communication with activities. When an activity is created, it is assigned a globally unique ID. When sending an event to an activity, this ID can be used as a destination address. Events are mainly used for signaling between activities, for example, to indicate that certain data is available or that certain processing steps have finished. However, if necessary, they can also be used to transfer data between activities.

The complete set of activities does not have to be known in advance. During the application's lifetime, new activities may be submitted to the Constellation RTS, either by some external application, or spawned dynamically by activities that are already running. Newly created activities are activated on a suitable executor to perform processing. When finished, the activity may decide to either terminate, or suspend and wait for events. Whenever an activity receives an event, it is reactivated to allow it to process the event and perform additional processing, if necessary. When finished, the activity must again decide to terminate or suspend.

It is up to the Constellation RTS to ensure that all activities in an application are run on suitable executors. In addition, the RTS must route any events that are generated to the correct destination. In the following sections we will take a closer look at the Constellation API and then describe the implementation in more detail.

3.2 Constellation API

In this section, we will describe the Constellation API. Like most of the software developed in the Ibis project, Constellation itself is implemented in Java, which provides both portability and acceptable performance. In our current prototype, we assume that the Constellation application (i.e., the glue code connecting the activities) is also a Java application. Although a Java class is used to represent an activity, the code performing the actual processing may be implemented using scripts, C, CUDA, MPI, etc. To implement this, the Java activity class can execute an external application or perform a library call when it is activated.

Pseudo code for the Constellation API is shown in Figure 2. An application can create its own activities by extending the `Activity` class. This class expects the application-

```
class Activity {
    // Constructor, invoked by subclass
    Activity(Context c);

    // Usable by subclass
    AID submit(Activity a);
    void send(Event e);

    // Implemented by subclass
    boolean initialize();
    boolean handleEvent(Event e);
    void terminate();
}
-------------------------------------------------
class Event  {
    Event(AID target, Object message);
}
-------------------------------------------------
class Context {
    Context(String label, long rank);
}
class ContextList extends Context {
    ContextList(Context [] c, bool priority);
}
-------------------------------------------------
class Executor {
    enum Preference { BIGGEST,SMALLEST,RANGE,ANY };

    // Constructor, invoked by subclass
    Executor(Context context, Preference p, ...)

    // Usable by subclass
    boolean processActivity();
    boolean processActivities();

    // May be overwritten by subclass
    void run();
}
-------------------------------------------------
class Constellation {
    // Factory method to create instance
    static Constellation create(Properties p,
                                Executor [] e);
    // Shut down instance
    void done();

    // Submit an activity
    AID submit(Activity act);
}
```

Figure 2: Constellation API

defined subclasses to provide a `Context` to its constructor upon creation. This context is used by Constellation to perform matchmaking.

When an application extends the *Activity* class, it must implement three methods, `initial`, `handleEvent` and `terminate`. These methods will be invoked to perform the initial processing when the activity is first activated, when event handling is required, and when the activity is terminated, respectively. The boolean result returned by `initial` and `handleEvent` is used to indicate if the activity wishes to suspend (`true`) or terminate (`false`). Two additional methods `submit` and `send` are provided that can be used to submit new activities and send events. When submitting an activity, a globally unique *activity identifier* (AID) is returned. This activity identifier can be used as a target address when an *Event* is send to the new activity.

Figure 2 also shows pseudo code for the Context API used in Constellation. A `Context` is defined by two parameters. First, a *label* must be specified, which 'defines' the context. All activities with the same label belong to the same class of contexts. Second, an optional *rank* parameter may be provided. This allows a ranking to be imposed on the context objects in single class. This can be used, for example, to specify differences in data size or priority.

Using *ContextList* it is possible to define a list of contexts. This way an executor can have multiple options for selecting activities, and an activity can indicate that it can run on several types of executors. This list can be treated as a regular 'or', or as a priority list where the order of the contexts signifies a preference. Using this mechanism, executors can specify that they prefer a certain type of activities, but accept others if necessary.

The `Executor` class, as shown in Figure 2, does not have to be extended by the application. The default behavior of the executor is to repeatedly find and execute activities that match the context that is provided to its constructor. The `Preference` parameter specifies which activity should be selected if multiple are available. This selection is done based on the rank of the contexts. It is possible to select the biggest, the smallest, any in a range, or any available activity. When selecting a range, two additional parameters are used to specify the lower and upper bound of the range.

The behavior of the executor can be changed by overriding the `run` method. When doing so, the `processActivity` or `processActivities` methods can be used to instruct the executor to (attempt to) execute one or more activities.

Finally, Figure 2 also shows the `Constellation` class that can be used to interact with the Constellation RTS. This class contains a factory method `create` that can be used to create a new instance of Constellation on the current machine. One or more executors can be started by providing them as parameters. When finished, the `done()` method can be used to terminate the Constellation instance.

The `submit` method can be used to submit activities to Constellation. These activities may be executed by any suitable executors in the pool of Constellation instances. The Constellation RTS will then use a *context aware work stealing* algorithm to perform load balancing over the available executors. This will be explained in the next section.

3.3 Constellation Run Time System

It is the task of the Constellation RTS to ensure that all activities in an application are run on suitable executors. In addition, load balancing should be performed to utilize the available executors as fully as possible. In Constellation, a single *context aware work stealing* algorithm is responsible for both. Whenever an executor becomes idle, it selects another executor and sends it a request for work that includes its context. The executor receiving the request can then perform context matching to find a suitable activity in its queues. If one is found, it is returned. Otherwise the idle executor is notified that no work is available. This process is repeated until work is found.

There are several possibilities when selecting an executor as a steal target. The most straightforward option is to designate a single executor as master and store all unprocessed activities there. For many applications this *master-worker* approach to load balancing is sufficient.

However, Constellation does not assume that the activities are generated in a single location. Instead, any participating Constellation instance may be used to submit activities to the system. In addition, running activities may also submit new activities. As a result, new activities may be generated in a distributed fashion.

```
1. prepare temporary directory and copy input files
2. detect candidate objects in image A
3. detect candidate objects in image B
4. match results of 2 and 3 to produce candidate list
5. equalize image A
6. equalize image B
7. divide results of 5 and 6 into tiles
     for each tile
     if there are enough candidates object in tile
       compare tile A' and B'
8. extract the coordinates of supernova candidates
9. cleanup the temporary directory
```

Figure 3: Sequence of activities used in supernova detection application.

For such scenarios, Constellation also supports *random work stealing*, where idle executors send their steal requests to randomly selected targets. As shown in [28], random work stealing is well-suited to perform load balancing in a distributed environment, provided that a reasonably up-to-date list of executors is available.

Currently, these are the only two work stealing algorithms supported by the Constellation RTS. However, the RTS can easily be extended to support hierarchical work stealing [12] or algorithms based on peer-to-peer techniques [8], for example. When starting an application, the desired work stealing algorithm can be selected by providing a parameter to the Constellation RTS.

4. EXAMPLE APPLICATION

The example application that we will use is our winning contribution to the International Data Analysis Challenge (DACH) for Finding Supernovae, which was held in conjunction with the IEEE Cluster/Grid 2008 conference[2]. This is one of the applications that inspired us to develop Constellation. It is representative of many scientific data-analysis applications, both in scale, form and requirements.

For the challenge, a large number of celestial images were provided, based on data collected by the Subaru Telescope (Hawaii), operated by the National Astronomical Observatory of Japan[3]. A sequential supernova detection application was also provided. The application takes two celestial images of the same segment of the sky, taken about a month apart. It aligns the images, compares them, and uses the result to detect objects with varying light intensity (so called supernova candidates). Since the object detection is based on heuristics, the processing time required varies significantly between image pairs. The resolution of the images also varies, further increasing the variation in processing time.

The goal of the challenge was to process all image pairs as quickly as possible, using 11 different cluster sites in Japan. Each cluster consisted of a homogeneous set of compute nodes with either 2, 4 or 8 cores. As part of the challenge, the input images were distributed over the sites. Although individual image pairs were always stored together, no single site contained the entire input data set. Instead, the data set was both partitioned and partly replicated over the different locations. Therefore, coordination between the sites

[2]http://www.cluster2008.org/challenge
[3]http://subarutelescope.org

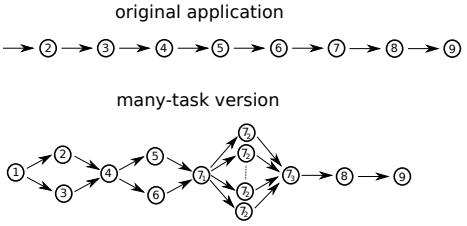

Figure 4: Task graph of original and parallel version of data challenge code.

was necessary in order to ensure that each image pair was only processed once. The amount of data stored in each site was not adapted to the processing capacity of the site. As a result, some sites contained more data than they could handle locally, while others ran short. For optimal load balancing, data had to be shipped between sites. This had be reduced to a minimum.

The application did not consist of a single executable. Instead, it was a sequence of several smaller activities, as shown in Figure 3. Each of these activities used a separate executable and/or script to perform some processing on the output files of its predecessors. Of these activities, (7) was by far the most compute intensive. While most activities required 1 to 8 seconds of processing, (7) can take anywhere between 5 and 20 minutes, depending on the resolution of the input images, the hardware that was used, and the number of candidate objects identified by the heuristics.

In the original application, the activities were executed in sequence (as shown in shown in Figures 3 and 4). As a result, the processing time was dominated by the time a single core required to process the largest inputs.

Our contribution to the data analysis challenge converted the original sequential application into a parallel one, as is shown in Figure 4. By running activities 2 and 3, and 5 and 6 in parallel and by exploiting the fact that activity 7 can be split up into many smaller tasks, we could take advantage of the multi-core properties of the machines in the testbed. Although each image pair is still processed by a single machine, several tasks in the graph can be processed in parallel.

For the challenge, we created an ad-hoc work scheduling infrastructure to coordinate the computation between the different sites. This infrastructure was one of the inspirations for Constellation. Next, we will describe two ways in which Constellation can be used to implement this distributed supernova detection application.

4.1 Monolithic Version

First, we will use Constellation to create a straightforward implementation of the supernova detection application. This implementation replicates the behavior of our original contribution to the DACH. Pseudo code is shown in Figure 5.

The implementation presented here uses the supernovae detection as a single monolithic activity. Each participating machine is represented by a single executor. We assume that the necessary scripts and applications are pre-installed on all participating machines. It is up to the supernova detection code to detect how many cores are available on the machine it executes on, and adjust its parallelism accordingly. Therefore, Constellation is only used to assign

```
class DetectNova extends Activity {
    String file1, file2;
    AID parent;

    DetectNova(Context c, AID parent,
               String file1, String file2) { ... }

    boolean initial() {
       String output = doDetection(file1, file2);
       send(new Event(parent, output));
       return false; // please terminate
    }
}

class Main extends Activity {
    String [] clusters;

    Main(Context c, String [] clusters) { ... }

    boolean initial() {
      // Get list of files and their location
      InputInfo info = getPairs(clusters);

      for (Input i in info) {
         Context c = new Context(i.location, 0);
         submit(new DetectNova(c, myID,
                               i.file1, i.file2));
      }

      return true; // please suspend
   }
   boolean handleEvent(Event e) {
      ... // handle result here
      boolean done = ... // determine if we are done
      return done;
   }
}
```

Figure 5: Pseudo code for activities used in the monolithic supernova detection activity.

activities to machines. It has no control over how efficiently each machines is then used.

Figure 5 shows the two activity classes needed to implement this application. Additional code needed to create the executors and set up the pool of Constellation instances is not shown. The `Main` activity is submitted once. It first gathers a list of input image pairs and their locations, using the `getPairs` method (implementation not shown). Next, the main application creates a single `NovaDetection` activity for each set of input files. Since we assume that the necessary script and applications are pre-installed on all sites, only the paths (or URIs) of the input files need to be provided. The globally unique ID of `Main` is provided to `NovaDetection` to allow the result to be returned.

The final parameter provided to `DetectNova` is the context needed to match it with suitable executors. In this example, we use the location where the input files can be found as a label. A similar label is used by the executors which describes the location in which they are running. By matching these labels we can ensure that the activities will only be run by executors that have the data available locally. Note that this is just one example of how we can use contexts in this application. In Section 4.3, several alternatives will be shown.

4.2 Many-Task Version

The previous implementation treats the supernova detection as a single monolithic application that is applied to a set of independent input files. However, as Figures 3 and 4 show, the supernova detection application internally consists of a large number of tasks. Therefore, we can decompose the application into separate activities for each processing step, thereby transforming it into a many-task application.

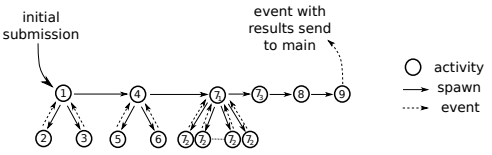

Figure 6: Task graph of the many-task version.

To do so, most of the 9 steps shown in Figure 3 need to be converted into separate activities. Steps 2 and 3 represent the same activity with a different input files, as do steps 5 and 6. Step 7 can be split further into three separate types of activities, one that splits the input images into tiles (7.1), one that processes a pair of tiles (7.2), and one that recombines the results of the tiled processing (7.3). The exact number of (7.2) activities required depends on the resolution of the image pair that is being processed. The resulting chain of activities is shown in Figure 6.

Figure 7 shows pseudo code for the first two activities. The subsequent activities can be implemented in a similar fashion. As in the monolithic implementation, a `Main` activity submits a `DetectNova` activity for each image pair. When activated, this `DetectNova` now submits two new `Detect` activities which (in parallel) perform step 2 and 3 of the analysis. `DetectNova` then suspends while waiting for the results to be returned using events. Once the results are in, the next activity `Match` (step 4, not shown) is submitted, after which `DetectNova` terminates. Similarly, the `Match` activity will then proceed to submit step 5 and 6, wait for the results, submit step 7 and terminate. This continues in a similar fashion until the last activity (step 9) is finished and sends the result back to `Main`.

If all activities in the many-task version of the supernova detection are assumed to be sequential, we can use a separate executor for each core. This is only one of the possibilities, however. If certain activities require multiple cores, or specialized hardware (such as GPUs), the configuration and context of executors can be changed to reflect this. In the next section, we will take a closer look at how contexts can be used to change the run time behavior of the monolithic and many-task applications.

4.3 Manipulating Application Behavior

The behavior of the supernova detection applications described above can be manipulating by changing the set of labels used for the activities and executors. This simple but powerful mechanism allows us to adapt the behavior of an application to the available hardware with very little effort. Table 1 shows several options.

The first entry in Table 1 shows a example where no special context is used to run the activities. If all activities and executors in our supernova detection application use the same label, in this case `"anywhere"`, the activities will be executed in random order and distributed randomly over the executors. The label `"anywhere"` is an arbitrary choice. Any label can be used, as long as the activities and executors agree. For our application, this approach does require that all input file data is available on all participating machines.

```
class DetectNova extends Activity {
    String file1, file2;
    AID main;

    DetectNova(Context c, AID main,
        String file1, String file2) { }

    boolean initial() {
        submit(new Detect(file1, myID);
        submit(new Detect(file2, myID);
        return true; // please suspend
    }
    boolean handleEvent(Event e) {
        if (both results are in) {
           // Submit next processing step
           submit(new Match(file1, file2, main);
           return false; // please terminate
        } else {
           ... // store result here
           return true; // please suspend
        }
   }
}
class Detect extends Activity {
    String file;
    AID parent;

    Detect(String file, AID parent) { ... }

    boolean initial() {
        ... // Retrieve input file if necessary
        String result = execDetect(file);
        send(new Event(parent, result));
        return false; // please terminate
    }
}
```

Figure 7: Pseudo code for the first two activities of the many-task supernova detection application.

Label	Behavior
"anywhere"	Random distribution
location	Fixed location
location + size	Fixed location + ordered
location list	Choice of locations
location list + "anywhere"	Choice of locations + fallback
"CPU" or "GPU"	specialized hardware

Table 1: Several example contexts that can be used with the supernova detection application.

Alternatively, a *location* can be used as a label. For the activities, the label represents the location where the input files can be found. For the executors the label contains the location where they are running. As before, the exact content of the labels does not matter. We could, for example, use network domain names (e.g. `"cs.vu.nl"`) or institutions names (e.g. `"VU"`). As long as the labels are used consistently throughout the application, their exact content is irrelevant. By using this location as a label for both activities and executors, we can ensure that all activities are run in the right context, i.e., the location where the data is available. When the size of the image pair is added to the location, the processing of the activities can be ordered, for example, to process them largest first. As we will show in the next section, ordering the activities by size can improve the load balance in an application.

In DACH, the data was not only partitioned, but also partly replicated. Therefore, for some activities there are multiple locations where the data is available. To express this, a context lists can be used (as explained in Section 3.2) to create a *location list* of all locations where an activity may be processed.

We can combine two of the previous approaches to create a priority list context where preferred locations for an activity are listed first, followed by an `"anywhere"`. Similarly, the executors use a priority list context containing their location and `"anywhere"`. This way, the executors will first process all activities for which data is available locally. Once this set has been exhausted, other, non-local activities will be selected. This prevents load balancing problems when the data is distributed unevenly across the sites, while also keeping the number of remote data transfers to a minimum.

The previous examples focus on using contexts to express locality of the data. However, contexts can also be used to describe other aspects, such as specific hardware requirements. For example, in Section 5.3 we will extend our application with an Activity that requires GPU support. To indicate what hardware is required by each activity, we attach the label `"GPU"` to some activities, and `"CPU"` to others. Similarly, the executors are labeled `"GPU"` or `"CPU"` (or both) when they are started. To do so, they need to know if a GPU is available locally. This information can be provided by an external source (e.g., the application user or the batch scheduler) or it can be detected by adding code to the application. Once both activities and executors are labeled, the Constellation RTS has sufficient information to ensure that all activities will be run on suitable hardware.

These examples illustrate that run time behavior of the application can be changed radically by simply changing the labels attached to activities and executors. It is important to note that for all the examples described above, very little changes to the application's source code are necessary. For most strategies, the only difference is in the code that determines which labels to use. Even extending the application with GPU support only requires the definition of an additional new activity (provided that the actual GPU code is available). In the next section, we show that this mechanism can be used to almost effortlessly tune our supernova detection application to run efficiently on three very different hardware configurations.

5. EVALUATION

In this section, we will evaluate the effect of changing context and executor configurations on the performance of the supernova detection application. We explore three scenarios: a distributed set of heterogeneous clusters, a large multi-core machine, and a GPU cluster.

5.1 Scenario 1: Distributed Processing

This first scenario closely resembles the original DACH challenge: the supernova detection application needs to process a data set of 1052 image pairs (73 GBytes in total) using a distributed set of clusters. This data set is both partitioned and replicated over the participating clusters.

As a testbed, we will use four clusters that are part of the Distributed ASCI Supercomputer, DAS-3 and DAS-3[4], two wide-area distributed systems for Computer Science research in the Netherlands. Both systems consist of multiple clusters located at different universities and research institutes in the Netherlands. The wide-area interconnect between the clusters is based on lightpaths, provided by SURFnet-6

[4]http://www.cs.vu.nl/das3, http://www.cs.vu.nl/das4

cluster	cores per node	cores used	available image pairs
LU	2x 2.6 Ghz	40	105 (105 shared w. UvA)
UvA	4x 2.4 Ghz	80	210 (105 shared w. LU)
VU3	4x 2.4 Ghz	80	525 (105 shared w. VU4)
VU4	8x 2.4 Ghz	160	422 (105 shared w. VU3)

Table 2: Testbed used in scenario 1.

configuration	processing time (s)
anywhere	1942
anywhere sorted	1721
location sorted	1774
location sorted + anywhere	1416

Table 3: Total processing time for each configuration in scenario 1.

(the Dutch NREN). The DAS-3 consists of dual-CPU AMD Opteron 280 (dual core) and AMD Opteron 252 (single core) compute nodes with 4 GBytes memory each. DAS-4 contains dual-CPU, quad core Intel Xeon E5620 compute nodes with 24 GBytes memory each.

Details for the configuration used in this scenario are shown in Table 2. For DAS-3 we use the abbreviations LU, UvA and VU3 for the clusters at Leiden University, the University of Amsterdam and VU University, respectively. We use VU4 for the DAS-4 cluster at VU University. We use 20 nodes on each cluster, resulting in a total of 80 nodes containing 360 cores. Table 2 also shows how the 1052 image pairs are distributed over the clusters. The total data set is partly partitioned and partly replicated. As in the original data challenge, the data is distributed in an arbitrary way. The LU and UvA clusters contain 210 unique files, of which they share 105, and 105 are only stored on UvA. The VU3 and VU4 clusters contain 842 unique files, of which 105 are shared between the two clusters.

All data is stored on the cluster head nodes, and made available to the compute nodes (both local and remote) using an http server. Because the clusters are geographically quite close and connected via a high-speed optical interconnect, file transfer between sites is generally not a bottleneck. Therefore, to simulate the large geographical distance and limited bandwidth we encountered in the original DACH challenge, the http file transfers between clusters are artificially limited to 1 MByte/sec (per transfer). File transfers within a cluster have no bandwidth limit.

We ran the monolithic version of the application described in Section 4.1 on this testbed using four different configurations, each using a different context for the activities and executors. In all configurations, a node is represented by a single executor, regardless of the number of cores a node has available. It is up to the external supernova detection application to use these cores efficiently. The total processing time required for the entire data set is shown in Table 3. In addition, Figure 8 provides a detailed view of the utilization of the cores during single run of each experiment. While the exact utilization may vary between experiments due to the variation in work distribution, the overall trend is the same. We will explain each configuration below.

The first configuration uses `"anywhere"` as a label (see Section 4.3). As a result, the data is processed in arbitrary order. The location of the data is not taken into account, resulting in many expensive inter-cluster data transfers. It takes 1942 seconds to process the entire data set.

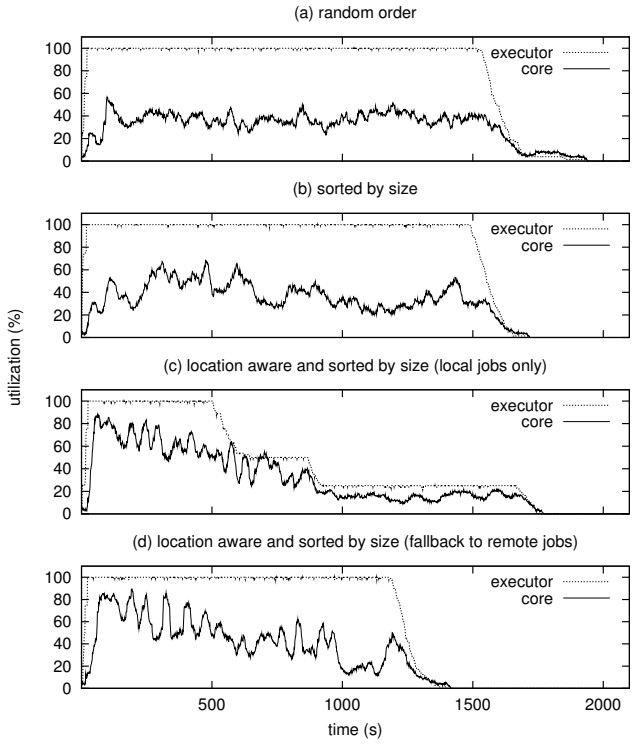

Figure 8: Executor and core utilization of 4 configurations of the monolithic supernova detection application running on 4 heterogeneous clusters.

Figure 8 (a) shows the utilization (in %) of the system while the application is being run. Two lines are shown, the executor utilization and the core (CPU) utilization. The executor utilization indicates the percentage of executors that is busy at any given time. As the figure shows, executor utilization is 100% for most of the run. Therefore, Constellation is successful in balancing the load between executors.

Core utilization is much lower, however, generally around 40%. This difference can easily be explained. Whenever an executor is running an activity, it registers as being fully utilized. This does not mean, however, that the activity is capable of fully utilizing all cores that are available to the executor. The first stage of the supernova detection performs (potentially expensive) file transfers, which require little CPU time. In addition, the parallel task graph of the supernova detection contains several sequential stages and stages with limited parallelism (as explained in Section 4). This further limits the core utilization.

As Figure 8 (a) shows, an interesting effect occurs at the end of the execution. When most executors have finished, a small set of executors remains that still requires a significant amount of processing time. This is causing a ”tail” in the utilization graph. Such *stragglers* are caused by the random execution order of the activities. Because of this random order, there is a chance that a few relatively large activities are executed last, thereby extending the execution time significantly. Fortunately, this problem can easily be solved by sorting the activities according to there size.

The second configuration still uses using `"anywhere"` as a label, but adds a rank to each context equal to the size of the

input image pair. In addition, the executors are configured to prefer activities with the largest rank they can find. As a result, the largest activities will be processed first. Their location is not taken into account, so file transfer between clusters are still required.

By processing the largest image pairs first, the "tail" effect appearing in the first configuration is removed, as is shown in Figure 8 (b). This reduces the overall processing time by 11%, from 1942 to 1721 seconds. The core utilization is still low, however. As explained above, this is partly caused by expensive file transfers.

To reduce this file transfer overhead, the third configuration uses a context list for each activity containing all cluster names where the input images can be found. As before, a rank is added based on the size of the image pair. The executors are labeled with the name of their local cluster and have a preference for larger activities. As a result, executors will only process activities for which the image pair can be found locally, thus preventing remote file transfers altogether.

As Figure 8 (c) shows, this approach initially increases core utilization to over 80%. This slowly drops as the activities become smaller. After approximately 500 seconds, the UvA and LU clusters run out of work prematurely and become idle, thereby causing a sharp drop in executor utilization. After approximately 800 seconds, VU4 also runs out of work. Despite this severe load imbalance, the overall processing time of 1774 seconds is close to the previous configuration. This indicates that the performance degradation caused by the load imbalance is canceled out by a performance gain caused by a reduction in file transfers times.

In the fourth configuration we again label the activities with list clusters name where their input can be found. In addition, we also add `"anywhere"` as a fallback. Similarly, the executor will use their local cluster name and `"anywhere"` as a label. As a result, executors will first process all activities for which image pairs are available locally. When this set is exhausted, they will fall back to processing remote image pairs in random order. As Figure 8 (d) shows, this approach ensures that all clusters remain active until the end of the application, thereby reducing the processing time to 1416 seconds.

As the experiments in this scenario show, Constellation allows the behavior of applications to be altered significantly by simply changing the context labels used for the activities and executors. Using this mechanism, applications can almost trivially be made size-aware, location-aware, or both. The performance of an application can be improved significantly by simply labeling the activities and executors in a way that is suitable for the environment in which the application is run. In this scenario, the performance difference between the first and last configuration is 27%. Note that only the labels were changed. No other changes where made to the application.

However, the experiments in this scenario also show that the external supernova detection application itself is not capable of fully utilizing all cores, even if expensive inter-cluster file transfers are reduced to a minimum. In the next scenario, we take a closer look at this problem.

5.2 Scenario 2: Multi-core processing

The previous scenario has shown that the utilization of the cores in a machine may be suboptimal, even if Constellation fully utilizes all executors. The core utilization was reduced

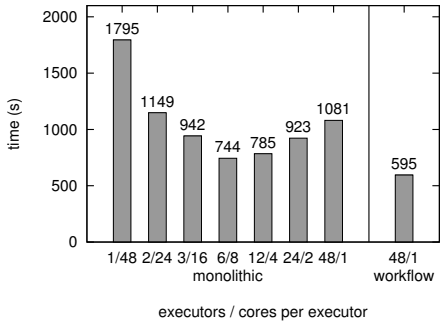

Figure 9: Processing time required for a data set of the 30 largest image pairs using various configurations of the supernova detection application running on an 48-core machine.

by the lack of parallelism in the existing supernova detection application. This is unfortunate, as there is a current trend towards machines with more instead of faster cores.

In this scenario we will take a closer look at how we can solve this problem, either by reconfiguring the number of executors used by Constellation, or by giving Constellation more control over the individual activities that constitute the supernova detection application.

As a testbed, we will use a Dell PowerEdge R815 containing four 2.1 Ghz AMD Opteron 6172 (Magny Cours) processors (12 cores each) and 128 GBytes of memory. As this machine has a total of 48 cores, it is unlikely that the existing application will scale sufficiently.

Instead of processing the entire data set, we have created a subset of the 30 largest images pairs. Together, they represent about 10% of the entire data set. Their sequential processing time ranges from 65 seconds to 1040 seconds, with an average of 820 seconds. All data is available locally.

First, we ran the monolithic version of the supernova detection application described in 4.1 on the 48-core machine with various executor configurations. These configurations range from 1 executor offering 48 cores (referred to as 1/48), to 48 executors offering 1 core each (48/1), with other configurations such as 2/24 and 6/8 in between. When an executor starts a supernova detection, it informs the application of the maximum number of cores it is allowed to use.

The results are shown in Figure 9. Since locality is not an issue, the application simply uses an `"anywhere"` label for activities, with a rank equal to the size of the image pair. The executors will execute the largest activities first.

As Figure 9 shows, using a single executor does not perform well. As expected, the limited scalability of the detection application reduces the core utilization significantly, as Figure 10(a) shows. In this figure, a repeating pattern can be seen. Whenever an image pair is being analyzed, much of the time is spent in pre and post processing, which offers only limited parallelism. The machine is only fully utilized when the individual image tiles are being processed.

When using the opposite approach, 48 executors of 1 core each, the performance is improved. However, as there are only 30 activities available, part of the cores remain idle. This can clearly be seen in Figure 10(b), where the utilization is limited to 65%. In addition, since each supernova detection now runs sequentially, the total processing time

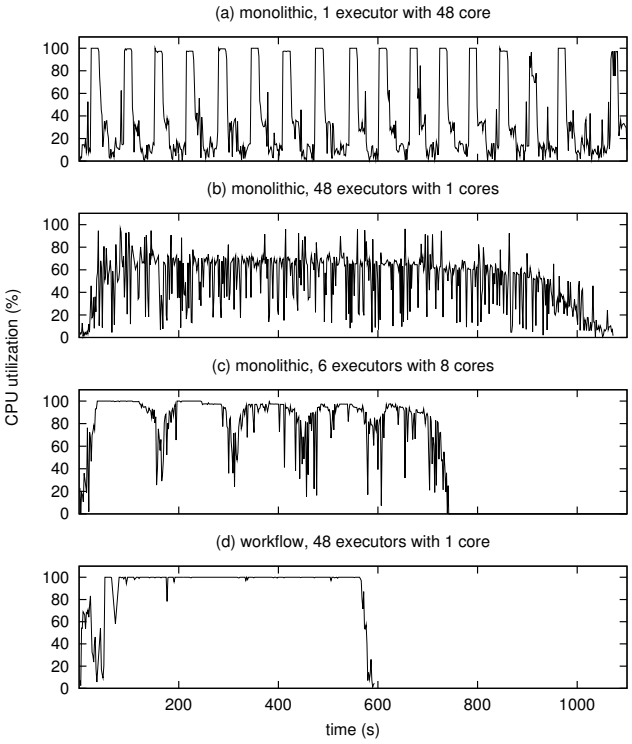

Figure 10: Core utilization of 4 configurations of the supernova detection application running on an 48-core machine.

depends on the longest running activity, causing a small "tail" effect we also saw in the previous scenario.

By selecting an intermediate number of executors, we can configure how many supernova detections we wish to run in parallel, and how many cores each of these detections may use. Figure 9 shows that an optimum is reached when using 6 executors. When using more executors, the increase in processing time per activity caused by the reduced number of cores per executor outweighs the benefit of processing multiple images pairs simultaneously. Figure 10(c) shows the core utilization when using 6 executors. Although the utilization has increased significantly, there is still some room for improvement.

As explained in Section 4.2, one approach to further improve the performance is to decompose the application into separate activities for each processing step. This gives Constellation more control over the individual activities which improves the load balancing during the computation.

Figure 9 shows that the many-task version of the application outperforms the best performing monolithic configuration (6/8) by 20%. When looking at Figure 10(d) it immediately becomes clear why. Unlike the previous versions, the many-task version is capable of maintaining 100% core utilization for most of the execution time. Unlike the monolithic version, the individual activities that comprise a single supernova detection are not forced to run on a fixed set of cores. Instead, the activities of multiple supernova detections are mixed in order to obtain an optimal load balance between the cores.

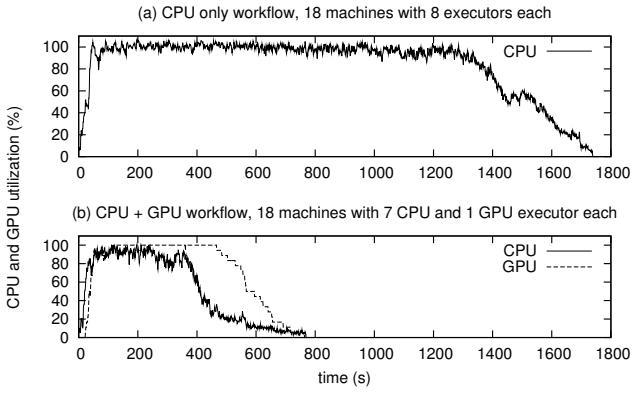

Figure 11: Core and GPU utilization of 2 configurations of the supernova detection application running on a GPU cluster.

Finally, when using the 48-core machine to process the entire data set of 1052 image pairs, the many-task version outperforms the 6/8 monolithic configuration by 42% (5243 vs. 9043 seconds). This performance difference is larger than in the previous experiment due to the many smaller image pairs in the data set. In the monolithic version, a smaller image pair generally leads to a lower core utilization and thus a longer processing time. Due to superior load-balancing properties of the many-task version, the core utilization remains high during the entire application run.

5.3 Scenario 3: Heterogeneous processing

In the previous section, the application assumed that each activity could be run on any of the available executors. Therefore, a simple `"anywhere"` context label was used. There is a trend, however, towards incorporating many-core technologies, such as GPUs, into the computing infrastructure. Therefore, many recent applications consist of a combination of generic CPU code and code that is specifically targeted at a many-core accelerator. Constellation is well suited to express such applications, as we will show in this scenario.

As a testbed, we use 18 compute nodes of the DAS-4 cluster at VU University (see 5.1). Each node contains two quad core Intel Xeon E5620 CPUs (8 cores in total). In addition, 16 of the compute nodes contain a single NVidia GTX480 GPU, while two contain an Nvidia C2050 Tesla GPU. All image pairs are initially stored on the head node of the cluster. Before a compute node starts processing an image pair, the required data is first copied to its local scratch disk.

Figure 11(a) shows the CPU utilization of the testbed when running the many-task version of the application using the entire dataset of 1052 image pairs. For each machine, a separate executor is used for each its 8 cores. The results are similar to those shown in Figure 10(d), which used a smaller 30 image pair dataset. After a small start-up delay caused by the initial copying of the input data, the CPU utilization remains close to a 100% for most of the run. The application completes in 1740 seconds.

Compared to Figure 10(d), Figure 11(a) does show more load imbalance at the end of the run. Currently, the application does not allow individual activities belonging to a single image pair to leave a machine, as that would require transferring a large number of temporary files stored on the

machine's local scratch disk. Therefore, for a single image pair, load balancing is only performed between the cores of a single machine. In the previous section this did not influence our performance, as we used a single 48-core machine. When running on 18 machines, however, this limitation does cause a slight load imbalance at the end of the run. To reduce this problem, we are planning to extend the application to support the transfer of temporary files.

To use the GPUs available in our testbed, we have ported one of the application's activities to CUDA. As explained in Sections 4, activity (7.2) (see Figure 6) is by far the most compute intensive. Only a small part of the code of this activity needs to be ported, as most code performs file I/O, initialization and pre-processing of the data. We have therefore split this activity into two separate activities: one that will run on the CPU and contains most file I/O, initialization and pre-processing, and one that runs on the GPU and contains the compute intensive image processing. Only the second type of activity will be labeled `"GPU"`. All other activities will use the `"CPU"` label.

As before, each machine uses 8 executors, 7 labeled `"CPU"`, and one labeled `"GPU"`. Each machine will therefore have a single executor responsible for running the GPU activities. Note that a single CPU core is allocated for managing the GPU. This is the best fit for our scenario, as the machines only contain a single GPU. To run the application on multi-GPU compute nodes it would be sufficient to mark an additional executor as `"GPU"`. No further changes to the application are required.

Figure 11(b) shows the CPU and GPU utilization of the testbed when running the heterogeneous application. As before, after a short delay, the utilization increases to about 100% for both CPUs and GPUs. After some 300 seconds however, the CPU utilization starts dropping. At that time, the pre-processing for most image pairs is done and the CPUs start running out of work. The GPU utilization remains high however. Some 500 seconds into the experiment, the GPUs also start running out of work. This is caused by the same load imbalance we saw in the CPU-only version. Despite this load imbalance, the application completes after 765 seconds, less then half of the time needed by the CPU-only version.

This experiment shows that when alternative implementations of certain activities are available, very little programming effort is required to use them in an application. After tagging the activities and executors with the appropriate labels, Constellation automatically map all activities to the appropriate hardware. Further extending the application for execution on an even more heterogeneous Jungle Computing system would be just as easy.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced Ibis/Constellation, a light-weight many-task computing platform specifically targeted at running applications on a distributed and heterogeneous set of resources. The key advantage of the MTC paradigm is that applications consists of *distinct activities* which can be implemented *separately* using the tools, and targeted at the HPC architecture, that suit them best.

Constellation allows for a simple definition of activities and the relationships between them. In addition Constellation uses *contexts*, application-defined labels that can be attached to both activities and executors (i.e., hardware). These contexts provide the application programmer with a simple mechanism to express which activities can be executed where. A context can be used the describe data locality, hardware capabilities, problem and resource sizes, etc. Constellation provides a simple and efficient matchmaking and load balancing mechanism, based on *context aware work stealing*, that ensures that each activity is forwarded to a suitable resource.

We have shown that an existing supernova detection application can be ported to Constellation with relatively little effort. By simply changing the context definition of the activities and the executor configuration, this application can run efficiently in three very different HPC computing environments: a distributed set of clusters, a large 48 core machine, and a GPU cluster. By only changing the context used, a 360 core distributed application can almost trivially be made size-aware, location-aware, or both, improving the performance significantly in the process. By changing the executor configuration, the same application can be optimized to run efficiently on a large 48-core machine.

The performance was improved even further by decomposing the application into separate activities, and giving Constellation full control over the load balancing. By replacing a single activity in the application with a GPU version, and using the appropriate contexts and executors, we created a heterogeneous application that ran efficiently on 18 node GPU cluster (144 cores, 18 GPUs), more than doubling the performance compared to a CPU-only version.

The work we have presented in this paper are just the first steps toward Jungle Computing. As future work, we are planning to evaluate Constellation with more complex applications that require combinations of different types of hardware (GPUs, FPGAs, Cells, etc.) to be used simultaneously. In addition, we would like to experiment with applications that provide several alternative implementations of activities thereby giving Constellation a choice of what hardware to use.

We would also like extend Constellation to allow for dynamically changing sets of executors. By allowing the set of executors to grow, shrink or change their context on demand, we can tune the set of resources used to the (possibly dynamic) needs of the application. This would also required Constellation to do its own resource management instead of expecting the user to reserve the necessary resources in advance. In the Ibis project, we have ample experience in this area [2, 8]. We are also planning to further evaluate the efficiency of Constellation's work stealing algorithms when running very fine grained applications. Finally, we aim to use Constellation as a basis for further development of high level domain specific programming models.

Acknowledgments

We would like to thank Kees Verstoep at VU University for his efforts to get the DAS-4 cluster running in time for our experiments. We would also like to thank Dr. John Romein at ASTRON for providing access to a 48-core machine.

7. REFERENCES

[1] C. Augonnet et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011.

[2] H. E. Bal et al. Real-world distributed computing with Ibis. *Computer*, 43:54–62, August 2010.

[3] B. Bode et al. The portable batch scheduler and the maui scheduler on linux clusters. In *Proc. of the 4th annual Linux Showcase & Conference*, ALS'00, pages 27–27, Atlanta, Georgia, 2000. USENIX Association.

[4] M. Bornemann et al. MPJ/Ibis: a flexible and efficient message passing platform for Java. In *Proc. of 12th European PVM/MPI Users' Group Meeting*, pages 217–224, Sorrento, Italy, September 2005.

[5] NVIDIA CUDA Programming Guide Version 2.0, 2008. http://www.nvidia.com/cuda.html.

[6] E. Deelman et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13:219–237, July 2005.

[7] E. Deelman et al. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[8] N. Drost et al. Zorilla: A peer-to-peer middleware for real-world distributed systems. *Concurrency and Computation: Practice and Experience*, 2011. Accepted for publication.

[9] W. Gentzsch. Sun Grid Engine: Towards creating a compute power grid. In *1st International Symposium on Cluster Computing and the Grid*, 2001.

[10] R. L. Graham et al. Open MPI: A flexible high performance MPI. In *Proc. 6th Int. Conf. Par. Proc. and Appl. Math.*, pages 228–239, Poznan, Poland, Sept. 2005.

[11] T. D. Hartley et al. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proc. of the 22nd annual international conference on Supercomputing*, ICS'08, pages 15–25, Island of Kos, Greece, 2008. ACM.

[12] V. Karamcheti and A. A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Proc. of the 1998 ACM/IEEE conference on Supercomputing*, SC'98, pages 1–17, San Jose, CA, 1998. IEEE Computer Society.

[13] J. Maassen and H. Bal. SmartSockets: Solving the Connectivity Problems in Grid Computing. In *Proc. of the 16th International Symposium on High Performance Distributed Computing*, HPDC'07.

[14] J. Makino et al. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. *Publications of the Astronomical Society of Japan*, 55:1163–1187, Dec. 2003.

[15] E. Ogasawara et al. Exploring many task computing in scientific workflows. In *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS'09, pages 2:1–2:10, Portland, Oregon, 2009. ACM.

[16] A. Plaza. Recent developments and future directions in parallel processing of remotely sensed hyperspectral images. In *Proc. of the 6th International Symposium on Image and Signal Processing and Analysis*, pages 626–631, Salzburg, Austria, september 2009.

[17] S. Portegies Zwart et al. A multiphysics and multiscale software environment for modeling astrophysical systems. *New Astronomy*, 14(4):369 – 378, 2009.

[18] I. Raicu et al. Falkon: a fast and light-weight task execution framework. In *Proc. of the 2007 ACM/IEEE conference on Supercomputing*, SC'07, pages 1–12, Reno, Nevada, 2007. ACM.

[19] I. Raicu et al. Many-task computing for grids and supercomputers. In *Proc. of the 1st Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS'08, 2008.

[20] I. Raicu et al. Toward loosely coupled programming on petascale systems. In *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, SC'08, pages 1–12, Austin, Texas, 2008. IEEE Press.

[21] R. Raman et al. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing*, HPDC'98.

[22] F. Seinstra et al. High-performance distributed video content analysis with Parallel-Horus. *IEEE Multimedia*, 14(4):64–75, October-December 2007.

[23] F. J. Seinstra et al. Jungle computing: Distributed supercomputing beyond clusters, grids, and clouds. In M. Cafaro and G. Aloisio, editors, *Grids, Clouds and Virtualization*, Computer Communications and Networks, pages 167–197. Springer London, 2011.

[24] D. Thain et al. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17:2–4, 2005.

[25] A. Tsaregorodtsev et al. DIRAC: A scalable lightweight architecture for high throughput computing. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID'04, pages 19–25, Washington, DC, USA, 2004. IEEE Computer Society.

[26] J. Urbani et al. Owl reasoning with WebPIE: calculating the closure of 100 billion triples. In *Proc. of the ESWC'10*, Heraklion, Greece, may-june 2010.

[27] R. V. Van Nieuwpoort et al. User-friendly and reliable grid computing based on imperfect middleware. In *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, SC'07, Reno, NV, Nov. 2007.

[28] R. V. Van Nieuwpoort et al. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.*, 32(3):1–39, 2010.

[29] K. van Reeuwijk. Maestro: a self-organizing peer-to-peer dataflow framework using reinforcement learning. In *Proc. of The International ACM Symposium on High Performance Distributed Computing*, HPDC'09, Munich, Germany, June 11-13 2009.

[30] E. Walker et al. Personal adaptive clusters as containers for scientific jobs. *Cluster Computing*, 10(3):339–350, 2007.

[31] D. Warneke and O. Kao. Nephele: efficient parallel data processing in the cloud. In *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS'09, pages 8:1–8:10, Portland, Oregon, 2009. ACM.

[32] S. Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing*, 1992.