Applying semi-synchronised task farming to large-scale computer vision problems

Steven McDonagh, Cigdem Beyan, Phoenix X. Huang and Robert B. Fisher

University of Edinburgh

Abstract

Distributed compute clusters allow the computing power of heterogeneous (and homogeneous) resources to be utilised to solve large-scale science and engineering problems. One class of problem that has attractive scalability properties, and is therefore often implemented using compute clusters, is task farming (or parameter sweep) applications. A typical characteristic of such applications is that no communication is needed between distributed subtasks during the overall computation. However interesting large-scale task farming problem instances that do require global communication between subtask sets also exist. We propose a framework called *Semi-synchronised task farming* in order to address problems requiring distributed formulations containing subtasks that alternate between independence and synchronisation. We apply this framework to several large-scale contemporary computer vision problems and present a detailed performance analysis to demonstrate framework scalability.

Semi-synchronised task farming splits a given problem into a number of stages. Each stage involves distributing independent subtasks to be completed in parallel followed by making a set of synchronised global operations, based on information retrieved from the distributed results. The results influence the following subtask distribution stage. This subtask distribution followed by result collation process is iterated until overall problem solutions are obtained. We construct a simplified Bulk Synchronous Parallel (BSP) model to formalise this framework and with this formalisation, we develop a predictive model for overall task completion time. We present experimental benchmark results comparing the performance observed by applying our framework to solve real-world problems on compute clusters to that of solving the tasks in a serial fashion. Furthermore by assessing the predicted time savings that our framework provides in simulation and validating these predictions on a range of complex problems drawn from real-world computer vision tasks, we are able to reliably predict the performance gain obtained when using a compute cluster to tackle resource intensive computer vision tasks.

1. Introduction

Many computational tasks that employ serial code are limited by the total CPU time that they require to execute. When the individual tasks that make up an overall computation are independent of each other it is possible that they run simultaneously (in parallel) on different processors. Using this approach has the potential to greatly reduce the wall-clock time (real-world time elapsed from process start to completion) needed to obtain scientific results. Distributing separate runs of the same code while varying model parameters or input data in this way is known as *task farming* and has been the focus of much work of both cluster and grid computing [1, 2, 3]. Trivial task farming is a common form of parallelism and relies on the ability to decompose a problem into a number of nearly identical yet independent tasks. Each processor (independent node) runs a local copy of the serial code, often with its own input and output files, and no communication is required between these processes. This form of task farming is well suited to exploring large parameter spaces or large independent data sets. On the assumption that all tasks take

Email address: `s.g.mcdonagh@sms.ed.ac.uk, c.beyan@sms.ed.ac.uk, xuan.huang@ed.ac.uk, rbf@inf.ed.ac.uk` (Steven McDonagh, Cigdem Beyan, Phoenix X. Huang and Robert B. Fisher)

Preprint submitted to International Journal of High Performance Computing Applications *April 1, 2014*

a similar amount of time to complete then there are no load imbalance issues and linear scaling can often be achieved in relation to the number of processors employed.

Many interesting problems do however require some level of communication between tasks during distributed execution. In this work we develop a framework to enable *semi-synchronised task farming* in which an overall computation involves distributing many *sets* of parallel tasks such that all tasks *within a set* are independent yet these tasks must finish before a following task set is able to begin execution. Taking into account a level of communication between tasks has been approached previously with a focus on (*e.g.*) the scheduling aspects of aperiodically arriving non-independent tasks [4], data staging effects on wide area task farming [5] and cost-time optimisations of task scheduling [6]. Given that we propose to handle global communication between *task sets* with a post task set completion synchronisation step after a round of concurrent computation, components of the Bulk Synchronous Parallel (BSP) model are a suitable basis for our framework. The BSP model is a bridging model originally proposed by [7] and further detail of how to realise our framework and hybrid time prediction model is provided in Section 3.

Numerical algorithms can often be implemented using either task or data parallelism [8, 9]. Task farming algorithms can be considered a simple subset of task parallel methods that break a problem down into individual segments, such that each problem segment can be solved independently and synchronously on separate compute nodes. The task parallel model typically requires little inter-node communication. Data parallel models conversely share large data sets among multiple compute nodes and then perform similar operations independently on the participating nodes for each element of the data array. Data parallelism therefore typically requires that each processor performs the same task on different pieces of the distributed data. In this way, HPC data parallelism often results in additional communication overhead between nodes and requires high bandwidth and low latency node connectivity. In practice most real parallel computations fall somewhere on a spectrum between task and data parallelism. This is also true of the task farming framework that we introduce (see Section 3).

Computer vision, like many fields, contains algorithms that are challenged by the size of the data sets worked with, the number of parameters that must be estimated or the requirement of highly accurate results. These requirements often result in computationally expensive algorithms that demand time consuming batch processing. One efficient solution for accelerating these processes involves executing algorithms on a cluster of machines rather than on a single compute node or workstation. Our *semi-synchronised task farming* framework provides a simple form of parallel computation that is able to reduce the wall-clock time required by such computationally expensive tasks that might otherwise take several hours, days or even weeks on a single workstation.

Here we choose computer vision applications as the test bed for our framework. Once an algorithm has been formulated under our framework we use simple performance modelling to accurately predict overall computation time and therefore the likely speed up made possible by employing a distributed implementation over a serial approach. In this way we provide a framework that enables the straightforward task distribution for problems, comprised of many individual tasks that likely require communication upon completion, coupled with a modelling process capable of predicting the available speed benefit of instantiating the distributed implementation.

Our contributions in this paper can be summarised as follows:

- We introduce a simple framework for non-independent task farming based on the Bulk Synchronous Parallel (BSP) model [7]. The framework allows us to formulate problems by dividing them into many independent parallel tasks that also require some level of communication and synchronisation between tasks before an overall solution to the problem can be obtained.

- As part of this framework we develop a computation-time model capable of predicting overall application completion time for problems that are formulated using the task farming framework that we introduce. Providing this simple tool affords a method to reliably predict time requirements and evaluate computation-time and solution-quality trade offs prior to runtime.

- We apply our semi-synchronised task farming framework to three contemporary computer vision problems and report on our experiences of implementing distributed solutions to these problems and explore predicted and experimental speed up available when deploying these implementations on an HPC cluster.

The HPC system that we make use of experimentally is described in Section 3.1. We outline our task farming framework and relate it to the BSP model in Section 3.3. We then introduce performance modelling techniques to facilitate predictions about computational time required for problems formulated under our framework in the remaining part of Section 3. Results from simulation experiments that verify our predictive model are given in Section 4. Section 5 details the results of implementing several real world computer vision applications under our framework and these are compared to sequential implementations of the equivalent problems. Finally Section 6 provides some discussion.

2. Related work

The task farming model of high-level parallelism has been the basis for much HPC cluster based work with recent examples utilising HT Condor [10], Google's MapReduce [11] and Microsoft's Dryad [12]. The HT Condor framework is able to harnesses idle cycles from both a network of non-dedicated desktop workstation nodes (cycle scavenging) and dedicated rack-mounted clusters. The framework then employs these cycles to run coarse-grained distributed parallelisation of computationally intensive tasks. Task farming is also common in data centres, for example MapReduce and Dryad both make use of task farming to schedule parallel processing on large terabyte scale datasets. In systems such as these a master process manages the queue of tasks and distributes these tasks amongst the collection of available worker processors. The master process is typically also responsible for handling load balancing and worker node failure. In the current work, master and worker node interaction is handled by Sun Grid Engine (SGE) [13] using a batch queue system similar to the Condor framework. This queueing system is responsible for accepting, scheduling and managing the distributed execution of our parallel tasks. This approach allows the distribution of arbitrary tasks as there is no requirement for a specialised API. Using SGE to manage our task queueing system allows our developers to concentrate on the image processing aspects of the problems that we investigate.

Using the SGE environment, jobs typically request no interaction during execution unless they contain the integrated ability to find their interaction partners from their dynamically assigned worker node. The *semi-synchronised task farming* model that we build on top of the SGE layer respects this such that only after a *set of tasks* has completed are results collated to make decisions regarding the distribution and form of the following set of tasks. In standard task farming, when a worker node completes a task it will request another from the master node and our framework also does this until all tasks in a *task set* are processed. Once all tasks in a *task set* are finished the results are collated before the following set of tasks are defined and distributed. In comparison to standard task farming, many *task sets* likely contribute to a single overall computation under our framework.

Dedicated parallel computer architecture has also been employed to develop computer vision systems. In [14] a Beowulf architecture dedicated to real-time processing of video streams for embedded vision systems is proposed and evaluated. The parallel programming model made use of is based on algorithmic skeletons [15]. Skeletons are higher-order program constructs that encapsulate common and recurring forms of parallelism to make them available to application developers. Skeleton-based parallel programming methodology offers a partially automated procedure for designing and implementing parallel applications for a specific domain such as image processing. An application developer provides a skeletal parallel program description, such as a task farm, and a set of application specific sequential functions to instantiate the skeleton. The system then makes use of a suite of tools that turn these descriptions into executable parallel code. The system in [14] was tested by implementing simple image processing algorithms such as a convolution mask and Sobel filter.

In comparison to classical HPC applications, embedded computer vision on dedicated parallel machines will often be able to offer advantages such as mobile, real-time performance yet places demands on programmers if no high-level parallel programming models or environments are available such as skeletons or the SGE that we make use of in this work (see Section 3.1 for further details). If these tools are not available then programmers must explicitly take into account all low-level aspects of parallelism such as task partitioning, data distribution, inter-node communication and load balancing. If developer expertise lies in (for example) image processing, rather than parallel programming, then accounting for these low-level considerations likely results in long and error-prone development cycles.

In contrast to [14] here we perform task farming as opposed to low-level data parallelism involving geometric partitioning of images for image processing tasks. This results in a coarser level of abstraction that we apply to higher level computer vision problems involving much larger data sets where we do not regard real-time performance as a critical factor. It is for this reason that we consider the BSP model a good basis for our framework. The original BSP model considers computation and communication at the level of the entire program. The BSP model is able to achieve this abstraction by renouncing locality as a performance optimisation [16]. This in turn simplifies many aspects of algorithm design and implementation and does not adversely affect performance for most application domains. Low-level image processing however is an example domain for which locality might be critical so a BSP based framework is likely not the best choice there.

Parallel and distributed computing systems are designed with performance in mind and significant previous work has been carried out developing approaches for performance modelling and prediction of applications running on HPC systems. In addition to the BSP inspired framework that we build on top of the SGE layer we also formulate application performance modelling allowing us to predict the run time performance of the parallel algorithms implemented with our framework. Application performance modelling involves assessing application performance through system modelling and is an established field [17]. Several examples of where this approach has proven advantageous include: input and code optimisation [18], efficient scheduling [19] and post-installation performance verification [20]. The process of modelling itself can be generalised to three basic approaches; modelling based on analytic (mathematical) methods, (*e.g.* LoPC [21]), modelling based on tool support and simulation (e.g. DIMEMAS [22], PACE [23]), and a hybrid approach which uses elements of both (*e.g.* POEMS [24], Performance Prophet [25]). In this work we also choose a hybrid approach and combine basic analytical modelling inherited from the BSP model with traditional code profiling, details of our performance modelling approach are provided in the following section.

3. Semi-synchronised task farming

3.1. HPC experimental implementation

In this work we make use of the Edinburgh Compute and Data Facility (ECDF) [26] to test the parallel implementations of the computer vision problems that we investigate. The ECDF is a Linux compute cluster that comprises of 130 IBM iDataPlex servers, each server node has two Intel Westmere quad-core processors sharing 24 GB of memory. The system uses Sun Grid Engine [13] (SGE) as a batch queueing system. By tackling computer vision problems through parallel computation with SGE we show that increasing the number of participating processors reduces the wall-clock time required for algorithms implemented under our semi-synchronised task farming framework (see Section 5 for experimental details). All algorithms are implemented in Matlab and computation times are recorded using the built-in Matlab command `cputime`. We report on the savings due to application speed up in terms of reduced execution time when running our parallel implementations using many processors compared to employing sequential implementations to perform the same tasks. Our parallel implementations make use of the Distributed Computing Engine (DCE) and Distributed Computing Toolbox (DCT) from MathWorks. These products offer a user-friendly method of parallel programming such that master-slave communication between cluster machines is hidden from the developer, allowing them to focus on domain specific aspects of each problem. Our task farming framework is language independent and we concede that problem instance wall-clock times can likely be reduced further by making use of (*e.g.*) an alternative compiled language. However the primary focus of the current work is to provide evidence that the proposed framework is able to formulate problems consistently and reduce wall-clock times predictably, compared to the related serial implementations, regardless of the language used. We leave a study of time critical applications benefiting from (*e.g.*) compiled languages like C/C++ to future work.

3.2. The Bulk Synchronous Parallel model

The BSP model is a bridging model originally proposed in [7]. It is a style of parallel programming developed for general purpose parallelism, that is parallelism across all application areas and a wide range of architectures [27]. Intended to be employed for distributed-memory computing, the original model assumes a BSP machine consists of p identical processors. The related semi-synchronised farming framework we propose (Section 3.3) does not strictly enforce a homogeneous resource requirement in comparison. This enables our experimental setup, using IBM iDataPlex servers, to contain similar but not necessarily identical nodes. In accordance with the original BSP model we do assume homogeneous resources during our theoretical performance modelling for simplicity and we therefore leave a heterogeneous performance modelling treatment to future work. In the original BSP model, each processor has access to its own local memory and processors can typically communicate with each other through an all-to-all network. In this work we make the simplifying assumption that processes only contribute information to a global decision making process at the end of each *set of tasks* and therefore do not need to communicate with each other directly. A BSP algorithm consists of an arbitrary number of *supersteps*. During supersteps, no communication between processors may occur and all processes, upon completing their current task must then wait at a *barrier*. Once all processes complete their current task a *barrier synchronisation* step occurs and then the next round of tasks (superstep) can begin. In this fashion a BSP computation proceeds in a series of global supersteps and we utilise these supersteps to model *sets of* parallel distributed tasks in our framework. To summarise, a superstep typically consists of three components:

1. Concurrent computation: computation takes place on each of the participating processors p. Processors only make use of data stored in the local processor memory. Here we call each independent process a *task*. These *tasks* occur asynchronously of each other.
2. Communication: Processors exchange data between each other. Our framework makes the simplifying assumption that *tasks* do not need to exchange data with each other individually yet the result of each local computation contributes to the following Barrier synchronisation step (global decision making). This assumption holds for each computer vision application that we investigate (see Section 5).
3. Barrier synchronisation: When each *task* reaches this point (the barrier), it must wait until all other *tasks* have finished their required processing. Once all *tasks* have completed, we make a set of global decisions before the next superstep may begin (the next round of concurrent computation and so on).

3.3. Proposed task farming framework

As noted, our framework involves global communication between *task sets* during a post task-set-completion synchronisation step following a round of concurrent computation. The components and fundamental properties of the Bulk Synchronous Parallel (BSP) model provide a suitable basis for this framework. Namely moving from a sequential implementation to describe the use of parallelism with a BSP model requires only a bare minimum of extra information be supplied. BSP models are also independent of target architecture making a task farming framework based on BSP portable between distributed architectures. Finally the performance of a program distributed using a BSP based framework is predictable if a few simple parameters from the target program can be provided (*e.g.* task-length distribution parameters). This leads to a hybrid performance modelling technique capable of predicting the runtime of algorithms implemented with our framework.

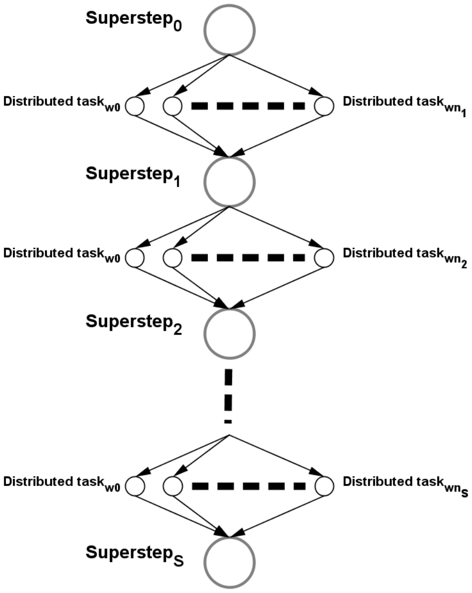
We solve large scale problems by sharing large data sets among multiple processors yet the *semi-synchronised task farming* framework, in consonance with a task parallelism model, involves only little inter-node communication between tasks running in parallel. However, similar to data parallelism models, the framework allows us to split these large data sets between compute nodes and perform independent calculations on participating processors in parallel. As the calculations *within each task* are independent, no information needs to be exchanged between nodes during task runtime and sharing of results is postponed until all tasks in a set have completed. As discussed, once a set of tasks has been completed we are able to collate results and use this information to make decisions relating to how the following round of tasks should be formulated. The outputs from the final round of tasks are combined to provide the global program output. This framework is formally defined in the following pseudocode and Figure 1 depicts the process in diagrammatic form.

```
Let:
$\{I_i^{[t]}\}_{i=1}^{N_t}$be the set of $N_t$ input tasks at superstep $t$

$\{O_i^{[t]}\}_{i=1}^{N_t}$be the set of $N_t$ outputs gained from the tasks completed at superstep $t$

Input:
$N_0$ tasks at superstep $t = 0$
terminate := 0

begin
  while (NOT terminate)
        parallel for $i \in N_t$
            $O_i^{[t]} = \text{process}(I_i^{[t]})$
        end
        $\{I_{i=1}^{[t+1]}\}_{i=1}^{N_{t+1}} = \text{recompute\_inputs}(\{I_{i=1}^{[t]}\}_{i=1}^{N_t}\ ,\ \{O_{i=1}^{[t]}\}_{i=1}^{N_t})$
        terminate $\overset{?}{=}$ test_termination_criteria($\{O_i^{[t]}\}_{i=1}^{N_t}$)
        $t = t + 1$
  end
  $last = t$
  $R = \text{combine\_outputs}(\{O_i^{[last]}\}_{i=1}^{N_{last}})$
end

Output:
$R$
```

The advantage of adding the BSP synchronisation step between task sets allows all tasks in a set the opportunity to collate and communicate information resulting from the completion of their collective execution. The collective results of a task set can influence decisions involving the form, model parameters and possibly the number of tasks making up the following task set input. Once formulated, the following set of tasks can be distributed to the participating processors. It is this process of dispatching multiple rounds of parallel independent tasks, where task formulation may be influenced by information from previous task set results, that we call *semi-synchronised task farming.* This approach allows us to find distributed solutions to non-trivial problems that require a level of communication between nodes during overall computation while retaining much of the simplicity of the standard task farming model. If all tasks *within a task set* take a similar amount of time to complete then it allows for simple modelling and task distribution. If however tasks exhibit completion times with high variance, then a smart scheduler (such as SGE) can still be used efficiently to ensure that load balancing is not problematic for our framework. The wall-clock time, now related to both the number of task sets and the number of available processors, is much improved over serial implementations.

The synchronisation aspect allows us to solve problem decompositions that require a level of inter-node communication while retaining the main advantages of a standard task farming approach such as ease of implementation, level of achievable efficiency (on the assumption that individual tasks in a set require similar time to complete) and, given that existing serial code can often be used with minimal modification, users can produce solutions without requiring detailed knowledge of (*e.g.*) MPI techniques. We do however note that if tasks take widely different amounts of execution time then the total wall-clock time of a task set is governed by the slowest process.

Figure 1: Our *semi-synchronised task farming* framework. Light grey *superstep* nodes indicate task synchronisation and collective global decisions based on information obtained from the previous set of distributed *tasks*. These decision points influence the input data, form (and possibly the number) of the following set of distributed *tasks*. Each task in a task set is distributed to an individual processor. The distributed *tasks* following each *superstep* are not regarded as having a particular linear order (from left to right or otherwise) and may be mapped to processors in any way.



3.4. Simulation and analytical hybrid performance modelling

We undertake simple performance modelling to evaluate the distributed job submission behaviour on a CPU cluster allowing prediction of the run time performance of algorithms realised with our framework. Performance modelling of distributed systems enables an understanding of code and machine behaviour and can be broadly split into two categories; analytical modelling and simulation based techniques. As previously mentioned, analytical models are typically developed through the manual inspection of source code and subsequent formulation of critical path execution time. This approach usually involves the implementation of a modelling framework (*e.g.* LoPC [21]) to reduce the work required by the performance modeller. Analytical approaches are effective yet often require manual analysis of source code necessitating knowledge of the task domain, implementation languages and communication paradigms.

Here we follow a coarse grained alternative approach of simulation based performance modelling. Many simulation tools exist to support this form of performance modelling (*e.g.* the DIMEMAS project [22]). Such tools often involve replaying the code to be modelled instruction-by-instruction and the related use of machine resources can then be gathered by the simulator. More recent work such as the WARPP toolkit [28, 17] make use of larger computational events (as opposed to instruction based simulation) improving simulator scalability. Here we take a similar approach; instead of using single application instructions we model coarse grained computational blocks. We choose a coarse level of granularity by defining a computational block as one distributed task in our framework. We then obtain run times for these computational blocks through traditional code profiling. An additional advantage of this coarse-grained simulation is that hybrid models (combining analytical and simulation-based approaches) can be built. By combining these coarse-grained computational events with an analytical model typical of the Bulk Synchronous Parallel (BSP) [7] model we obtain a straightforward hybrid model capable of predicting application run-time for the algorithms that we implement using our task farming framework.

3.5. BSP cost in relation to task farming

The cost of an algorithm represented by the BSP model is defined as follows. The cost of each superstep is determined by the sum of three terms; the cost of the longest running local *task* w_i, the global communication cost g per message between processors where the number of messages sent or received by *task* i is h_i and the cost of the barrier synchronisation at the end of each superstep is l (which may be negligible and therefore the term is dropped).
The cost of one superstep for p processors is therefore:

$$\max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + l \tag{1}$$

We make standard simplifying assumptions that we have homogeneous processors and that *tasks* do not need to exchange data with each other individually or with the master node during each superstep thus ensuring that $h_i = 0$ for all i. We assume homogeneous processors for simplicity during our cost treatment but note that in the current landscape of computation, heterogeneous resources are also common. Although our framework is applicable to heterogeneous resources in practice, we leave a theoretical treatment of heterogeneous processor cost to future work (see Section 4 for related discussion of this point). It is common for Equation 1 to be written as $w + hg + l$ where w and h are maxima and with our simplification this reduces further to $w + l$. The cost of the algorithm then, is the sum of the costs of each superstep where S is the number of supersteps required.

$$W + Hg + Sl = \sum_{s=0}^{S} w_s + 0 + Sl \tag{2}$$

3.6. Our hybrid BSP simulation

We simulate total parallel algorithm execution times by firstly generating random trials to simulate individual distributed *task* timings. To simulate a real-world task set, we generate trials from a Gaussian distribution parametrised by the mean time required in practice for a single distributed task to complete and add these to the time cost of barrier synchronisation. Task timing distribution parameters are found through code profiling and making use of the Matlab function `cputime`. We assert that this is a reasonable method to simulate task timings as the task farming applications that we investigate all distribute sets of similar length tasks during each superstep. By specifying or observing the number of *supersteps* required for a given real-world computation and the number of distributed tasks required in each *superstep*, we are able to approximate the total time required by the parallel algorithm as:

$$\sum_{s=0}^{S} w_s + Sl \tag{3}$$

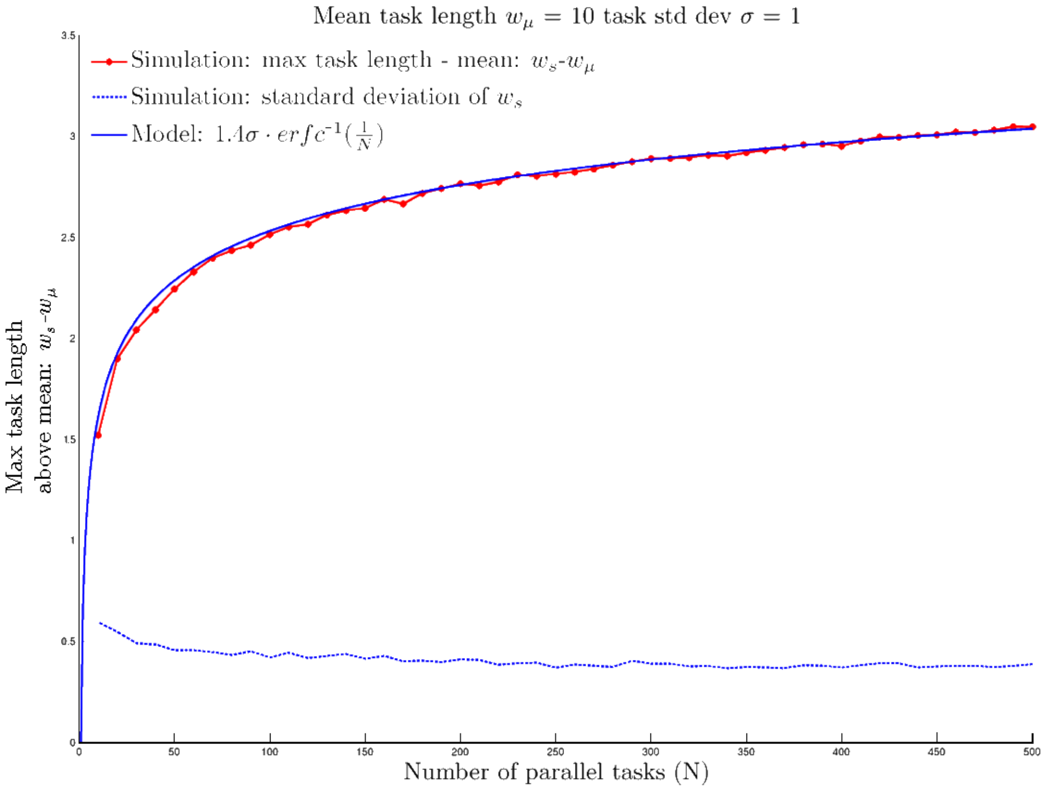
where w_s is the longest running local task in superstep s, barrier synchronisation time cost is l and the total number of supersteps is S. In practice we run this simulation over many trials and look at the mean result for an algorithm that requires N_s distributed tasks during each superstep.

3.6.1. Limitless CPU node model

As a simple example we take a mean task length of $w_\mu = 10$ time units and a task length standard deviation of $\sigma = 1$, and simulate an application making use of only a single superstep. We find that, using the additional assumption of limitless computational nodes, as we increase the number of distributed tasks required in the superstep the difference between the longest task length w_s and the mean task length w_μ grows sub-linearly with the number of submitted distributed tasks N (Figure 2). From this simple example we are able to conclude that, not taking into account limited computational resources, if we have an application that benefits from increasing the number of distributed tasks during a superstep (*e.g.* by an order of magnitude - see for example Section 5.1), we can expect improved results for only a small increase in predicted wall-clock time cost.

We can fit this simulated computation time accurately using the standard inverse complementary error function. The complementary error function *erfc* (also known as the Gaussian error function) provides

Figure 2: Predicted difference between maximum distributed task time and mean task time $w_s - w_\mu$, where $w_\mu = 10$, $\sigma = 1$ for an algorithm distributing N tasks in one superstep.



us with an accurate predictor for the maximum job length w_s increment over the mean job length w_μ, in relation to the number of submitted jobs, that we are likely to observe assuming that the true job length distribution resembles a Gaussian distribution. The *erfc* function is often used in statistical analysis to predict behaviour of any sample with respect to the population mean. Here we fit our simulation data by applying the inverse *erfc* to $\left(\frac{1}{N_s}\right)$, where N_s is the number of submitted tasks in superstep s (see Figure 2). The error function *erf* is defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Then the complementary error function, denoted *erfc* and its inverse *erfc*$^{-1}$ are defined as:

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

$$\mathrm{erfc}^{-1}(1-x) = \mathrm{erf}^{-1}(x)$$

The model that empirically fits the simulation for mean task length w_μ, with standard deviation σ distributing N_s tasks in parallel, lets us predict the maximum task time w_s for superstep s as:

$$w_s = w_\mu + \left(1.4\sigma \cdot \mathrm{erfc}^{-1}(\frac{1}{N_s})\right) \tag{4}$$

The scalar 1.4 is needed to fit our empirical data. We hypothesise that the true scalar value providing the best fit to our empirical curve here is $\sqrt{2}$ but we leave investigation of this to future work. In Figure

2 we use $w_\mu = 10$ and $\sigma = 1$ and simulate for various task set sizes N_s. If computational resources are not a limiting factor, then once we know the number of distributed tasks N_s required per superstep, and have estimates for w_μ and σ we are able to approximate the expected time w_s required for a single superstep of a given algorithm and, given the number of supersteps, the expected time required for the entire algorithm. This model is valid in cases where the number of available parallel worker processors is equal to or exceeds the number of tasks required per superstep. We have access to 130 iDataPlex servers with multiple CPUs, however in many practical applications this requirement will not hold (the number of tasks per superstep will exceed available participating worker nodes) therefore we also consider a finite CPU model in the following section.

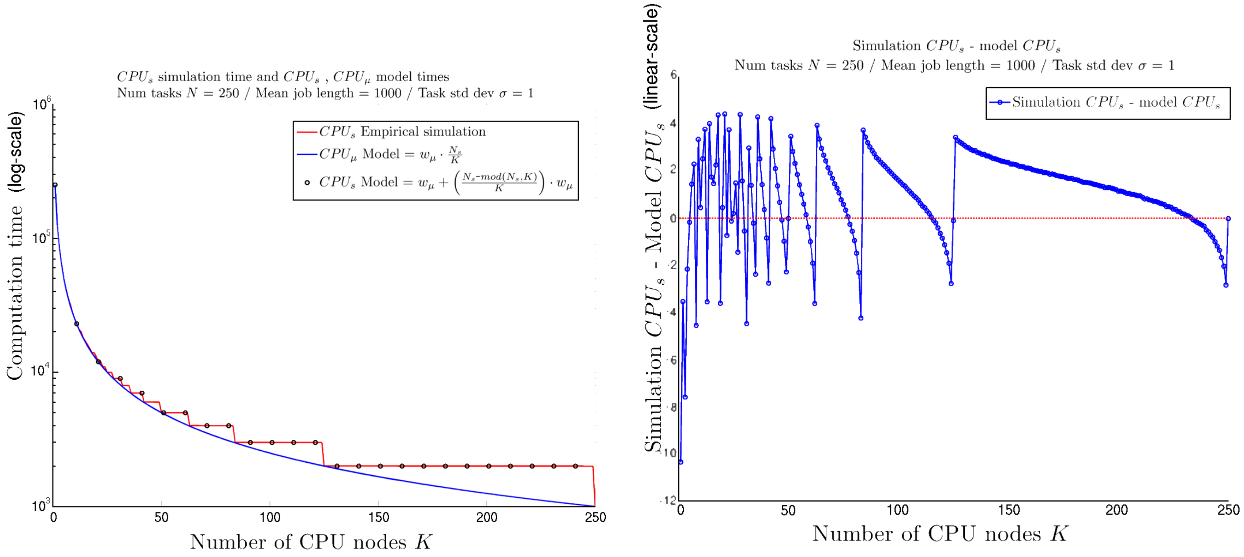
3.6.2. Finite CPU node model

The previous simulation model does not take into account CPU worker node limits. In this section additional simulations are performed to explore the effect of capping the number of available CPU nodes K in relation to the number of submitted distributed tasks per superstep N_s. This allows us to fit a model that reflects our real distributed system pragmatically. In this case, we assume that $N_s > K$ and therefore each CPU node is responsible for the computation of a number of tasks in sequence in order to complete a superstep. In our task farming framework under SGE, when a CPU worker node completes the computation of the current task then the next task from the set still waiting to be processed will be assigned to the finished core such that each core is continually utilised until all tasks have been processed. For each simulation trial, the maximum cumulative CPU computation time used by a worker node during a superstep; CPU_s must now be found. This value is the maximal sum of task computation times assigned to an individual CPU. From this max cumulative computation time found during a superstep, we subtract $w_\mu \cdot \left(\frac{N_s}{K}\right)$ where w_μ is the mean task length, N_s is the number of parallel tasks making up the superstep and K is the number of participating processors. This effectively subtracts the mean amount of work we expect a CPU to perform per superstep. This mean amount of work per CPU is denoted $CPU_\mu = w_\mu \cdot \left(\frac{N_s}{K}\right)$. The resulting difference tells us how much more work, than the mean cumulative work, we expect the node assigned the most work to carry out. As a result, CPU_s provides the time we expect the full superstep s to take to complete.

The final point above holds because all CPU worker nodes must be allowed to finish their assigned cumulative task computation before it is possible to synchronise and conclude a superstep s. When accounting for a finite set of CPU worker nodes we therefore model the time it takes to complete a superstep s as the longest cumulative CPU computation time CPU_s. When accounting for a fixed number of worker nodes K, the model that we find (approximately) empirically fits the simulation data is:

$$CPU_s = \begin{cases} w_\mu \cdot \left(\frac{N_s - \mathrm{mod}(N_s, K)}{K}\right) + w_\mu & \text{if } \mathrm{mod}(N_s, K) \neq 0 \\ w_\mu \cdot \left(\frac{N_s}{K}\right) + 1.4\sigma \cdot \mathrm{erfc}^{-1}\left(\frac{1}{N_s}\right) & \text{if } \mathrm{mod}(N_s, K) = 0 \end{cases} \tag{5}$$

We model CPU_s as the mean computational work done at each worker, CPU_μ plus some additional work that must be carried out by the CPU that has performed the most work in the current superstep. We model this additional work in the following way: when we consider a finite set of CPU worker nodes, the difference between the longest cumulative CPU computation time CPU_s and the mean cumulative CPU computation time CPU_μ is primarily influenced by: 1) how evenly the number of distributed tasks N_s are distributed to the number of participating CPU nodes K and 2) the mean task length w_μ. Advanced task farm models (*e.g.* [29]) employ various strategies dictating how tasks should be distributed to workers. Here we take the simple approach that, on the assumption that tasks belonging to a task set have similar length, each task still waiting to be processed will be assigned in turn to the CPU worker node that finishes its current computational work load first. A consequence of this is that if the total number of distributed tasks N_s required by the superstep is exactly divisible by the number of participating CPU nodes K (*i.e.* $\mathrm{mod}(N_s, K) = 0$) then, excluding cases involving extremely high task length variance σ^2 in relation to w_μ, each CPU will receive an identical number of tasks and therefore the difference between the longest cumulative CPU computation time CPU_s and the mean time CPU_μ will be small and only influenced by the number of tasks N_s and the task length variance σ^2 in a similar fashion to the limitless worker node model. In such cases this small difference is once again accounted for using the $erfc^{-1}$ function as before (see Figure 2 and Equation 4). If, contrarily, the number of tasks N_s divided by the number of participating CPU nodes K leaves a remaining number of tasks that is small in relation to K (*i.e.* $\mathrm{mod}(N_s, K) \ll K$) then, again assuming moderate task length variance σ^2 in relation to w_μ, the CPU node completing the most computational work will contain one more task than $\lfloor(\frac{N_s}{K})\rfloor$. We account for this additional task in our model by adding the mean task length w_μ (our additional task) to the mean cumulative work done, adjusted by the number of CPU worker nodes that are assigned an additional task such that they must complete $\lfloor(\frac{N_s}{K})\rfloor + 1$ tasks in total. This models the fact that the difference between CPU_s and CPU_μ will be greater when fewer worker nodes are assigned $\lfloor(\frac{N_s}{K})\rfloor + 1$ tasks to complete since the true mean work done per CPU will be close to $w_\mu \cdot \lfloor(\frac{N_s}{K})\rfloor$ when many nodes are completing only $\lfloor(\frac{N_s}{K})\rfloor$ tasks. The difference between CPU_s and CPU_μ is therefore essentially linear in mean task length w_μ once N_s, K and σ are known. Intuitively, if $\mathrm{mod}(N_s, K)$ is low but non-zero (*e.g.*) equal to one, then the single CPU that is assigned this extra task will be required to complete almost exactly one extra task length of work in comparison to the mean amount of work $CPU_\mu \approx w_\mu \cdot \lfloor(\frac{N_s}{K})\rfloor$. As $\mathrm{mod}(N_s, K)$ grows, the value representing the mean amount of work done per CPU is adjusted accordingly. The special case where $\mathrm{mod}(N_s, K) = 0$ we expect, as discussed previously, only adds a constant amount of excess work above the mean for large N_s similar to the case explored previously using an unbounded K (see Section 3.6.1). We validate this model using empirical simulation data for various K and task length w_μ. A sample of these simulation and model prediction results, exploring simulated and predicted times for various K are found in Figure 3.



(a)

(b)

Figure 3: (a) We plot the model of the mean work we expect each CPU to carry out CPU_μ (blue line) in terms of overall (log-scale) computation time units for varying K processors. We show using empirical simulation (red line) how the longest CPU queue CPU_s deviates from this value in practice in relation to N_s and K. Our model prediction of the maximum work carried out by a CPU: 'CPU_s Model' (circles plotted for every 10th K value) exhibits how our model is able to account for this. Here we show a simulation distributing $N = 250$ tasks over one superstep with a mean task length of $w_\mu = 1000$, $\sigma = 1$. (b) The difference found between model prediction of CPU_s and empirical simulation for each value of $K \in \{1..250\}$. We exhibit model prediction error of < 10 time units (Y-axis) when using a mean job length $w_\mu = 1000$ units for each value of K explored. Our prediction makes small periodic errors but this error reduces further as K increases. For the number of CPUs that we make use of in practice (*e.g.* > 20) we see an overall computation time prediction error of < 4 time units when using $w_\mu = 1000$ units.

4. Hybrid BSP model predictions

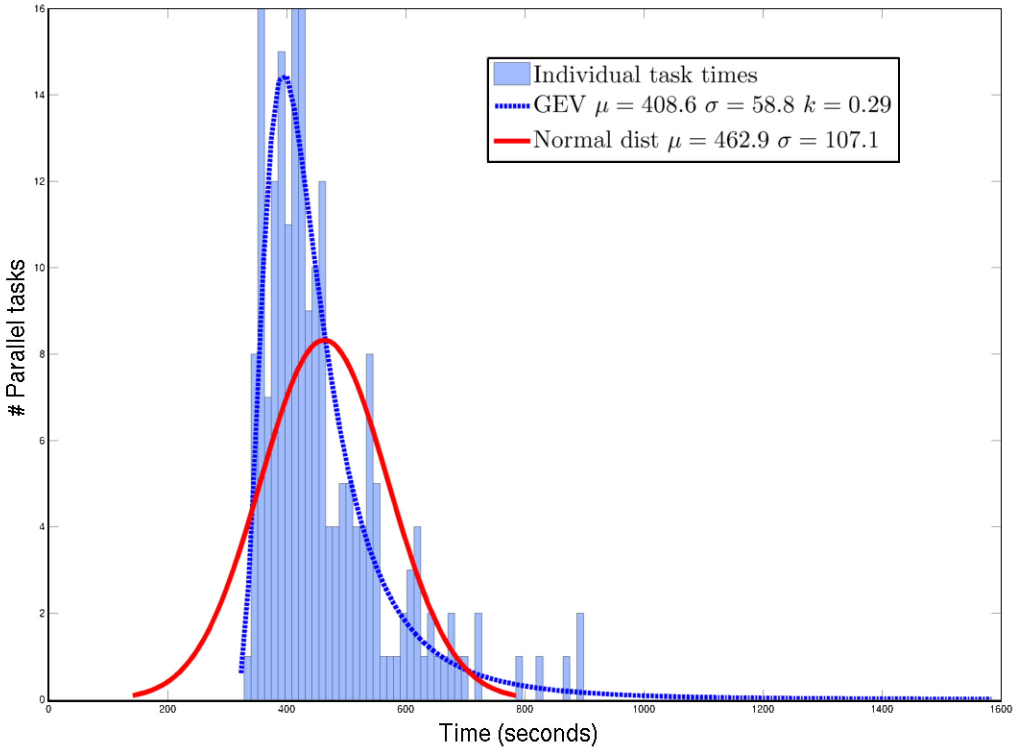
In this section, we use our hybrid BSP model (introduced in Sections 3.6.1 and 3.6.2) to predict the expected run time of real-world applications that we distribute to our SGE cluster under our task farming framework. We present the results of submitting jobs under real network and Grid Engine loading conditions and compare job timing results with our predictions to test the validity of the models developed in Section 3.6.

We submit various application configurations to our SGE cluster that involve distributing $N_s = 20$, 40 and 100 tasks during each superstep in applications making use of $S = 5$, 10 and 30 supersteps. The applications that we utilise for testing our model contain parallel tasks with cost durations of comparable length by design. Details of the applications we experiment with are given in Section 5. To calculate true overall application time cost we record individual parallel task run times and are therefore able to find the longest running (highest cost) task within each superstep. We then sum the times required for the longest running task w_s in each superstep s such that $\sum_{s=0}^{S} w_s + Sl$ provides the total time needed to execute the parallel application in practice, assuming that all tasks within a superstep are able to run in parallel. With regard to the sample applications that we investigate during this experiment we find that the time cost for the *barrier synchronisation* steps l are negligible in practice and therefore we neglect these in the runtime calculation. Although barrier synchronisation is negligible in the sample application investigated here, we note that this is certainly not always the case and we therefore choose not to oversimplify the model.

We perform repeated trials ($n = 10$) for each application configuration tested. Here we provide detail of a configuration distributing $N_s = 20$ tasks during each of 10 supersteps as an example. In this example, we measure mean total real-world cost to be $\sum_{s=0}^{9} w_s = 123.06$ minutes of parallel computation time with an average task length of $w_\mu = 462.9$ seconds ($\sim$ 8 minutes), and a mean parallel task length standard deviation of $\sigma = 107.13$ seconds. The recorded individual task times, across all 10 supersteps from one trial, are shown in Figure 4. Examining the real-world run times of the distributed tasks highlights a slightly heavy-tailed distribution for the particular application employed in this experiment. This typically results in several long runtime outliers that contribute to the total runtime cost using our overall runtime calculation method. For expository purposes we also fit a GEV (Generalised Extreme Value) model to the data here, providing a reasonable fit (*i.e.* resulting in a slightly lower BIC value of 2343.39 compared to the Gaussian BIC of 2446.78 for this data set). In future work we plan to re-examine our hybrid model using (*e.g.*) a GEV distribution in place of our current Gaussian timing model to predict run times in cases where this provides a better fit to the independent task times. We also note that one potential route towards accounting for heterogeneous participating processors p during runtime prediction would involve making use of mixture distributions (*e.g.* a mixed GEV distribution). We leave more sophisticated task time distribution fitting to future work. We obtain individual runtime costs by profiling the application (detailed in Section 5.1) through the use of the Matlab function `cputime`. By additionally including Sun Grid Engine queueing (non-working) time, mean wall-clock time for the application run in this example was 173.46 minutes (non-working time is attributed to sharing the SGE cluster with other users).

Using the distributed task model that we introduce in Equation 5, and assuming that we have sufficient participating processors K to accommodate 20 tasks in parallel, we predict the maximum work performed by a single processor in a superstep to be $CPU_s = 669.86$ seconds for this example (an underestimation, the mean value found in practice across $n = 10$ trials for this configuration is 738.37 seconds of CPU time). Using $S = 10$ supersteps the total runtime predicted by our model for this experiment is therefore 111.6 minutes. This results in a slight underestimation of the true mean total cost by 11.4 minutes ($\sim$ 10%) for this distributed configuration. This underestimation is probably explained by the slightly non-Gaussian distribution observed in Figure 4. Results for the predicted and measured job completion times for the distributed configurations investigated in this way are summarised in Tables 1 and 2. In Table 2 we present measured and predicted **overall computation time** and note that the difference between measured time and our model prediction is always within 11% of the true value. Our approximate model provides a simple yet moderately accurate method for predicting the amount of computational work required by applications

Figure 4: Individual parallel task timings across all 10 supersteps from one trial.



formulated under our task farming framework and distributed to Sun Grid Engine, or other queue based, cluster systems. For completeness we contrast the computational time required to mean wall-clock time used by the cluster in practice. We note in general wall-clock time is significantly larger than required computational time however we find that wall-clock time is subject to high variance between trials as we have little control over multi-user cluster wall-clock time. This is due to the queueing aspect of sharing the SGE cluster with other users.

Table 1: Parameter sets used for four different sets of distributed application experiments varying the number of distributed tasks (N_s) and supersteps (S).

	# CPU nodes (K)	Tasks per superstep (N_s)	Supersteps (S)
Model prediction (eq. 5)	20	20	10
Measured timing set 1	20	20	10
Model prediction (eq. 5)	20	20	30
Measured timing set 2	20	20	30
Model prediction (eq. 5)	20	40	05
Measured timing set 3	20	40	05
Model prediction (eq. 5)	20	100	05
Measured timing set 4	20	100	05

5. Example semi-synchronised task farming applications

We introduce three computationally demanding computer vision problems and propose solutions implemented using our semi-synchronised task farming framework. We focus on simple farming applications

Table 2: Distributed application measured timing results and BSP model predictions for four sets of distributed tasks with rows corresponding to Table 1. We obtain the predicted overall computation time by taking the product of the predicted w_S and the number of supersteps (S). The difference between our overall computation time model predictions and measured results are always within 11% of the true value.

	True w_μ (sec)	Task time σ	Predicted w_S (eq. 5) and True w_S (sec)	**Overall computation time** (min)	Wall-clock time (min)
Model prediction (eq. 5)	N/A	N/A	$(462.0 + 207.86)$=669.86	(669.86 sec $\cdot$10) = 111.6	N/A
Measured timing set 1	462.0	107.13	738.37	123.06	173.46
Model prediction (eq. 5)	N/A	N/A	$(348.17 + 168.02)$=516.19	(516.19 sec $\cdot$30) = 258.1	N/A
Measured timing set 2	348.17	86.60	740.0	287.4	434.08
Model prediction (eq. 5)	N/A	N/A	$(57.1 + 19.8)$=76.9	(76.8 sec $\cdot$5) = 6.40	N/A
Measured timing set 3	57.1	8.95	91.3	6.89	41.3
Model prediction (eq. 5)	N/A	N/A	$(214.4 + 96.46)$=310.86	(310.86 sec $\cdot$5) = 25.9	N/A
Measured timing set 4	214.4	37.83	353.6	27.3	133.0

that are able to benefit from performing many tasks in parallel yet require some form of communication between rounds of parallel tasks (supersteps). As described previously, these parallel task sets and synchronisation steps make up a larger computational process. The example applications that we study here all share the following properties:

- Large input data set. Our input data sets are large relative to the number of model parameters and control options that dictate the data processing procedures.
- Large number of tasks. The number of tasks N that make up the overall computational process is large and may not be known in advance. Each application launches sets of tasks that are processed in parallel. All tasks in a synchronised superstep must complete before the following round of tasks can begin. Task parameters are defined by fixed model parameters and potentially information resulting from the completion of previous task sets.
- Task independence. Each task is defined by model parameters, the global input data and potentially the task set results from the previous superstep. For tasks that are contained in *the same superstep*, no dependencies exist between superstep members.

5.1. Application 1: Multi-view point cloud registration

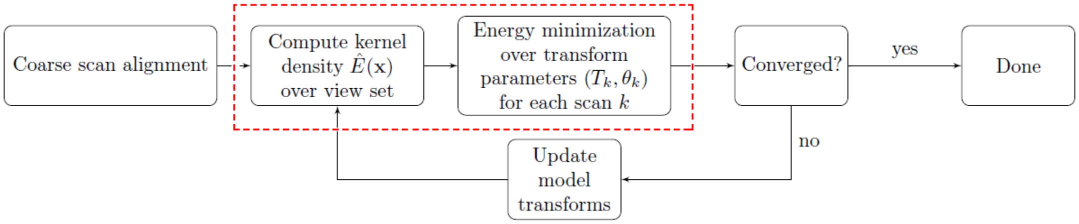
5.1.1. Multi-view registration

3D surface registration can be considered one of the crucial stages of reconstructing 3D object models using information obtained from range images captured from differing object viewpoints. Point correspondences between range images and view order are typically unknown. Aligning *pairs* of these depth images is a well studied problem that has resulted in fast and usually reliable algorithms. The generalised problem of globally aligning *multiple* partial object surfaces is a more complex task that has received less attention yet remains a fundamental part of extracting complete models from multiple 3D surface measurements for many useful applications such as robot navigation and object reconstruction. This is the *multi-view* registration problem.

Early solutions to the multi-view registration problem typically proposed defining one view position as an anchor point and then progressively aligning overlapping range scans in a pairwise fashion such that applying the rigid transforms found at each pairwise step in a chain brings each additional viewpoint into the coordinate frame of the anchor scan, thus obtaining a complete object model. Although straightforward and fairly computationally inexpensive, this technique often results in registration error accumulation and propagation. In an attempt to address this issue more recent work [30, 31, 32] proposes various techniques for aligning all surface viewpoints *simultaneously* in an attempt to reduce errors and make use of information from all views concurrently. Performing view registration in this fashion is typically able to improve alignment quality by distributing registration errors evenly between overlapping range views. Considering all views simultaneously does however typically incur increased computational cost as these approaches must, at each iteration, compute the registration error between each range view and some form of reference. A solution to the multi-view registration problem, capable of handling large data sets, consisting of many viewpoints, therefore provides a good candidate for a parallelised implementation.

In this paper we present our approach for the simultaneous global registration of depth sensor data from many viewpoints, represented by multiple dense point clouds [33], implemented in the *Semi-synchronised task farming* framework described in Section 3. This framework allows us to process large numbers of range images per object reconstruction whilst retaining the accurate high quality view alignment results typical of simultaneous registration approaches.

Figure 5: Our multi-view registration method. Stages of the algorithm within the dashed line area are distributed to our cluster in parallel.


Coarse scan alignment
Compute kernel density $\hat{E}(\mathbf{x})$ over view set
Energy minimization over transform parameters (T_k, θ_k) for each scan k
Converged?
yes
Done
no
Update model transforms

5.1.2. Simultaneous registration using task farming

Given many partial object views represented by point clouds with a typical set of seed positions providing a coarse alignment initialisation, we construct a kernel-based density function of the point data to determine an estimation of the sampled surface. Using this surface estimate we define an energy function that implicitly considers the position of all viewpoints simultaneously. We use this estimation of the sampled surface to perform an energy minimisation in the scan pose transform space, on each scan in parallel, to align each viewpoint to the object surface estimate and implicitly, to each other. After alignment, we recompute the energy function and then re-minimise all scan positions. This process is repeated to convergence. Figure 5 outlines this approach, for more details see [33].

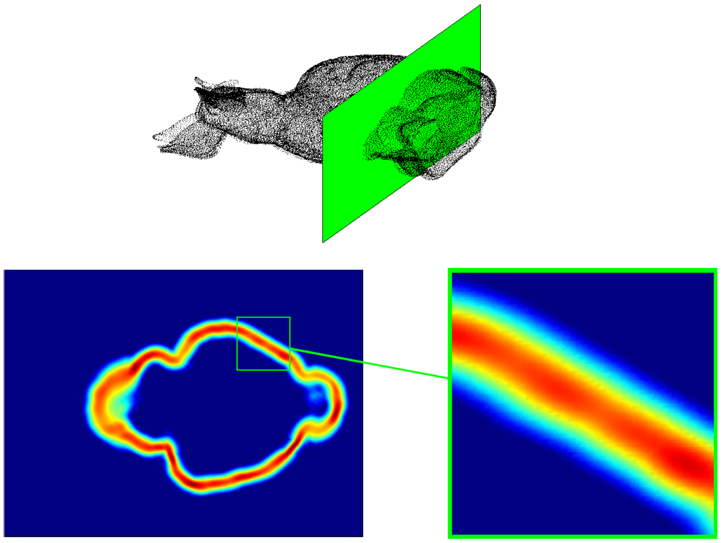
Since range viewpoints are aligned in parallel we are able to accommodate many view sets without increasing the wall-clock time, unlike typical serial solutions. Utilising many object viewpoints affords benefits over sparse sets of views for the task of object reconstruction such as better object surface coverage, hole filling and reconstructed object detail improvement.

For N view-points we define N independent parallel tasks in each superstep and in each of these tasks we use the current pose of the remaining $N-1$ scans for the purpose of computing a surface estimate and a related energy function. We allow the final, active scan to move in the transform space by searching for optimal pose parameters. Each parallel task assigns a different view-point as the active scan. Independently evaluating the position of each moving scan in relation to the inferred surface and therefore minimising our energy function brings the active view into better alignment. After this minimisation has taken place for each viewpoint in parallel, we have N sets of optimal rigid transform parameters; 3 translation (θ_x, θ_y, θ_z) and 3 rotation (θ_α, θ_β, θ_γ) parameters that bring each view into alignment with the estimated surface (and therefore the other views). Once each independent task has found a set of rigid transform parameters (reached the superstep synchronisation barrier), we apply the transform parameters found for each view, thus bringing the entire set into better alignment with one another, completing our barrier synchronisation step. We then redistribute the tasks to perform a re-estimation of the sampled surface, using the new view-point positions, for each view in parallel. This typically results in a tighter, more accurate, estimation of the surface. We iterate this process for S supersteps until viewpoint registration convergence has been reached. Convergence can be identified by looking at residual point alignment error or the magnitude of the transforms being found by each task optimisation. In practice convergence is usually reached within $S = 10$ supersteps however for the purposes of the timing experiments in Section 4 we use up to $S = 30$ supersteps.

This optimisation algorithm can be summarised as follows: we define $\{V_i\}$ as the set of N individual point sets V_i and S_i as the collective surface estimate found using the points in point sets V_j where $j = 1 \ldots N$ and $j \neq i$. We define our energy function $E(\cdot)$ to evaluate the alignment of 3D points $x \in V_i$ in relation to surface estimate S_i. Therefore $E(V_i, S_i)$ evaluates the current pose of point set V_i in relation to how well registered it is with surface estimate S_i. We perform minimisation in the transform space of V_i, evaluating how well the viewpoint is aligned to our surface estimate S_i at each iteration step. This minimisation lets us find optimal pose parameters θ_i for each V_i in parallel. We use these parameters to apply pose transformations T_{θ_i} to each point set V_i. This transform optimally aligns point set V_i with the related current surface estimate. In parallel we align each point set V_i to the surface estimate provided by S_i. By doing this we implicitly register each viewpoint with all others. We then re-estimate S_i from the resulting new poses of $\{V_i\}$, and iterate this process to convergence. This algorithm is described using the following pseudocode:

```
Input: Range scans $V_1, \ldots, V_N$
begin
  converged := 0
  while (NOT converged)
        parallel for i=1 ... N
           $S_i = \text{estimate\_surface}(\bigcup_{\substack{j=1 \\ j \neq i}}^{N} V_j)$
           $\theta_i = \arg\max_{\theta} E(T_\theta(V_i), S_i))$
        end
        parallel for i=1 ... N
           $V_i = T_{\theta_i}(V_i)$
        end
        converged = test_convergence($V_1, \ldots, V_N$)
  end
end
```

Figure 6: Top: A planar slice of our energy function through coarsely aligned partial scans (Stanford bunny data set) Bottom: our energy function approximating the underlying surface defined by the coarsely aligned range scans. A zoom of the slice region shows surface function values that are represented by colours increasing from deep blue to red. We align each partial view point cloud with this surface estimate in parallel.



5.1.3. Experimental setup

We evaluate this parallel alignment strategy quantitatively on synthetic and real range sensor data where we find that we have competitive registration accuracy with existing frameworks for this task. See [33] for registration accuracy results. Here we evaluate application speed up due to parallelisation. As discussed we are able to register all views simultaneously by taking advantage of many cluster nodes, and thus distribute the work. Here we explore various distributed *task* and *superstep* configurations and look at the performance gained by making use of a distributed system compared to performing the work on a single node. In the case of the single CPU experiments we register each scan serially using an individual cluster node and then find the related surface estimates once rigid transforms have been found for all scans. Figure 6 shows a partial example midway through alignment.

We record runtime results as follows: for Single CPU results no job queueing is involved as the algorithm performs the registration of each scan in series until completion. The time reported is the total time required to register N viewpoints in series over S supersteps. For the parallel distributed experiments we measure the time taken in two ways. As discussed in Section 3.1, the distributed system we make use of employs a multi-user job queueing system. Firstly we measure the wall-clock time by recording the total real-world time required from the point of submitting our work to the job queue until the job is complete (when the registration of all viewpoints V_i has converged in this case). Here job queueing (non-working) time cost may be incurred by each individual distributed task, (the alignment of a single view V_i to the related surface estimate to find the optimal pose transform T_{θ_i}). In Table 3 this timing result is referred to as “ECDF wall-clock time”. The second distributed timing measure excludes this queueing (non-working) time and for each superstep finding the maximum task length of an individual distributed task (scan alignment) in a similar measurement process to that outlined in Section 4. The time reported for this second metric is then the sum of the maximum task lengths over the total number of supersteps, we call this the “Distributed ideal time”. We consider this to be an accurate assessment of the computation time required, as each superstep must wait for all member distributed tasks to finish before it may apply the global synchronisation step and then launch the following set of distributed tasks. This second metric excludes real-world queueing time. Furthermore, for this experiment, we have sufficient worker nodes to process all distributed tasks in a superstep concurrently (true in the case of our current HPC cluster). These measurements allow us to compare the optimal theoretical performance gain to real-world speed up, achieved in practice on our multi-user system.

5.1.4. Performance evaluation

The success of employing an HPC system to solve computationally demanding problems resulting from large real-world data sets depends on the system architecture (*e.g.* number of available processors) and algorithmic design. The performance of an algorithm on an HPC system can be evaluated by calculating the speedup provided over a single node or single CPU system. Here we use speedup S_p and efficiency E_p (Equations 6 and 7) to show the improvement we achieve by formulating computer vision problems under our task farming framework. Assuming that the speed of processors and the network is constant; then speedup [34, 35] is often defined as:

$$S_p = \frac{T_1}{T_p} \tag{6}$$

where p is the number of participating processors, T_1 is the computational time needed for sequential algorithm execution and T_p is the execution time required by the parallel algorithm when making use of p processors. Ideal (linear) speedup is obtained in the case $S_p = p$. Although super linear speedup is possible in some cases (*e.g.* due to cache effects in multi-core systems), when using task farming and an HPC cluster we consider linear speedup as ideal scalability. In the linear speedup case, doubling the number of processors p will double the speedup S_p (halving the required execution time T_p). The second, related performance metric we make use of is efficiency (Equation 7). The E_p metric, typically in range [0..1] attempts to estimate how well utilised p processors are when solving the problem at hand compared to how much time is spent on activities such as processor communication and synchronisation.

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \tag{7}$$

For our viewpoint registration algorithm Table 3 shows that, in experiments performing only a single superstep (surface estimation), when we compare the serial and distributed computation times (excluding job queueing time) we are able to achieve significant speed up in each case (where here $p = 5, 20$ and T_1, T_5 and T_{20} timings are in minutes) with $S_5 = \frac{37.26}{8.74} = 4.26$ and $S_{20} = \frac{95.38}{7.74} = 12.32$. We note that the experiment aligning fewer viewpoints, using fewer nodes ($|\{V_i\}| = 5$, $p = 5$, $S = 1$) achieves a result closer to optimal speedup (and efficiency). We reason that a longer maximum task time (the superstep time) is likely to be observed for the larger experiment ($|\{V_i\}| = 20$, $p = 20$, $S = 1$) as it contains more distributed tasks per superstep. This point holds in practice here and was explored during our predictive model formulation and related scalability experiments in Section 3.3. Table 3 also shows the same task set sizes ($|\{V_i\}| = 5, 20$) but with multiple supersteps ($S = 5$), which achieve slightly improved speedup and efficiency performance: $S_5 = \frac{176.06}{39.12} = 4.50$ and $S_{20} = \frac{835.02}{52.40} = 15.94$. Again our hybrid model predictions come within 10% of the measured values in each case and we include ECDF wall-clock time results in the distributed experiments for completeness. The time required to align 20 range image viewpoints over 5 supersteps using our simultaneous method can be effectively reduced from ~ 14 hours to fifty minutes.

Table 3: Multi-view registration algorithm timing results: single CPU vs distributed cluster.

	Single CPU (min)	Distributed ECDF wall-clock time (min)	Distributed ECDF ideal time (min)	Model prediction (min) (Eq. 5)	S_p
5 views 1 superstep	37.26	10.77	8.74	8.37	4.26
20 views 1 superstep	95.38	10.89	7.74	8.28	12.32
5 views 5 supersteps	176.06	49.22	39.12	36.06	4.50
20 views 5 supersteps	835.02	185.94	52.40	49.37	15.94

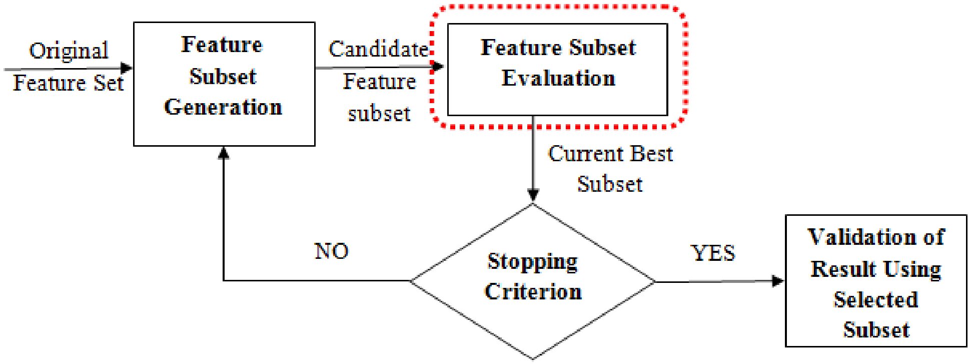
5.2. Application 2: Feature selection

5.2.1. Feature selection for classification

The aim of feature selection in computer vision and pattern recognition problems is to obtain a small subset of a larger full set of features which gives *e.g.* accurate classification. The benefits of feature selection are to reduce the dimensionality of data which decreases the classification time and decreases the chance of over-fitting during training. Besides, it is important to eliminate irrelevant, redundant features and even the features which might cause inaccurate classification. Popular computer vision applications which utilise feature selection are face recognition [36], trajectory analysis [37], image segmentation [38], gesture recognition [39], and medical image processing [40]. In general, feature selection consists of feature subset generation, feature subset evaluation, a stopping criteria and validation of results using the selected final subset [41, 42].

Feature subset evaluation can be in terms of a criterion such as maximising a performance criterion. The iterations continue until the value of the performance criterion is accepted which is often when adding additional features reduces performance. Feature subset generation can be divided into two categories [43]; filters and wrappers. The filter approaches do not use a learning algorithm and are usually faster and computationally efficient. Filter approaches rank the features and evaluate them in terms of their goodness / relevance such as using distance, consistency, and mutual information between a feature and the class labels [44]. On the other hand, wrapper methods use a learning algorithm to evaluate the quality of the feature subset. Wrappers are usually superior in accuracy when compared to filters [45]. In this study, we use the Sequential Forward Feature Selection (SFFS) algorithm (Section 5.2.2) which is a wrapper method with a parallel schema that suits the semi-synchronised task farming framework that we have introduced (Section 3, Figure 1).

Figure 7: Steps of feature selection (adapted from [42]). The dashed box contains the stages where we evaluate the candidate feature subsets independently and in parallel, using our task farming framework.


Original
Feature Set
Feature Subset Generation
Candidate Feature subset
Feature Subset Evaluation
Current Best Subset
NO
Stopping Criterion
YES
Validation of Result Using Selected Subset

5.2.2. Sequential Forward Feature Selection

The forward feature selection procedure begins with an empty feature subset. In the first iteration, it initialises the feature subset by trying features one by one and evaluates the subset in terms of the performance criterion. At the end of the first iteration, the first best feature is selected. In subsequent iterations, the subset of features that is selected in the previous iteration is extended by one of the remaining features. Hence, in the second iteration the feature subsets have two features to be evaluated. After all feature subsets are evaluated, the current best result of the new subset is compared with the previous iterations best result and, using the stopping criteria, a decision is made to continue to a third iteration or to stop selecting features in order to validate the results. If the decision is to continue, then a similar procedure iterates to produce three features in the candidate subset for the third iteration, four features for the fourth iteration and so on.

5.2.3. Sequential Forward Feature Selection (SFFS) using task farming

Similar to many other wrapper approaches, the SFFS procedure is computationally expensive especially if the number of features is large, the learning algorithm has a high time complexity and the required number of iterations is large. Therefore, efficient implementations of this method are needed for many computer vision applications. The procedure that we use to accelerate SFFS is based on the semi-synchronised task farming framework that we present above (See Figure 7: the dashed box shows where we apply task farming). In this context, in each superstep, we first build the subsets and then distribute each subset as a parallel task to be processed using the learning algorithm. After all the distributed tasks finish (the superstep conclusion) we collect them to find the current best criterion value and the feature corresponding to it. The new best feature is selected and becomes a member of all following feature subsets. During this task synchronisation stage, we also apply the stopping criteria to decide if we are going to continue to select features or not. If the decision is to continue, the new feature subsets are built and the new tasks are distributed. At the following iteration, the number of distributed tasks is one less than the previous iteration. This distribute-and-collate procedure continues until the value of the performance criterion decreases compared to the previous iteration. When this value decreases the decision to stop expanding the feature subset is made and the SFFS process is complete. A formal description of our SFFS algorithm using semi-synchronised task farming is as follows:

```
Input: N features $\{f_i\} = F$, Evaluation function $E$
Output: The selected features $S$, $S \subseteq F$
begin
```

```
    converged := 0
    $S := \{\}$
    while (NOT converged)
            parallel for $f_i \in F$
                evaluate $e_i = E(S \cup \{f_i\})$
            end
            select $j = \arg\max_i(e_i)$
            $S = S \cup \{f_j\}$
            $F = F \setminus \{f_j\}$
            converged = $E(S) \overset{?}{\leq} E(S \setminus \{f_j\})$
    end
end
```

5.2.4. Experimental setup

The presented feature selection procedure, formulated under our task farming framework, was tested using a fish trajectory dataset which has 3102 trajectories in total. In this dataset 3043 trajectories are normal (show typical behaviour) while 59 of them are rare behaviours. There are in total 179 trajectory description features which are obtained from the curvature scale space [46], moment descriptors [47], velocity, acceleration, angle, central distance functions [46] and vicinity [48] etc. of trajectories. The aim is to select the feature subset which can best distinguish normal and rare trajectories with high class accuracy. The learning algorithm that we utilise is based on affinity propagation and class labels (see [37] for details). The experiments were performed using 9-fold cross validation which constructs the training and testing sets randomly while maintaining an even distribution of normal and abnormal trajectories between folds. Table 4 displays the best feature subset performance after a new feature is selected in each iteration. The performance metric is the average trajectory class classification accuracy. The total number of features that were chosen for each fold were 3,2,2,6,2,5,2,3 and 2 respectively and feature selection stops when the observed average classification accuracy is lower than the previous superstep (iteration). The final (best) criterion value for each fold are shown by shaded cells in Table 4.

Table 4: The results of applying distributed Sequential Forward Feature Selection to a 9-fold real-world fish trajectory dataset. The table shows average trajectory-class classification accuracies during training for the best performing feature subset of each length, for each fold. Shaded values show the best criteria value found for each fold and the following criteria value (to the right of the best value) shows the value found when an additional feature is added (producing a lower criterion value by definition, hence the algorithm terminates).

	Feature subset cardinality (# Supersteps)	1	2	3	4	5	6	7
Fold								
1		0.9467	0.9482	**0.9497**	0.9305			
2		0.9527	**0.9689**	0.9586				
3		0.9305	**0.9749**	0.9734				
4		0.8677	0.8841	0.9169	0.9481	0.9585	**0.9588**	0.9567
5		0.8649	**0.9586**	0.9481				
6		0.9567	0.9675	0.9704	0.9734	**0.9749**	0.9689	
7		0.9438	**0.9689**	0.9585				
8		0.9201	0.9689	**0.9808**	0.9567			
9		0.9645	**0.9822**	0.9438				

To evaluate the speed and efficiency of our distributed SFFS algorithm using our task farming framework we compare it to sequential SFFS performed on a single compute node and again make use of speedup and efficiency metrics (Section 5.1.4). We test both implementations by varying the total feature pool size $\in \{10, 20, 50, 100, 179\}$ and cap the number of potential new features added to the optimal feature subset by limiting the number of superstep (feature selection) rounds to 2, 6 and 10.

During each feature selection superstep, we employ the learning algorithm: affinity propagation and class labels (see [37] for details). The results are presented in Table 5 in terms of processing time (minutes). We compare the results obtained using a single CPU (sequential SFFS) to the distributed SFFS implementation again recording both the case including SGE queueing (ECDF wall-clock time) and the case where it is disregarded (Ideal ECDF time). Discounting the SGE queueing time effectively assumes that we have a sufficient number of cluster nodes available to process all feature subset tasks in parallel.

Table 5: Feature selection algorithm training time results (in minutes): single CPU vs distributed cluster. Our timing model accurately predicts expected ideal distributed time and we again display large speedup S_p gains over the single CPU implementation. The difference between predicted and measured time grows for the large feature set experiments (*e.g.* 100,179) where we gain the largest speedup S_p. One application specific cause for this discrepancy involves the particular image processing features extracted. When experimenting with more features (100,179) we include the extraction of computationally expensive image features that result in long individual task times. These outliers do not significantly effect superstep mean task length w_μ but do however increase the ECDF ideal time by providing large w_s. Re-examining our hybrid model with a non-Gaussian individual task time distribution may help to improve these estimates. We again include wall-clock time for completeness.

	Single CPU (min)	Distributed ECDF wall-clock time (min)	Distributed ECDF ideal time (min)	Model prediction (Eq. 5) (min)	S_p
10 features 2 superstep	162	31	19	18.45	8.53
10 features 6 supersteps	412	75	55	56.14	7.49
10 features 10 supersteps	322	153	132	156.32	2.44
20 features 2 superstep	323	35	18	18.01	17.94
20 features 6 supersteps	888	113	86	76.40	10.33
20 features 10 supersteps	951	211	172	184.36	5.53
50 features 2 superstep	1045	79	45	30.91	23.22
50 features 6 supersteps	1975	217	123	93.70	16.05
50 features 10 supersteps	3111	526	248	249.11	12.54
100 features 2 superstep	1749	132	60	33.80	29.15
100 features 6 supersteps	4023	417	170	107.22	23.66
100 features 10 supersteps	6493	957	303	208.53	21.43
179 features 2 superstep	2548	314	189	76.24	13.48
179 features 6 supersteps	6788	1027	276	233.53	24.60
179 features 10 supersteps	11712	2354	436	380.40	26.86

5.2.5. Performance evaluation

The results in Table 5 show that formulating this problem under our task farming framework is again worthwhile, speeding up the completion times of our SFFS application significantly. This is especially true in the cases where the cardinality of the total feature pool (number of parallel tasks) is large *i.e* where $F =$ 50, 100, and 179. The single CPU implementation is slower than distributed SFFS in every case, even when taking into account the SGE queueing time. The performance of our distributed SFFS implementation achieves a speedup of $S_p \in [2 \ldots 30]$ (see Table 5) over the serial timings with the assumption that sufficient compute nodes are available to process all distributed tasks in parallel. When the SGE queueing time is included we achieve $S_p \in [2 \ldots 13]$ (not shown). In practice this allows us to evaluate a feature set containing *e.g.* 179 features to find an optimal feature subset during training for the purpose of fish trajectory classification in ~ 7 hours (excluding queueing time) in comparison to the corresponding serial computation that took 195 hours (>1 week) to complete. Determining optimal feature subsets in this way allows us to construct a fish trajectory classification system capable of $> 95\%$ accuracy on over 3000 trajectories during the training stage.

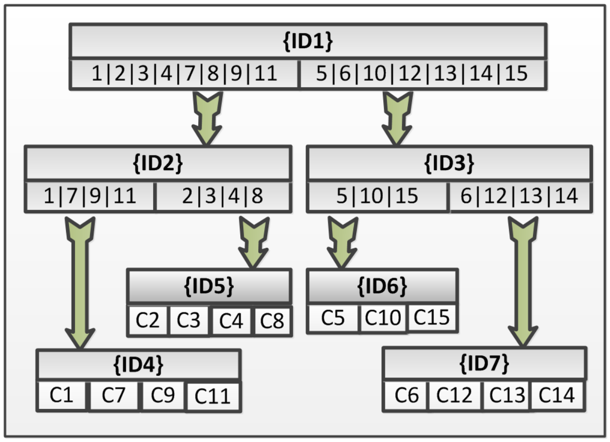
5.3. Application 3: Hierarchical classification

5.3.1. Hierarchical classification method

The final application that we implement under our task farming framework is a hierarchical classification algorithm called the Balance-Guaranteed Optimised Tree (BGOT). The BGOT is a classification method that has been shown to perform well when handling data points originating from imbalanced classes [49]. We use BGOT here for the task of object classification. Using hierarchical classification, data to be classified is pushed down a tree path according to a decision made at each tree node (a classifier) [50, 51]. This effectively narrows down the classes that a sample is believed to belong to. Each tree leaf node represents a single class and a data point reaching a leaf is assigned to that class. During the training phase, the BGOT method selects effective subsets of predefined image features used at each node of the tree with the goal of maximising the mean classification accuracy among classes arriving at that node. This increases the weight of minority and under represented classes.

The BGOT algorithm applies two strategies to help control classification error [52]: 1) apply more accurate classifiers at a higher tree level (earlier) and leave less certain decisions until deeper levels and 2) keep the hierarchical tree balanced to minimise the maximum tree depth. A hierarchical classifier h_{hier} is designed as a structured node set. Nodes are defined as triples: $\text{Node}_t = \{\text{ID}_t,\ \tilde{\text{F}}_t,\ \hat{\text{C}}_t\}$, where ID_t is a unique node number, $\tilde{\text{F}}_t \subseteq \{f_1, ..., f_m\}$ is a feature subset (chosen by a feature selection procedure [53]) that is found to be effective for classifying $\hat{\text{C}}_t$ (a subset of classes). For the classification task we use the m-class SVM classifier [54]. An example classification hierarchy with 15 classes is shown in Figure 8. Each node, identified as ID_t, illustrates the class separation decision $\hat{\text{C}}_t$ made at that node. The example BGOT is capable of classifying 15 classes by making use of 7 classifier nodes and a tree-depth of 3 levels. The first level splits the set of classes into two groups.

Figure 8: A classification tree automatically generated by our BGOT algorithm. The hierarchical classification strategy uses 7 node classifiers to classify 15 classes ($C_1, ..., C_{15}$).



5.3.2. Generating the hierarchical tree

The tree building algorithm chooses the image feature subset that maximises the average classification accuracy for images belonging to the aforementioned two groups. Each class set is then split into two subsets and a new node in the tree is created for each subset. This procedure continues until all nodes contain at most four classes. The automatically generated hierarchical tree (BGOT) chooses the best class set split by exhaustively searching all possible combinations of class splits that maintain a balanced tree (an equal number of classes assigned to each of two child nodes). As a result, there are two parameter sets to search over when building the tree: 1) all possible 2-partitions of the classes at each node, 2) the related optimal feature subset in terms of classification performance. This dual parameter search results in a computationally demanding process and suggests that a parallel approach using our framework would prove advantageous. Parallelising feature subset selection is discussed previously (Section 5.2) so here we focus on the tree construction technique, involving the designation of image classes to tree nodes, that we realise under our task farming framework.

5.3.3. Generating a BGOT using semi-synchronised task farming

In this section, we focus on the part of BGOT generation involving the binary split procedure that finds the best class subset split by exhaustively searching all possible combinations of class subsets. At each non-leaf tree node, the set of classes are split into two groups and a SVM classifier [55] is trained to separate samples between these two groups. Finding an optimal class split is exponentially complex and sensitive to the number of classes. In the example provided there are $\binom{15}{8} = 6435$ possible combinations to divide the 15 classes (at the top level) into two subsets of cardinality 7 and 8 which then require an additional $\binom{8}{4} = 70$ and $\binom{7}{4} = 35$ combinations to split the tree at the following level. On average the classifier quality of a subset split takes over two minutes to evaluate therefore $>$ 250 CPU-hours are required if we wish to run the entire exhaustive evaluation process on a single compute node. This process is therefore a good candidate to make use of our parallel framework.

More formally, our tree generation algorithm can be described as follows:

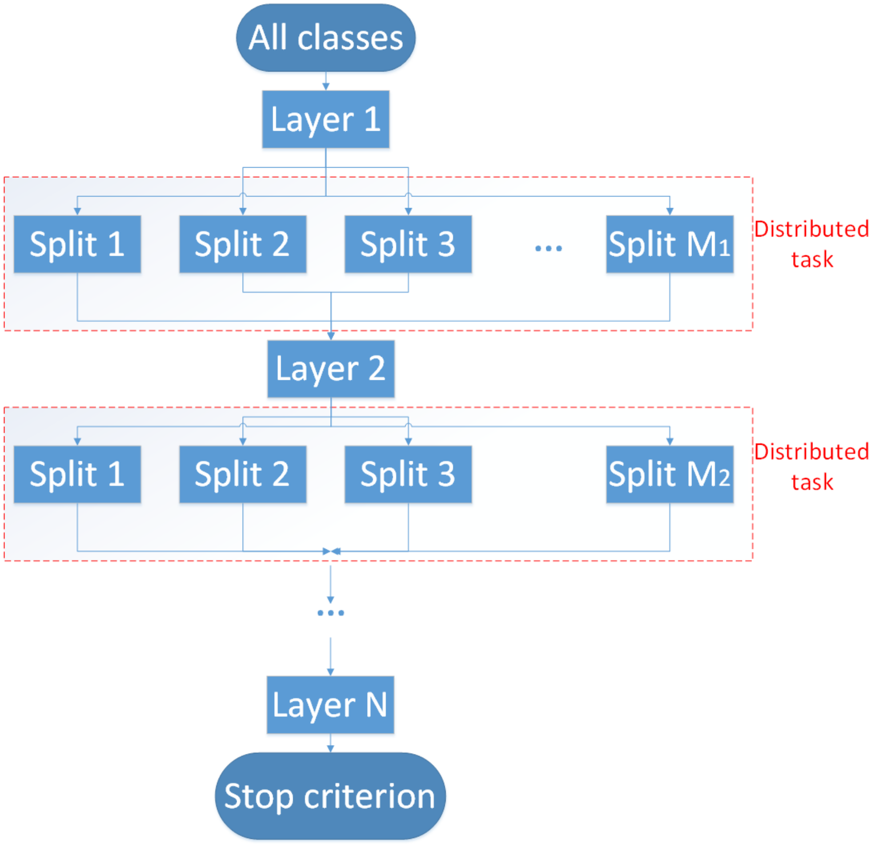
```
Input: class $C_1$ to $C_n$
begin
  $c := \{C_1, ..., C_n\}$
  $level := 0$
  $featureSet :=$ FeatureSelection($c$)
  construct($c, level$)
end
proc construct($c, n$) $\equiv$
  if $n >$ MAXDEPTH
    exit
  end
  comment: Evaluate classification accuracy on each split of classes $c$ in parallel
  parallel for {binary splits of c}
      $r =$ evaluate($c, featureSet$)
  end
  comment: The ChooseSplit function finds the optimal class subset pair based on the set of $r$ evaluations
  $[cLeft, cRight] :=$ ChooseSplit($\{r\}$)
  comment: The maximum leaf node subset size is set to 4 to limit max tree depth
  if size($|cLeft|$) $> 4$
    construct($cLeft, n + 1$)
  end
  if size($|cRight|$) $> 4$
    construct($cRight, n + 1$)
  end
end
```

A schematic of the program flow is illustrated in Figure 9. Firstly the algorithm splits the current set of classes c into all combinations of pairs of disjoint subsets with size $\frac{|c|}{2}$ and then sends each combination to the performance evaluation stage. After evaluating all of the possible splits, the best subset pair, in terms of classification accuracy, is chosen and this split is used to construct two new child tree nodes. This procedure is iterated for both child branches until the stopping criterion is satisfied. Each subset classification accuracy performance evaluation at a given tree level is independent of every other split, and the evaluation tasks do not need to communicate. Furthermore, all tasks have the same work-flow yet have varying input: the subset class member combination. As a result, we find this process a good candidate for our semi-synchronized task farming framework and our HPC cluster. We assign each combination of class set split to a distributed parallel task. Each pair of subsets is then evaluated with an accuracy score in parallel (the accuracy score for each distributed task is found by taking the mean classification accuracy of the two subsets assigned to the task). After all distributed tasks in a superstep have concluded, we collect all of the mean accuracy scores and select the class split with the highest score (our superstep conclusion). Given True Positive and False Negative classifications, the mean accuracy (recall rate) per distributed task is defined as:

$$AR = \frac{1}{|c|}\sum_{j=1}^{|c|}\left(\frac{\text{True Positive}_j}{\text{True Positive}_j + \text{False Negative}_j}\right) \tag{8}$$

where $|c|$ is the number of image classes.

Figure 9: The algorithm to generate our balanced hierarchical classification tree (BGOT). At each tree level, we select the optimal disjoint and balanced class subset split by exhaustively searching all possible splitting combinations. Each set of algorithm stages within a dashed area represents a superstep that is distributed to our cluster in parallel.



5.3.4. Experimental setup

We perform species classification experiments using 6875 fish images with a 5-fold cross validation procedure. The training and testing sets are isolated such that fish images from the same trajectory sequence (containing the same fish) are not used during both training and testing. We extract 66 different image features for the classification task. These features are a combination of colour, shape and texture properties in varying local spatial areas of the fish images such as the tail/head/upper/lower body area, as well as collecting features from the entire fish body area. Sequential Forward Feature Selection (SFFS) is applied to find an optimal feature subset to provide input for the classification task. We use an SVM variant for the classification task. Since SVMs were originally developed for the binary classification problem, we introduce a one-vs-one strategy with a voting mechanism to convert the binary SVM into a multi-class classifier [54]. The mechanism is based on a classify-and-vote procedure. Specifically, each class is trained in a set of binary classifiers against each other class individually. The optimal BGOT result found is shown in Figure 8, where 15 classes are classified using a tree of depth three. See [49] for further species classification details.

5.3.5. Performance evaluation

We explore the computational time requirements for executing our BGOT algorithm in a similar fashion to the previous applications deployed under our task farming framework. The most expensive superstep for this application is (by far) the initial superstep, involving the evaluation of $\binom{15}{8} = 6435$ possible pairs of image class subset splits. This initial step is therefore the section of the application that we focus our timing evaluation on during this experiment. As each subset split takes on average $\sim$ 2 minutes of computational time to evaluate we choose to perform the evaluation of a number of subset combinations in each distributed task. Explicitly we evaluate the time and efficiency performance using experiments involving the distribution of $1, 25, 50$ and 100 tasks in parallel for this large initial superstep. Using 15 image classes, this results in assigning $\frac{6435}{1}$, $\frac{6435}{25}$, $\frac{6435}{50}$ and $\frac{6435}{100}$ subset evaluations to each distributed parallel task during each experiment respectively. We focus here on timing results from the initial large superstep and therefore find that queueing (non-working) time will be minimal and therefore display ECDF ideal time and not wall-clock time in Table 6. We show the ECDF ideal time metric (defined in Section 5.1.3) in Table 6 and note that we are again able to significantly decrease the required processing time in relation to the single computational node case by increasing the number of p processors invoked. By increasing the number of tasks distributed in parallel in the superstep (and therefore reducing the number of subset evaluations assigned to each task) we reduce the ECDF ideal time (and therefore increase our speedup metric) in a near linear fashion achieving speedup metrics of $S_{25} = 14.7130$, $S_{50} = 27.9121$ and $S_{100} = 46.4207$ in practice. While increasing the number of parallel tasks reduces both the ECDF ideal time (and wall-clock time) metrics in the case of the experiments performed here we expect to find a limit to the efficiency of doing this in practice. We see from Table 6 that our efficiency metric (defined in Section 5.1.3) begins to drop as we increase the number of parallel tasks (and therefore processors invoked p). For example, using our current multi-user SGE cluster, it is doubtful that assigning only a single two minute SVM evaluation to each distributed task would provide further improvement as, given that we do not have access to 6435 processors in parallel, queueing time in practice would likely begin to counteract the linear speedup improvement we observe in the experiments performed here. We leave finding the optimal trade-off between speedup and efficiency (*i.e.* the optimal number of image class subset evaluations to assign per distributed task) to future work.

By applying our task farming framework to this problem we are able to effectively evaluate > 6500 BGOT graphs and find the graph configuration that is able to classify 15 species of fish with the highest accuracy. Using our task farming approach reduces the time needed in practice for this evaluation from > 260 hours (using a single compute node) to under 6 hours when making use of an SGE cluster ($p = 100$). By distributing this process with our task farming framework we have been able to easily experiment with and extend our species classification system (*e.g.* to include further fish species) even although this involves BGOT re-evaluation that would prove extremely time-consuming if only a serial implementation were available.

Table 6: We generate BGOTs whilst varying the number of potential graph node subset evaluations per distributed task (node). We are able to improve speedup by increasing the number of participating processors p at the cost of efficiency. The difference between our model predictions and measured computational time costs are within $\sim$ 10% of the true value.

	CPUs (K)	Distributed ECDF ideal time (hours)	Model prediction (Eq. 5) (hours)	S_p	E_p
6435 subset evaluations per node	1	260.42	N/A	1.00	1.00
257 subset evaluations per node	25	18.70	20.89	14.71	0.59
128 subset evaluations per node	50	9.33	10.23	27.91	0.56
64 subset evaluations per node	100	5.61	5.64	46.42	0.46

6. Discussion

In this paper, we formulate a semi-synchronised task farming framework for solving computationally intensive problems where independent problem components can be distributed across an HPC cluster. Results are collated to inform following rounds of task distribution, eventually leading to a global problem solution. Our contributions include the development of a model to predict overall application completion time for problems that are formulated using our framework. We validate this model using simulation and experimental results and find it to be sufficiently accurate, providing a simple tool that can be utilised when planning the time requirements of computationally expensive applications. Further to this we study the performance enhancement obtained by utilising our framework in practice to guide the algorithmic design of several computationally expensive computer vision problems and compare the throughput using our framework with that of solutions making use of only a single compute node. In each example provided we find near linear speedup improvements in the number of participating processors p over the related serial implementations. Also, in the case of each real-world problem investigated, we are able to provide model predictions for computation time that are typically within $\sim 10\%$ of the execution time required in practice.

Based on our experimental results we show that processing large data sets using algorithms formulated with our framework, and deployed on an HPC cluster, obtain significant time saving over single node computation due to vast gains in terms of speedup. We note that in practice the human effort required to move from an original serial algorithm implementation to a distributed task farming application is very reasonable. By making use of SGE to handle the task queueing system and allowing developers to concentrate on domain specific problem aspects we are typically able to completely convert a serial code on the order of days. By also employing user-friendly languages for parallel programming, master-slave communication is also hidden from the developer allowing them to again focus solely on domain specific problems.

Distributed computing on HPC clusters offers an attractive option for our framework when compared to expensive integrated mainframe solutions. The main advantages of HPC clustering include distributed robustness and the ease of cluster scalability. When using an HPC cluster to accelerate the rate that we are able to solve computationally expensive problems the factors of data set size and algorithm design play important roles in determining the degree of success in parallelising an application. Our framework allows the performance of a distributed program on a given architecture to be predictable. Using our framework and simple timing parameters from the algorithm under evaluation allow us to reason about program design at an early stage.

All implementation examples presented in this work make use of Matlab and we find that the prerequisites for writing parallel code under the Distributed Computing Toolbox (DCT) from MathWorks are relatively low. There is no need for the developer to instruct cluster machines how to communicate, which part of the code to execute and how to assemble end results. We find that this provides a straightforward and intuitive approach to parallelising computationally demanding applications in a reasonable time frame. Parallelisation under this simple task farming framework results in potentially huge time savings without requiring extensive task or data parallelism knowledge. Possible extentions and interesting avenues of future work include implementing solutions using our framework with faster compiled languages (*e.g.* C/C++) and applying such solutions to time critical applications. Additionally, extending our performance modelling treatment, to account for heterogeneous processors, would likely improve the model predictive power. Related extentions might take the form of re-examining individual task time fitting using more sophisticated distributions to improve modelling in the heterogeneous processor case (*e.g.* employing distribution mixtures). Finally during the experimental work performed here it was noted that in practice there is often contention between speedup and efficiency. In future we aim to find optimal-trade-off generalisations from the specific cases presented here. In sunmmary this work highlights a range of demanding vision applications that a straightforward parallelisation strategy such as ours can contribute to solving, whilst offering vast computational time savings.

7. Acknowledgement

We thank Murray Cole and Bastiaan Boom for helpful discussion. We also thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work is partially supported by the Fish4Knowledge project, which is funded by the European Union 7th Framework Programme [FP7/2007-2013] and by EPSRC [EP/P504902/1].

References

[1] Silva LM, Veer B, and Silva JG. How to get a fault-tolerant farm. In *World Transputer Congress*, pages 923–938, Aachen, Germany, September 1993.

[2] Casanova H, Kim M, Plank JS, and Dongarra J. Adaptive scheduling for task farming with grid middleware. *International Journal of High Performance Computing*, 13(3):231–240, August 1999.

[3] Casanova H, Obertelli G, Berman F, and Wolski R. The AppLeS Parameter Sweep Template: User-level Middleware for the Grid. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[4] Abdelzaher T, Thaker G, and Lardieri P. A Feasible Region for Meeting Aperiodic End-to-End Deadlines in Resource Pipelines. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 436–445, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Elwasif WR, Plank JS, and Wolski R. Data Staging Effects in Wide Area Task Farming Applications. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

[6] Buyya R, Murshed M, and Abramson D. A Deadline and Budget Constrained Cost-Time Optimisation Algorithm for Scheduling Task Farming Applications on Global Grids. Technical report, Monash University, March 2002. Available at: `http://http://arxiv.org/pdf/cs/0203020.pdf`.

[7] Valiant GL. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[8] Foster I. Task Parallelism and High-Performance Languages. *IEEE Parallel Distrib. Technol.*, 2(3):27–36, Sep 1994.

[9] Hillis WD, Steele J, and Guy L. Data Parallel Algorithms. *Commun. ACM*, 29(12):1170–1183, Dec 1986.

[10] Thain D, Tannenbaum T, and Livny M. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[11] Dean J and Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan 2008.

[12] Isard M, Budiuand M, Yu Y, Birrell A, and Fetterly D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[13] Gentzsch W. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 35–43, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Revenga PA., Sérot J, Lázaro JL, and Derutin JP. A Beowulf-Class Architecture Proposal for Real-Time Embedded Vision. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 8–16, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Cole M. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[16] Skillicorn DB, Hill J, and McColl WF. Questions and Answers about BSP. *Scientific Programming*, 6:249–274, January 1997.

[17] Hammond SD, Mudalige GR, Smith JA, Jarvis SA, Herdman JA, and Vadgama A. WARPP: a toolkit for simulating high-performance parallel scientific codes. ACM, 5 2010.

[18] Mudalige GR, Vernon MK, and Jarvis SA. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14, 2008.

[19] Spooner DP, Jarvis SA, Cao J, Saini S, and Nudd GR. Local grid scheduling techniques using performance prediction. *IEE Proceedings - Computers and Digital Techniques*, 150:87–96(9), March 2003.

[20] Kerbyson DJ, Hoisie A, and Wasserman HJ. Use of predictive performance modeling during large-scale system installation. *Parallel Processing Letters*, 15(04):387–395, 2005.

[21] Frank MI, Agarwal A, and Vernon MK. LoPC: modeling contention in parallel algorithms. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 276–287, New York, NY, USA, 1997. ACM.

[22] Labarta J, Girona S, and Cortes T. Analysing Scheduling Policies using DIMEMAS.

[23] Nudd GR, Kerbyson D, Papaefstathiou E, Perry S, Harper J, and Wilcox D. PACE: A toolset for the performance prediction of parallel and distributed systems. *International Journal High Performance Computing Applications*, 14(03):228–251, 2000.

[24] Adve V, Bagrodia R, Browne J, Deelman E, Dubeb A, Houstis E, Rice J, Sakellariou R, Sundaram-Stukel D, Teller P, and Vernon M. POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems. *Software Engineering*, 26(11):1027–1048, 2000.

[25] S. Pllana and T. Fahringer. Performance prophet: A performance modeling and prediction tool for parallel and distributed programs. *Proc. 2005 International Conference on Parallel Processing ICPP-05, Oslo*, 26(11):509–516, June 2005.

[26] ECDF - The Edinburgh Compute and Data Facility. `http://www.wiki.ed.ac.uk/display/ecdfwiki/`, June 2013.

[27] McColl WF. General purpose parallel computing. *Gibbons AM and Spirakis P. editors, Lectures on Parallel Computation. Cambridge International Series on Parallel Computation*, pages 337–391, 1993.

[28] Hammond SD, Smith JA, Mudalige GR, and Jarvis SA. Predictive Simulation of HPC Applications. In *The IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA-09)*, 2009.

[29] Poldner M and Kuchen H. On implementing the farm skeleton. In *Parallel Processing Letters*, pages 117–131, 2008.

[30] Pottmann H, Leopoldseder S, and Hofer M. Simultaneous registration of multiple views of a 3D object. In *Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 265–270, 2002.

[31] Toldo R, Beinat A, and Crosilla F. Global registration of multiple point clouds embedding the generalized procrustes analysis into an ICP framework. In *3D Proc. Vis. Transmission.*, 2010.

[32] Torsello A, Rodola E, and Albarelli A. Multi-view registration via graph diffusion of dual quaternions. In *IEEE Conf Computer Vision and Pattern Recognition.*, pages 2441–2448, 2011.

[33] McDonagh S and Fisher RB. Simultaneous registration of multi-view range images with adaptive kernel density estimation. *Under review*, pages x–x, 2013.

[34] Baker M, Carpenter B, and Shafi A. MPJ express: Towards thread safe java hpc, submitted to the. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, pages 25–28, 2006.

[35] Eager DL, Zahorjan J, and Lazowska ED. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[36] Yang AY, Wright J, Ma Y, and Sastry S. Feature selection in face recognition: A sparse representation perspective. *UC Berkeley Tech Report UCB/EECS-2007-99*, 2007.

[37] Beyan C and Fisher RB. Detecting abnormal fish trajectories using clustered and labelled data. *Proc. of IEEE International Conference on Image Processing*, 2013.

[38] Roth V and Lange T. Adaptive feature selection in image segmentation. *Pattern Recognition*, (3175):9–17, 2004.

[39] Guo GD and Dyer CR. Simultaneous feature selection and classifier training via linear programming: A case study for face expression recognition. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 346–352, 2003.

[40] McDonagh S, Fisher RB, and Rees J. Using 3D information for classification of non-melanoma skin lesions. *Proc. of Medical Image Understanding and Analysis, Dundee*, pages 164–168, 2008.

[41] Liu H Sr. Toward integrating feature selection algorithms for classification and clustering. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):491–502, 2005.

[42] Wang Q, Li B, and Hu J. Feature selection for human resource selection based on affinity propagation and svm sensitivity analysis. *World Congress on Nature and Biologically Inspired Computing (NaBIC 09), IEEE Press*, (doi:10.1109/NABIC.2009.5393596):31–36, 2009.

[43] Blum A and Langley P. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):245–271, 1997.

[44] Huang J, Cai Y, and Xu X. A filter approach to feature selection based on mutual information. *Proc. of IEEE Int. Conf. On Cognitive Informatics*, pages 84–89, 2006.

[45] Kohavi R and John G. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[46] Bashir FI, Khokhar AA, and Schonfeld D. View-invariant motion trajectory based activity classification and recognition. *Multimedia Systems*, 12(1):45–54, 2006.

[47] Suk T and Flusser J. Graph method for generating affine moment invariants. *Proc. International Conference on Pattern Recognition*, pages 192–195, 2004.

[48] Liwicki M and Bunke H. Hmm-based on-line recognition of handwritten whiteboard notes. *10th Int. Workshop on Frontiers in Handwriting Recognition*, pages 595–599, 2006.

[49] Huang PX, Boom B, He J, and Fisher RB. Underwater live fish recognition using balance-guaranteed optimized tree. In *Computer Vision ACCV 2012*, volume 7724, pages 422–433, 2012.

[50] Duan K and Sathiya KS. Which is the best multiclass SVM method? an empirical study. In *Proceedings of the 6th international conference on Multiple Classifier Systems*, MCS'05, pages 278–285. Springer-Verlag, 2005.

[51] Cai L and Hofmann T. Hierarchical document categorization with support vector machines. *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 78–87, 2004.

[52] Wang YF. and Casasent D. A support vector hierarchical method for multi-class classification and rejection. In *International Joint Conference on Neural Networks, 2009. IJCNN 2009*, pages 3281–3288, 2009.

[53] Weston J, Mukherjee S, Chapelle O, Pontil M, Poggio T, and Vapnik V. Feature selection for SVMs. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 13*, 13:668—674, 2000.

[54] Chang C and Lin C. LIBSVM: a library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):1–27, 2011.

[55] Cortes C and Vapnik V. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.