User Transparent Task Parallel Multimedia Content Analysis

Timo van Kessel, Niels Drost, and Frank J. Seinstra

Department of Computer Science, VU University
De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
{timo,niels,fjseins}@cs.vu.nl

Abstract. The research area of Multimedia Content Analysis (MMCA) considers all aspects of the automated extraction of knowledge from multimedia archives and data streams. To satisfy the increasing computational demands of emerging MMCA problems, there is an urgent need to apply High Performance Computing (HPC) techniques. As most MMCA researchers are not also experts in the field of HPC, there is a demand for programming models and tools that can help MMCA researchers in applying these techniques. Ideally, such models and tools should be efficient and easy to use.

At present there are several user transparent library-based tools available that aim to satisfy both these conditions. All such tools use a data parallel approach in which data structures (e.g. video frames) are scattered among the available compute nodes. However, for certain MMCA applications a data parallel approach induces intensive communication, which significantly decreases performance. In these situations, we can benefit from applying alternative parallelization approaches.

This paper presents an innovative user transparent programming model for MMCA applications that employs task parallelism. We show our programming model to be a viable alternative that is capable of outperforming existing user transparent data parallel approaches. As a result, the model is an important next step towards our goal of integrating data and task parallelism under a familiar sequential programming interface.

1 Introduction

In recent years, the desire to automatically access vast amounts of image and video data has become more widespread — for example due to the popularity of publicly accessible digital television archives. In a few years, the automatic analysis of the content of multimedia data will be a problem of phenomenal proportions, as digital video may produce high data rates, and multimedia archives steadily run into Petabytes of storage space. As a result, high-performance computing on clusters or even large collections of clusters (grids) is rapidly becoming indispensable for urgent problems in multimedia content analysis (MMCA).

Unfortunately, writing efficient parallel applications for such systems is known to be hard. As not all MMCA experts are also parallel programming experts, there is a need for tools than can assist in creating parallel applications. Ideally

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 38–50, 2010.

such tools require little extra *effort* compared to traditional MMCA tools, and lead to *efficient* parallel execution in most application scenarios.

MMCA researchers can benefit from HPC tools to reduce the time they spend on running their applications. However, they will have to put effort in learning how to use these tools. An HPC tool is only useful to MMCA researchers when the performance increase outweighs the extra effort required in using it. *User transparent* tools are special in that these try to remove the additional effort altogether by hiding the HPC complexity behind a familiar and fully sequential application programming interface (API).

It is well-known that applying data parallel approaches in MMCA computing can lead to good speedups [1,2,3,4,5] — even when applied as part of a user transparent tool [6]. In such approaches images and video frames are scattered among the available compute nodes, such that each node calculates over a partial structure only. Inter-node dependencies are then resolved by communicating between the nodes. Previous results, however, show that exploiting data parallelism does not always give the desired performance [7]. Sometimes applications offer not enough data parallelism, or data parallelism causes too much communication overhead for the application to scale well. For such applications using alternative techniques, like task parallelism or pipelining, often is beneficial.

In this paper we describe a user transparent task parallel programming model for the MMCA domain. The model provides a fully sequential API identical to that of an existing sequential MMCA programming system [8]. The model delays the execution of operations and constructs a task graph in order to divide these (sets of) operations over the parallel system. We show that in certain realistic cases our model can be more efficient than a user transparent data parallel approach. Ultimately, we aim to combine our task parallel efforts with our earlier results on a user transparent data parallel programming model [6] to arrive at a model that exploits both data and task parallelism (see also [9]).

This paper is organized as follows. In Section 2, we describe general user transparent parallel programming tools. Section 3 explains the design of our user transparent task parallel programming model. We describe a simple example MMCA application in Section 4. This is followed by an evaluation in Section 5. Finally, we describe our conclusions and future work in Section 6.

2 User Transparent Parallel Programming Tools

Whereas specifying the parallelization of applications by hand may be reasonable for a small group of experts, most users prefer to dedicate their time to describing *what* a computer should do rather than *how* it should do it. As a result, many programming tools have been developed to alleviate the problem of low level software design for parallel and distributed systems. In all cases, such tools are provided with a programming model that abstracts from the idiosyncrasies of the underlying parallel hardware. The relatively small user base of parallel computing in the MMCA community indicates, however, that existing parallelization tools are still considered too hard to use by MMCA researchers.

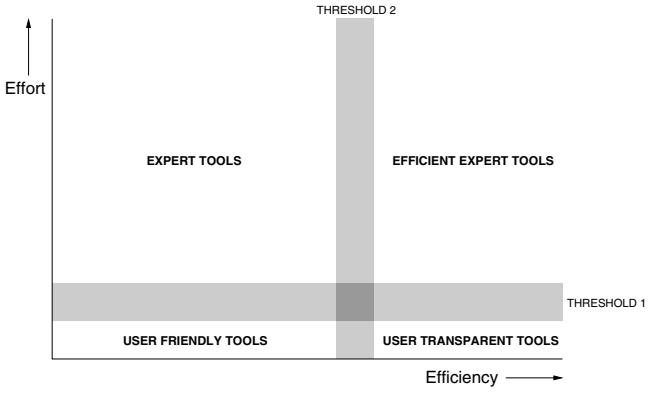


Fig. 1. Parallelization tools: effort versus efficiency. User transparent tools are considered both user friendly and sufficiently efficient.

The ideal solution would be to have a parallelization tool that abstracts from the underlying hardware completely, allowing users to develop *optimally efficient* parallel programs in a manner that requires *no additional effort* in comparison to writing purely sequential software. Unfortunately, no such parallelization tool currently exists and due to the many intrinsic difficulties it is commonly believed that no such tool will be developed ever at all [10]. However, if the ideal of 'obtaining optimal efficiency without effort' is relaxed somewhat, it may still be possible to develop a parallelization tool that constitutes an acceptable solution for the MMCA community. The success of such a tool largely depends on the amount of *effort* requested from the application programmer and the level of *efficiency* obtained in return.

The graph of Figure 1 depicts a general classification of parallelization tools based on the two dimensions of effort and efficiency. Here, the efficiency of a parallelization tool is loosely defined as the average ratio between the performance of any MMCA application implemented using that particular tool and the performance of an optimal hand-coded version of the same application. Similarly, the required effort refers to (1) the amount of *initial learning* needed to start using a given parallelization tool, (2) the additional expense that goes into obtaining a parallel program that is *correct*, and (3) the amount of work required for obtaining a parallel program that is particularly *efficient*. In the graph, the maximum amount of effort the average MMCA expert generally is willing to invest into the implementation of efficient parallel applications is represented by `THRESHOLD 1`. The minimum level of efficiency a user generally expects as a return on investment is depicted by `THRESHOLD 2`. To indicate that the two thresholds are not defined strictly, and may differ between groups of researchers, both are represented by somewhat fuzzy bars in the graph of Figure 1.

Each tool that is considered both 'user friendly' and 'sufficiently efficient' is referred to as a tool that offers full *user transparent parallelization*. An important additional feature of any user transparent tool is that it does not require the user

to fine-tune any application in order to obtain particularly efficient parallel code (although it may still allow the user to do so). Based on these considerations, we conclude that a parallelization tool constitutes an acceptable solution for the MMCA community only, if it is fully user transparent.

2.1 Parallel-Horus

In our earlier work, we have designed and implemented Parallel-Horus [6], a user transparent parallelization tool for the MMCA domain. Parallel-Horus, which is implemented in C++ and MPI, allows programmers to implement *data parallel* multimedia applications as fully sequential programs. The library's API is made identical to that of an existing sequential library: Horus [8]. Similar to other frameworks [2], Horus recognizes that a small set of *algorithmic patterns* can be identified that covers the bulk of all commonly applied functionality.

Parallel-Horus includes patterns for commonly used functionality such as unary and binary pixel operations, global reduction, neighborhood operation, generalized convolution, and geometric transformations (e.g. rotation, scaling). Recent developments include patterns for operations on large datasets, as well as patterns on increasingly important derived data structures, such as feature vectors. For reasons of efficiency, all Parallel-Horus operations are capable of adapting to the performance characteristics of a parallel machine at hand, i.e. by being flexible in the partitioning of data structures. Moreover, it was realized that it is not sufficient to consider parallelization of library operations *in isolation*. Therefore, the library was extended with a run-time approach for communication minimization (called *lazy parallelization*) that automatically parallelizes a fully sequential program at runtime by inserting communication primitives and additional memory management operations whenever necessary [11].

Results for realistic multimedia applications have shown the feasibility of the Parallel-Horus approach, with data parallel performance (obtained on individual cluster systems) consistently being found to be optimal with respect to the abstraction level of message passing programs [6]. Notably, Parallel-Horus was applied in recent NIST TRECVID benchmark evaluations for content-based video retrieval, and played a crucial role in achieving top-ranking results in a field of strong international competitors [6,12]. Moreover, recent extensions to Parallel-Horus, that allow for services-based multimedia Grid computing, have been applied successfully in large-scale distributed systems, involving hundreds of massively communicating compute resources covering the entire globe [6].

Despite these successes, and as shown in [7], for some applications a data parallel approach generates too much communication overhead for efficient parallelization. In these situations exploiting alternative forms of parallelism, such as task parallelism or pipelining, could improve the efficiency of the parallel execution. In this paper we investigate the design and implementation of a *task parallel* counterpart of Parallel-Horus. Keeping in mind the fact that applications should have a sufficient number of independent tasks available, and that load-balancing may be required for optimized performance, the following section discusses the design of our user transparent task parallel system.

3 System Design

In the MMCA domain, most applications apply a multimedia algorithm that transforms one or more input data structures into one or more output data structures. Generally, such input and output structures are in the form of dense data fields, like images, kernels, histograms and feature vectors. When we look more closely to the multimedia algorithms, we can regard them as sequences of basic operations, each transforming one or more dense data fields into another dense data field. These basic operations are exactly the ones that are provided by the Parallel-Horus API, referred to in Section 2.1. For our user transparent task parallel programming model we will use exactly the same sequential API.

In our task parallel approach, we aim to execute as many basic operations as possible in parallel. However, in most algorithms there are operations that require the output data of a preceding operation as one of its input data structure. As a result, to execute such an algorithm correctly in a task parallel manner, we need to identify these data dependencies. To that end we will not execute the operations of the programming model immediately, but we delay their execution and create a *task graph* of the basic operations in the application at run-time.

3.1 The Task Graph

When an operation is delayed in our programming model, an *operation node* is created instead, and added to the task graph[1]. The operation node contains a description of the operation itself, together with references to the nodes in the task graph corresponding to the input data structures of the operation. Furthermore, the operation node holds references to its parent nodes, as well as other parameters needed for the operation. In addition, each node registers how many children need their results as input.

A *future object* is created when the operation node is added to the task graph. The future object acts as placeholder for the result of the operation and it is returned to the application instead of the actual result (see Figure 2). At this point we take advantage of the fact that most intermediate results of an algorithm — by themselves — are not of much interest to the application (e.g. will not be written out to file). As a result, the application will not notice that we do not calculate the result data structure immediately.

Eventually the operation nodes will form a *Directed Acyclic Graph (DAG)*, containing all operations of the application as its nodes. In this graph, the *source nodes* are a special type of node. They do not have parent nodes, but they contain the data structure itself instead. For example, operations that import data structures (e.g., read an image from disk, create a kernel, etcetera) lead to the creation of such source nodes in the task graph.

[1] Some API calls lead to the creation of multiple operation nodes in the task graph. For example, a convolution with a Gaussian kernel is a single API call, but consists of one operation for creating the kernel and another one for the actual convolution.

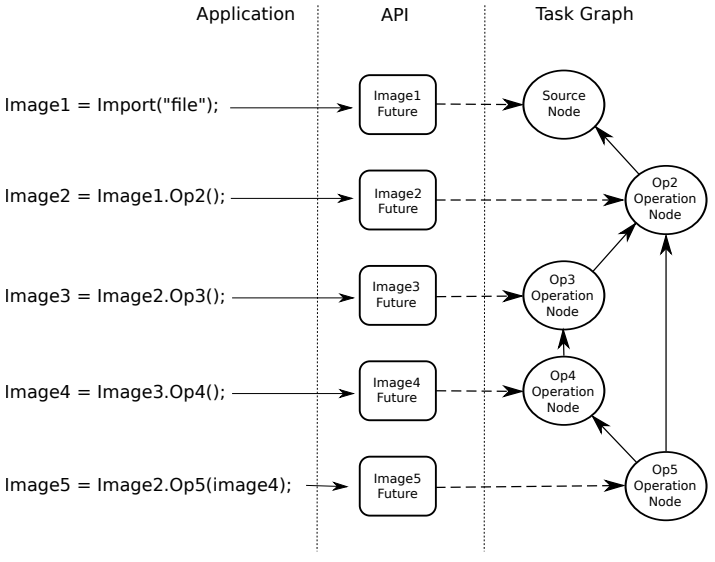


Fig. 2. Task graph construction. Each API call leads to the creation of a future object and an operation node; importing data from file leads to the creation of a source node.

3.2 Graph Execution

At some point during its execution, an application needs to access the data of future objects. At that point, the future will initiate the execution of all parts of the task graph that lead to the creation of the required result structure. The execution is initiated by requesting the result data from its corresponding operation node. This operation, in turn, needs to acquire the result data from all its parent nodes in order to calculate its own result data. The parent nodes will do the same until a source node is reached, which can return their result data immediately. This way, only those operations in the task graph that are needed to calculate the required data structure are executed, as shown in Figure 3. In addition, all nodes that took part in the execution now also have their result data and will become source nodes.

To ensure efficient execution of the task graph, we introduced a simple optimization to the execution process. First, we use the observation that any node that has only a single child node must be executed in sequence with that child node. Therefore, sequences of such operations are merged into a *composite operation* to prevent them from being executed on different processors. When the composite operations are identified, the execution of the task graph can be started. Parallel execution is obtained at nodes that have multiple parents. At these nodes, a separate job is created for each parent node to obtain its result. In contrast, any node with multiple children acts as synchronization point in the task graph, because all children will have to wait for the node to calculate its result, before they can continue their own calculations in parallel.

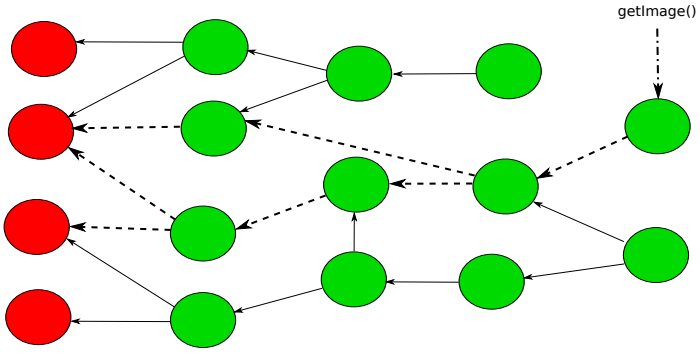


Fig. 3. Task graph execution. Only nodes leading to the required result are evaluated.

We implemented a prototype runtime system to execute the task graph as explained above. The runtime system is capable of merging nodes into composite nodes and then execute the required part of the task graph in parallel when a data structure is accessed through a future object. As part of our prototype runtime system we use the Satin [13] system to obtain parallel execution on clusters. Satin is a divide-and conquer system implemented in Java, that solves problems by splitting them into smaller independent subproblems. The subproblems are *spawned* by Satin to be solved in parallel. When all the subproblems are solved, the original problem collects all partial solutions and combines them to form the final result. This process of splitting problems into subproblems is repeated iteratively until trivial subproblems remain, which are calculated directly. It should be noted that Satin is a very powerful system. Apart from the automatic parallel execution of the spawned problems, Satin offers automatic load-balancing using random job stealing. Moreover, Satin offers fault-tolerance and automatic task migration without requiring any extra effort from the programmer.

To execute our task graph using Satin, the future spawns the operation node leading to the desired result data. In turn, the operation node will spawn all its parent operations in the task graph using Satin in order to get their result data. Consequently, the parent operations will spawn their parents until the source operations are reached. The source operations will not spawn any new jobs, but deliver their result data instead, thus ending the recursion. When all parents are finished and synchronized by the Satin runtime, the node can start its own execution and then deliver its results to it children.

Despite its benefits (see above), Satin induces overheads that could have been avoided if we would have implemented a complete run-time system from scratch. For example, Satin creates a spawn *tree* of the different parallel tasks instead of a DAG, to the effect that tasks are replicated in Satin — which may cause such tasks to be executed more than once. Also, Satin assumes the presence of many (even millions) of tasks, which may not be the case in all situations. However, the reader needs to keep in mind that the main contribution of this work is in the user transparent task parallel programming model. Despite these issues related

to Satin's overhead, in the remainder of this paper we do obtain close-to-linear speedup characteristics for several runtime scenarios.

4 A Line Detection Application

The following describes a typical, yet simple, example application from the MMCA domain. The example is selected because results for data parallel execution with Parallel-Horus are well available. Importantly, in some cases the *data parallel* execution of this particular application leads to non-linear speedups.

4.1 Curvilinear Structure Detection

As discussed in [14], the computationally demanding problem of line detection is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by:

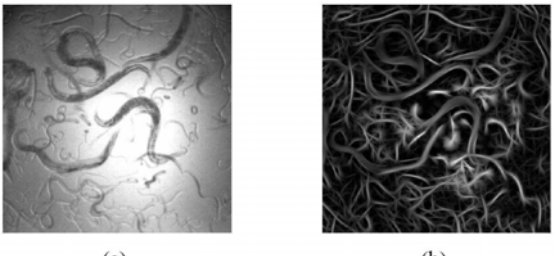
$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v \left| f_{vv}^{\sigma_u, \sigma_v, \theta} \right| \frac{1}{b^{\sigma_u, \sigma_v, \theta}}, \tag{1}$$

with b the line brightness. When the filter is correctly aligned with a line in the image, and σ_u, σ_v are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

$$R(x, y) = \arg \max_{\sigma_u, \sigma_v, \theta} r''(x, y, \sigma_u, \sigma_v, \theta). \tag{2}$$

Figure 4(a) gives a typical example of an image used as input. Results obtained for a reasonably large subspace of $(\sigma_u, \sigma_v, \theta)$ are shown in Figure 4(b).

The anisotropic Gaussian filtering problem can be implemented in many different ways. In this paper we consider three possible approaches. First, for each orientation θ it is possible to create a new filter based on σ_u and σ_v. In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Hence, a sequential implementation based on this approach (which we refer to as ***Conv2D***) implies full 2-dimensional convolution for each filter.



(a) (b)

Fig. 4. Detection of *C. Elegans* worms (courtesy of Janssen Pharmaceuticals, Belgium)

```
FOR all orientations θ DO
  RotatedIm = GeometricOp(OriginalIm, "rotate", θ);
  ContrastIm = UnPixOp(ContrastIm, "set", 0);
  FOR all smoothing scales σ_u DO
    FOR all differentiation scales σ_v DO
      FiltIm1 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v, 2, 0);
      FiltIm2 = GenConvOp(RotatedIm, "gaussXY", σ_u, σ_v, 0, 0);
      DetectedIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      DetectedIm = BinPixOp(DetectedIm, "mul", σ_u × σ_v);
      ContrastIm = BinPixOp(ContrastIm,"max", DetectedIm);
    OD
  OD
  BackRotatedIm = GeometricOp(ContrastIm, "rotate", −θ);
  ResultIm = BinPixOp(ResultIm, "max", BackRotatedIm);
OD
```

Fig. 5. Pseudo code for the ***ConvRot*** algorithm

```
FOR all orientations θ DO
  FOR all smoothing scales σ_u DO
    FOR all differentiation scales σ_v DO
      FiltIm1 = GenConvOp(OriginalIm, "func", σ_u, σ_v, 2, 0);
      FiltIm2 = GenConvOp(OriginalIm, "func", σ_u, σ_v, 0, 0);
      ContrastIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      ContrastIm = BinPixOp(ContrastIm, "mul", σ_u × σ_v);
      ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
    OD
  OD
OD
```

Fig. 6. Pseudo code for the ***Conv2D*** and ***ConvUV*** algorithms, with `"func"` either `"gauss2D"` or `"gaussUV"`

The second approach (referred to as ***ConvUV***) is to decompose the anisotropic Gaussian filter along the perpendicular axes u, v, and use bilinear interpolation to approximate the image intensity at the filter coordinates. Although comparable to the ***Conv2D*** approach, ***ConvUV*** is expected to be faster due to a reduced number of accesses to the image pixels. A third possibility (called ***ConvRot***) is to keep the orientation of the filters fixed, and to rotate the input image instead. The filtering now proceeds in a two-stage separable Gaussian, applied along the x- and y-direction.

Pseudo code for the ***ConvRot*** algorithm is given in Figure 5. The program starts by rotating the original input image for a given orientation θ. In addition, for all (σ_u, σ_v) combinations the filtering is performed by xy-separable Gaussian filters. For each orientation the maximum response is combined in a single contrast image. Finally, the temporary contrast image is rotated back to match the orientation of the input image, and the maximum response image is obtained.

For the ***Conv2D*** and ***ConvUV*** algorithms, the pseudo code is identical and given in Figure 6. Filtering is performed in the inner loop by either a full two-dimensional convolution (***Conv2D***) or by a separable filter in the principle axes directions (***ConvUV***). On a state-of-the-art sequential machine either program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace.

5 Evaluation

We implemented the three versions of the line detection application using our task parallel programming model. We tested the applications using the DAS-3 distributed supercomputer[2], which consists of 5 clusters spread over The Netherlands. For our experiments we used up to 64 nodes of the cluster located at the VU University in Amsterdam. On this cluster, each node contains 2 dual core AMD Opteron CPUs for a total of 4 cores and is connected by a high-speed 10 Gbit/s Myrinet network. We used the JRockit JVM version 3.1.2. In order to be able to compare the results of our experiments to previous experiments with Parallel-Horus, we only used a single worker thread on each node.

Table 1. Performance results (in seconds) for computing a typical orientation scale-space at 1° angular resolution (i.e., 180 orientations) and 8 (σ_u, σ_v) combinations. Scales computed are $\sigma_u \in \{3, 5, 7\}$ and $\sigma_v \in \{1, 2, 3\}$, ignoring the isotropic case $\sigma_{u,v} = \{3, 3\}$. Image size is 512×512 (8-byte) pixels.

# Nodes	ConvUV	Conv2D	ConvRot
1	198.23	2400.88	724.03
2	94.84	1193.74	387.04
4	48.20	593.18	190.31
8	24.08	298.45	98.25
16	12.43	150.73	53.60
32	6.53	77.50	30.21
48	4.58	52.94	19.77
64	3.63	40.66	17.00

Table 1 shows the results for the three applications under our programming model. For the sequential performance, it shows that the ***ConvUV*** approach is the fastest, followed by ***ConvRot***. As expected ***Conv2D*** is significantly slower, due to the excessive accessing of pixels in the 2-dimensional convolution operations. Overall, these results are similar to our earlier sequential results obtained with Parallel-Horus [7]. The speedup for the three versions of the application is shown in Figure 7(a). It shows that both ***ConvUV*** and ***Conv2D*** obtain close-to-linear speedups of 54.6 and 59.1, respectively (on 64 nodes). ***ConvRot*** scales worse than the other two, and reaches a speedup of 42.6 on 64 nodes.

For our task parallel programming model, scalability is influenced by the number of available parallel tasks. For all three algorithms, the inner loops are good candidates to serve as parallel tasks. As the structure of ***ConvUV*** and ***Conv2D*** is equal, we expect these algorithms to scale equally well. The marginal differences in the speedup results are explained by the fact that the ***ConvUV*** version contains much smaller task sizes, while the communication patterns are the same. As a result, the communication overhead is relatively larger. Unlike these two algorithms, in ***ConvRot*** some of the operations take place in the outer loop. As a result, the task graph constructed during ***ConvRot*** execution will contain more data dependencies between the tasks. This reduces the number of

[2] See http://www.cs.vu.nl/das3/

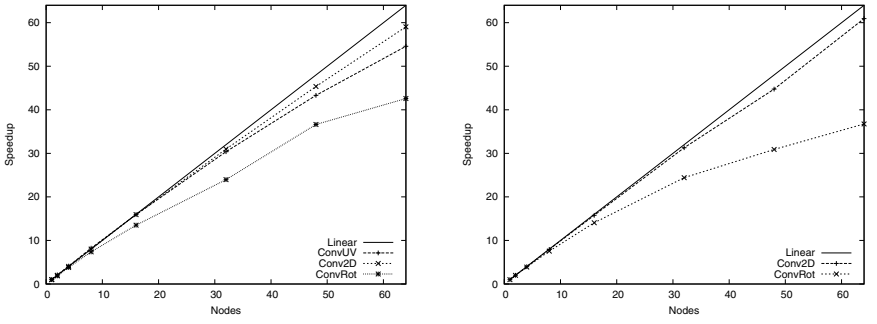


Fig. 7. (a) Speedup obtained with our model. (b) Parallel-Horus shows similar results.

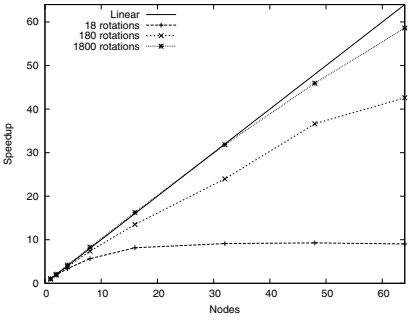


Fig. 8. Speedup of ***ConvRot*** for 18, 180 and 1800 rotations

tasks that can be run in parallel and complicates the distribution of the tasks over the compute nodes, which explains the lower speedup results.

Previous results [15] obtained for Parallel-Horus on DAS-3 are shown in figure 7(b). Despite the fact that the applied parallelization is entirely different, Parallel-Horus obtains very similar speedup characteristics in all compared cases. As a result, one could say that our task parallel model has not brought the desired improvement. As shown below, however, inspection of the Satin system reveals that our model still can perform much better than Parallel-Horus.

The load-balancing mechanism of the Satin system performs best when there are massive amounts of tasks available. As a consequence, our prototype runtime system works best for applications in which many tasks can be identified. As a result, the low number of tasks available in ***ConvRot*** reduces the effectiveness of our runtime system. Figure 8 shows that varying the number of rotations by a factor of 10 (by changing the angular resolution) greatly influences the speedup characteristics for ***ConvRot***. On 64 nodes, the speedup is only 9.0 for 18 rotations. For 180 and 1800 rotations, however, speedup is 42.6 and 58.6, respectively. In other words, for a very large number of rotations (thus: tasks) *all* versions of the application obtain close-to-linear speedup under our user

transparent task parallel programming model. In the near future we will reduce these scaling effects by adapting our runtime system (e.g. by replacing Satin) such that it requires less tasks to be effective.

Importantly, for Parallel-Horus the reduced speedup of ***ConvRot*** can *not* be improved by varying the chosen parameter space of the application [7,15]. The large data parallel overhead is due to the large number of global data dependencies present in the algorithm (i.e., the repeated image rotations). Clearly, in such cases our task parallel model provides a better solution.

6 Conclusions and Future Work

In this paper we introduced a user transparent task parallel programming model for the MMCA domain. The programming model delays execution and constructs a task graph in order to identify tasks that can be executed in parallel. When a result data structure is required by the application, only the relevant part of the task graph is executed by the runtime system in order to produce it.

We have built a prototype runtime system that is (currently) based on the Satin divide-and-conquer system. Measurement results for three versions of an example line detection problem show that a user transparent task parallel approach is a viable alternative to user transparent data parallelism.

In the near future we will investigate whether we can improve our graph execution engine, either by reducing Satin's overhead, or by replacing Satin altogether. Further investigations will be aimed at graph optimization strategies for improved performance. As the next major step, we aim to combine our task parallel model with Parallel-Horus to arrive at a user transparent programming model that employs both task and data parallelism. At the same time, we aim at extending our model to support many-core hardware (e.g. GPUs, FPGAs), which is particularly suitable for executing selected MMCA compute kernels.

References

1. Galizia, A., D'Agostino, D., Clematis, A.: A Grid Framework to Enable Parallel and Concurrent TMA Image Analysis. International Journal of Grid and Utility Computing 1(3), 261–271 (2009)
2. Morrow, P.J., et al.: Efficient implementation of a portable parallel programming model for image processing. Concur. - Pract. Exp. 11(11), 671–685 (1999)
3. Lebak, J., et al.: Parallel VSIPL++: An Open Standard Software Library for High-Performance Signal Processing. Proc. IEEE 93(2), 313–330 (2005)
4. Juhasz, Z., Crookes, D.: A PVM Implementation of a Portable Parallel Image Processing Library. In: Ludwig, T., Sunderam, V.S., Bode, A., Dongarra, J. (eds.) PVM/MPI 1996 and EuroPVM 1996. LNCS, vol. 1156, pp. 188–196. Springer, Heidelberg (1996)
5. Plaza, A., et al.: Commodity cluster-based parallel processing of hyperspectral imagery. J. Parallel Distrib. Comput. 66(3), 345–358 (2006)
6. Seinstra, F.J., et al.: High-Performance Distributed Video Content Analysis with Parallel-Horus. IEEE MultiMedia 14(4), 64–75 (2007)

7. Seinstra, F.J., Koelma, D.: User transparency: a fully sequential programming model for efficient data parallel image processing. Concurrency - Practice and Experience 16(6), 611–644 (2004)
8. Koelma, D., et al.: Horus C++ Reference. Technical report, Univ. Amsterdam, The Netherlands (January 2002)
9. Ramaswamy, S., et al.: A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. IEEE Trans. Parallel Distrib. Syst. 8(11), 1098–1116 (1997)
10. Blume, W., et al.: Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. IEEE Parallel Distrib. Technol. 2(3), 37–47 (1994)
11. Seinstra, F.J., Koelma, D., Bagdanov, A.D.: Finite State Machine-Based Optimization of Data Parallel Regular Domain Problems Applied in Low-Level Image Processing. IEEE Trans. Parallel Distrib. Syst. 15(10), 865–877 (2004)
12. Snoek, C., et al.: The Semantic Pathfinder: Using an Authoring Metaphor for Generic Multimedia Indexing. IEEE Trans. Pattern Anal. Mach. Intell. 28(10), 1678–1689 (2006)
13. van Nieuwpoort, R., et al.: Satin: Simple and Efficient Java-based Grid Programming. Scalable Computing: Practice and Experience 6(3), 19–32 (2005)
14. Geusebroek, J.M., et al.: A Minimum Cost Approach for Segmenting Networks of Lines. International Journal of Computer Vision 43(2), 99–111 (2001)
15. Liu, F., Seinstra, F.J.: A Comparison of Distributed Data Parallel Multimedia Computing over Conventional and Optical Wide-Area Networks. In: DMS, Knowledge Systems Institute, pp. 9–14 (2008)