Dynamic Process Partitioning and Migration for Irregular Applications*  

Pawel Czarnul†  
Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics  
Gdansk University of Technology, 11/12 Narutowicza, 80-952 Gdansk, Poland  
E-mail: pczarnul@eti.pg.gda.pl

Abstract

Many practical applications generate irregular, non-balanced divide-and-conquer trees which have different depths, possibly also different numbers of successors at different levels. Efficient parallelization is difficult as it requires dynamic partitioning and mapping of such trees to available processors. Irregular applications can obtain unpredictable intermediate results which then affect creation and termination of processes. The new proposed C++ framework called DAMPVM/DAC offers a combined scheme of dynamic process/data partitioning and migration which enables automatic parallelization of irregular divide-and-conquer applications taking into account processor speeds, network status, changing application requirements as well as external load introduced by other users. Experiments on a network of workstations include adaptive integration with and without process migration as well as static and dynamic codes for image recognition. The latter ones enable to assess both the overhead of the dynamic scheme compared to serial implementations for regular applications and scalability gains for non-uniform images.

Keywords: Parallel Software Environments, Methods for Automatic Parallelization, Optimization of Parallel Computations, Dynamic Process Migration, Dynamic Decomposition, Irregular Applications.

1. Introduction

The analyzed framework has been implemented ([4]) on top of DAMPVM ([3], [2]) – an extension of PVM implemented by the author. The proposed system handles irregular applications and dynamically adapts the allocation to minimize execution time. Applications are parallelized transparently to the user at runtime using partitioning of the DaC tree and process migration. [4] describes the details of the proposed object-oriented framework written in C++ which requires a programmer to override only a few virtual methods in order to obtain the divide-and-conquer functionality. The paper considers data decomposition for divide-and-conquer applications ([21], [14], [9], [6], [13]). Data partitioning is usually considered for SIMD (Single Instruction Multiple Data) computations and used in many algorithms e.g. sorting, image and sound manipulation ([21]). Similarly, the analysis of many processes (e.g. used in plane or car design) or physical phenomena modeled by partial differential equations solved in parallel using the finite difference method ([11]) must be preceded by proper data decomposition which is not trivial for irregular domains ([17]). There exist several other DAC systems targeted for both SMP and distributed memory systems. APERITIF (Automatic Parallelization of Divide and Conquer Algorithms, formerly APRIL, [5]) translates C programs to be run on parallel computers with the use of PVM ([7]). REAPAR (REcursive programs Automatically PARallelized, [8]) derives thread-based parallel programs from C recursive code to be executed on SMP machines. Cilk ([16]) is a similar thread-based approach. ATLAS ([11]) and Satin ([19]) are Java-based systems. Other framework-based approaches are Frames ([10]) and an object-oriented Beeblebox ([15]) system. An algebraic DaC model is described in [14]. There are also approaches in which different variants are considered at different levels ([6]).

1.1. Dynamic Partitioning vs Migration

There are two mechanisms in the proposed DaC scheme that work together. They are complementary and are used in different situations in order to achieve the same goal. The scheme tries to minimize the maximum of estimated execution times for the nodes used by the application ([3]). Processes can be created by others at runtime. It is possible that there are fewer processes than the number of nodes since processes can terminate any time. In order to resolve the above situations two mechanisms have been proposed:

---

*The paper partially sponsored by the Polish National Grant KBN No. 8T11C 001 17
†currently on leave at Electrical Engineering and Computer Science, University of Michigan, 3225 EECS Building, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, U.S.A., (734) 7640608, pczarnul@eecs.umich.edu
Process migration – migrates processes of the considered application from overloaded to underloaded nodes so that the total execution time is minimized ([18]), balances load when there are more processes than processors in the network (possibly hiding idle time).

Dynamic process partitioning – creates more processes so that there are at least as many processes as the number of nodes; balancing is done by: launching new processes on least-loaded nearest neighbors, process migration can tune the allocation afterward.

The above mechanisms are complementary in the sense that only one is activated in a certain situation. In the current implementation when a node detects low load (but higher than 0) it informs neighbors that it has dropped and they try to migrate tasks to the requesting node. If no tasks have been migrated since there were no migratable processes, the load on the underloaded node will drop down further eventually. Nodes check periodically if their loads have reached a certain very low threshold. If this is the case a node requests dynamic partitioning (and not process migration since it was unsuccessful) of processes (i.e., tree splitting, [4]) on its nearest neighbors. Newly spawned tasks are placed on the requesting node. Additionally, process migration can be used later (if supported by the processes) to balance the load even better. This scenario is shown in Figure 1.

Dynamic partitioning will not be activated if there are enough tasks in the system so it will not create unnecessary processes and additional runtime costs then. In the DAMPVM model processes can enable and disable migration at runtime depending on their intermediate results, system requirements etc. DAMPVM schedulers migrate only processes which have enabled migration but takes all into account when computing node loads.

2 Migration: Overhead vs Benefits

DAMPVM provides a template for migration but the programmer is responsible for writing of functions packing and unpacking the state of a process ([3]). It may be easy or quite difficult to write these routines to work effectively. As presented in Paragraph 3.1, efficient migration of a process state in the adaptive integration example is not difficult. On the contrary, other applications like computing Fibonacci numbers or in particular searching $\alpha\beta$-trees would require the tree to be rebuilt from scratch. This is costly or has to sacrifice some previously computed values if the state was saved periodically from time to time. In such a case there is a trade-off between how often the state should be saved and the cost of this operation. If the overhead for the migration routine is likely to be high or the recovery of the process state is very costly it is advisable to disable migration and rely only on the dynamic partitioning scheme. It can also balance load very well combined with the allocation strategy which spawns tasks on lightly-loaded nearest neighbors. Migration can be beneficial in the DaC scheme when: there are already more processes in the system than the number of processors (no overhead for creating new processes), the system is being used by other users intensely (process migration can move tasks from overloaded processors in order to shorten the total execution time). It can help to hide communication latency (migration enables to map more than one process per node so that the computations of one process can overlap the communication of the others). If there are no other users in the system or their activities do not affect the run of the application significantly there is a trade-off between how much idle time the migration scheme can save and the overhead caused by excessive communication (also pointed out in [20]).

3. Experimental Results

3.1. Numerical Adaptive Quadrature Integration Example

A DAC-based numerical integration example has been implemented which integrates any given function. The idea is proposed in [21], extended and implemented in [4]. Function $f(x)$ and range $[a, b]$ are given. The range is partitioned into two subranges which are divided recursively until areas can be approximated accurately enough by the two trapezoids determined by the two subranges ([4]). Irregular functions generate very irregular trees which require adaptive partitioning and mapping of data ranges to proces-
migration is supported. Suppose the whole tree is executed by one process and at the moment migration is invoked the computed value upon return. Thus functions for migration ([3]) can contain only these three values. If a process computes a subrange it always does this from left to right as it is implied by the DAC scheme. If its execution is interrupted i.e. process migration occurs it simply delivers the value for the left computed subrange as the initial value and the right not yet computed range for the migrated process (its new copy).

This also means that the application does not require much interprocess communication. Figure 2 shows how process migration is supported. Suppose the whole tree is executed by one process and at the moment migration is invoked the process is to compute area B and add it to the previously computed area A. If area B has not been computed or added to area A the process packs area A and range [5, 16] as the initial values for the new process. If area B has been added to area A then the process packs area A+B and range [6, 16].

![Figure 2. Adaptive Integration: Process Migration](image)

Apart from the heterogeneous configuration tested in [4] the implementation was run on a homogeneous network up to 16 identical processors. Execution times and speed-ups are shown in Figures 3 and 4 respectively. The code was tested using the same functions as in [4] but for larger ranges. Two functions were used:

1. \( i(x) = \sin^2(x) : \int_0^t f(x)dx \); this one is periodic and thus the execution times for same-size ranges are similar; since \( t \) is much larger than the period of the \( \sin^2(x) \) function then all processors get approximately the same amount of work and no significant imbalance occurs,

2. \( j(x) = \begin{cases} \sin^2(x) & 0 \leq x \leq \frac{t}{2} \\ x - \frac{t}{2} & \frac{t}{2} \leq x \leq t \end{cases} : \int_0^t g(x)dx \); this case leads to two separate ranges \([0, \frac{t}{2}]\) and \([\frac{t}{2}, t]\) which cause an imbalance since the latter one is computed very quickly. Fortunately DAMPVM/DAC partitions the first range then and assigns new processes with smaller subranges to idle processors.

### 3.1.1 Migration vs Dynamic Partitioning

Figures 3 and 4 show results with and without migration for the irregular function \( j(x) \) without external load. It is visible that migration does not improve the results and even introduces additional overhead for larger configurations. The implemented migration scheme tries to balance load in advance assuming certain (dynamically reported) process sizes. For irregular applications they are often impossible to predict accurately and thus the procedure may introduce unnecessary communication. However, this also depends on how large processes are. If their execution times are long then migration introduces additional overhead but if they are very short then it can hide communication latencies when mapping multiple processes onto a single node ([21]). Migration is still very useful if other users use the same machines. Also, for some configurations, especially if the number of initially spawned processes is not a multiple of the number of processors migration actually divides work at runtime ([22]).

### 3.2. Parallel Image Recognition Example

This application aims at the recognition of certain known diseases based on template images. There are (usually large) images taken from a patient’s body and several template images which represent some known symptoms typical of certain diseases. The goal is to determine whether an image matches any of the template images stored in a database. Template images represent so-called regions of interest (ROIs) and may be much smaller than the target image. As pointed out in [12] this task can be parallelized at three different levels:

**Target parallelism.** Pairs consisting of template images and copies of the target image are compared in parallel by different machines.

**ROI parallelism.** In this case one ROI is replicated and compared in parallel to many target images.

**MA parallelism.** A Matching Algorithm (MA) is executed in parallel by partitioning the target image and parallel comparisons to one selected ROI.

The target and ROI parallelism strategies are easy to implement when a sequential algorithm is available. The sequential program can be started on several machines using ROIs corresponding to various diseases or many target images respectively. In this work the MA parallelism has been investigated and implemented. It is assumed that a patient’s image is not smaller than ROIs. In this case domain decomposition is performed which partitions the target image into
parts and then one ROI is compared to the parts of the target image in parallel. Four versions of the code were developed as outlined below:

**Regular images/static code.** The target image is partitioned statically along the two dimensions which gives rectangular subdomains assigned to the processors. It is assumed that different parts of the target image take approximately the same time to be compared to the ROI (this can be dependent on the recognition algorithm). The code does not use DAMPVM/DAC and is implemented as a pure PVM application.

**Regular images/dynamic code with DAMPVM/DAC.**
The target image is recursively divided into four subdomains which are then partitioned again until the desired size is obtained. The subimages are compared in the leaves of the tree. Since the images are regular and there is no external load we can evaluate the overhead of the DAMPVM/DAC scheme compared to the previous configuration.

**Irregular images/static code.** Different parts of the target image take considerably less or more time to be compared to the ROI. This can cause a significant imbalance and poor scalability as the code is not able to predict and adapt to the irregularity. Similar imbalance can be caused by active processes belonging to other users.

**Irregular images/dynamic code with DAMPVM/DAC.**
Idle processors send dynamic partitioning requests and receive work spawned as new processes. Although this implementation introduces additional overhead for DAC and the recursive implementation, it is able to adapt to the current state of both the application and external load which results in better scalability.

### 3.2.1 Static vs Dynamic Partitioning

In the case of static decomposition a 2-dimensional image is cut along dimensions X and Z which produces rectangular domains assigned to processors. This version is fully optimized for static environments in which case there are no other activities thus the speeds of the nodes are guaranteed throughout the simulation. Moreover, it works efficiently provided that comparisons of different parts of the image (of the same size) take the same time. This does not have to be the case since some parts of the image may contain pixel patterns very similar but still different from the ROI and some may include very different ones which will be rejected immediately. This may lead to a very irregular application for which static partitioning will result in unequal load distribution and poor scalability. Unfortunately, it is impossible to determine in advance which parts are similar and which are not. It can be estimated by computing histograms which count the number of pixels of various colors. This information could be used as weights for the static partitioning strategy. Nevertheless, it would be difficult and inconvenient to obtain correct weights and good speed-up. However, in the case of regular images/uniform domains, static partitioning is better than any dynamic strategy as there is no overhead for the runtime part of the system. This is shown in Figures 5 and 6. The same static example was executed using the dynamic code. The size of the initial image was 4000x4000 pixels with the 8-bit color depth giving approximately a 16MB image. The ROI used was a 400x400 image with the same color depth. The dynamic version was using the 60 pixel limit for leaf computations. It is visible that the static version is better for regular images but only by a narrow margin. A simple pixel-by-pixel comparison method was used.

Dynamic partitioning is suitable when:

1. other users can run their applications which can increase the execution time of the image recognition code on those machines; they can also terminate their tasks thus increasing effective speeds of certain nodes,

2. some parts of the image can be very similar to the ROI which takes significantly longer compared to cases for which differences are detected in early stages of the recognition algorithm; this results in different execution times on different nodes in spite of same size subdomains.

The dynamic solution partitions the initial image recursively into four parts which are then recursively divided again in the same manner until a certain predefined size is achieved. The leaves compare small pieces of the initial image to the ROI. If an idle node requests work it is assigned some part of the tree by the DAMPVM/DAC schedulers and can execute the same algorithm for this subtree. This ensures the adaptivity of the scheme to both irregular applications and changing external load. The dynamic version (unlike the static one) introduces some additional overhead like graphical interface consoles for each node which receive data from DAMPVM kernels about the current load, number of processes etc. Moreover, DAMPVM kernels exchange some information between each other including monitoring the statuses of links (and their speeds) etc. The experiments used an irregular image for which one half was much different than the ROI which could be assessed more quickly than for the other upper part. If the allocation is static the performance is poor (Figures 7 and 8) because half of the tasks assigned to the lower half of the image become idle quickly. Unlike the dynamic code with DAMPVM/DAC, the static scheme is not able to assign work to idle processors after this has happened.
Figure 3. Adaptive Integration: Execution time

Figure 4. Adaptive Integration: Speed-up

Figure 5. Image Recognition for a Regular Example: Execution time

Figure 6. Image Recognition for a Regular Example: Speed-up

Figure 7. Image Recognition for an Irregular Example: Execution time

Figure 8. Image Recognition for an Irregular Example: Speed-up
4. Conclusions

Dynamic partitioning and migration were analyzed for two irregular applications. Although [18] proved that migration is useful in the presence of external load, Paragraph 3.1.1 concludes that it does not improve scalability for large divide-and-conquer trees without external load as shown for the integration example. Dynamic partitioning can create small processes which are dynamically allocated to processors. Paragraph 3.2 shows that the dynamic DAM-PVM/DAC code easily outperforms static implementations for irregular image recognition while being only slightly less scalable for regular images. This proves that the DAM-PVM/DAC software is suitable for irregular scientific applications ([21], [1]) executed on networks of workstations and supercomputers both shared by many users and dedicated. The proposed DaC software is available at the DAMPVM Web site ([2]) and can run on any parallel architecture that PVM can.

References