Programming, Tuning and Automatic Parallelization of Irregular Divide-and-Conquer Applications in DAMPVM/DAC

Pawel Czarnul

Computer Architecture Department
Faculty of Electronics, Telecommunications and Informatics
Technical University of Gdansk
11/12 Narutowicza, 80-952 Gdansk, POLAND
email: pczarnul@eti.pg.gda.pl
WWW: http://www.ask.eti.pg.gda.pl/~pczarnul
http://xinu3.eti.pg.gda.pl/~pczarnul
Phone: +48 58 3472524
Fax: +48 58 3472863

1 work partially sponsored by the Polish National Grant KBN No. 8T11C 001 17
2 partially developed on leave at Electrical Engineering and Computer Science, University of Michigan, EECS, 1301 Beal Avenue, Ann Arbor, 48109 MI, U.S.A.
Abstract. This paper presents a new object oriented framework DAMPVM/DAC which is implemented on top of DAMPVM and provides automatic partitioning of irregular divide-and-conquer (DAC) applications at runtime. The processes are then mapped dynamically to processors taking into account their speeds and even loads by other user processes. The paper presents the programming interface (API) of the framework, available tuning mechanisms, internal solutions with respect to automatic partitioning and mapping to processors. Finally, specific parameters, optimization techniques and simulation results are shown for a variety of irregular divide-and-conquer applications. The applications include $\alpha$/$\beta$ search, recursive Fibonacci, finding twin prime numbers in parallel and previously implemented and now carefully analyzed and tuned adaptive quadrature integration and image recognition. Various DAC parameters were tuned for specific applications including costs of computing vectors/subtrees, maximum partitioning levels etc. Moreover, the overhead of DAMPVM/DAC compared to sequential implementations is shown for all the implemented applications.

1 Introduction

This work deals with decomposing divide-and-conquer application data into processes and mapping them to a highly changeable environment at runtime so that the total execution time of the application is minimized. It is also assumed, unlike in many other approaches, that the environment is being used by other users at the same time who can act unpredictably with respect to the number of tasks being started, their processor and memory requirements, as well as the frequency of changes. Further, at the programming level the goal is to provide a high level programming framework that would partition an application into processes automatically and map them to the network at runtime. Unlike many other approaches which consider data decomposition and load balancing algorithms through data or process migration separately, this work combines dynamic process partitioning (dynamic data decomposition into processes) and dynamic process allocation and migration (mapping) into one system. Although data decomposition is determined by the algorithm in divide-and-conquer algorithms it does not indicate which subtrees should be spawned as separate processes. Dynamic partitioning into processes means that some subtrees can be spawned as new processes at runtime if needed. Additionally, dynamic allocation and migration are executed automatically by the hidden layer taking the application and environment parameters into account. DAMPVM/DAC is the implementation of these requirements and is based on DAMPVM ([1], [2], [3]). Both systems were developed by the author.

2 Existing Solutions

Since this paper discusses the DAMPVM/DAC framework it is necessary to present the developments of other authors with respect to the functionality of the system. This includes: data decomposition in parallel applications, load balancing algorithms used to distribute data (processes) at runtime and implementations of systems that include dynamic load balancing features and those dedicated for divide-and-conquer applications. This work focuses on efficient execution of time-consuming possibly irregular divide-and-conquer applications on networks of workstations, possibly heterogeneous and used by other users at the same time. There are parallel libraries available like MPI implementations (LAM ([4]) or MPICH ([5])) or the PVM environment ([6]) which are targeted for demanding scientific applications. The aforementioned parallel libraries and others similar are reasonably simple and do not take many important factors into account which appear in shared NOWs. In particular this refers to little support for dynamic changes of the environment or applications like: other users utilizing the same machines at the same time which effectively changes their speeds at runtime, communication links between processors which may have different bandwidth depending on the current communication between other user processes. Also there is no support for irregular divide-and-conquer applications, they have to be coded at a very low level and adaptivity due to the changing environment and the irregularity is difficult to achieve. The low level programming required by these libraries has driven parallel programmers to create higher level environments which support dynamic process management, process migration and also handle irregular divide-and-conquer applications at runtime. This section presents an overview of data decomposition techniques, available load balancing algorithms
and the environments which support dynamic process management, various dynamic load balancing algorithms incorporated into these systems as well as divide-and-conquer frameworks. However, the available systems either lack good support for irregularity detection and compensation in NOWs (many of them are targeted for shared memory systems) or do not offer capabilities to detect loads caused by other users and compensate for that in the assumed environment. In conclusion, existing dynamic load balancing environments do not provide specialized capabilities needed for irregular divide-and-conquer applications. Similarly divide-and-conquer environments do not provide efficient process/thread management for multi-user environments.

2.1 Data Decomposition

This work considers domain decomposition (also called data partitioning or decomposition) for divide-and-conquer applications ([7], [8], [9], [10], [11]) as opposed to functional decomposition. Data partitioning is usually considered for SIMD (Single Instruction Multiple Data) computations and used in many algorithms e.g. sorting, image and sound manipulation ([7]) as well as physical phenomena modeled by partial differential equations solved in parallel using the finite difference method or others ([12], [13], [14] and [15]). The difference compared to the irregular divide-and-conquer assumption is that the application graph is known in advance and is static since it corresponds to the simulated domain. It is usually a 2- or 3-dimensional grid as opposed to an irregular tree. The nodes in the domain form a graph (possibly with different weights for different cells representing the cost of computations) which are connected by edges representing data dependencies. The goal is then to partition the domain into as many subdomains as the number of available processors and minimize the number of edges connecting the subdomains. They represent communication which usually has to be committed every iteration and thus can be considerable compared to the costs of computations. This is different from a divide-and-conquer application in that the latter one divides data into parts and then collects results but generally does not require data exchanges in the meantime. On the other hand the sizes of subtrees can be difficult to predict in advance. There are many graph partitioning algorithms available including Recursive Coordinate Bisection (RIB), Recursive Inertial Bisection (RIB), space-filling curve techniques (e.g. Peano-Hilbert curves), Kernighan-Lin, spectral methods, Multilevel Recursive Bisection, Multilevel $k$-way Partitioning and others ([16], [17]). Parallel incremental graph partitioning with the use of linear programming is proposed in [18]. An algebraic model of divide-and-conquer is described in [8].

2.2 Mapping and Dynamic Load Balancing

Available dynamic load balancing algorithms differ in assumptions concerning the granularity of a distributed application, system topology and locality as well the way the optimized metrics are defined. A few classes of neighbor algorithms are distinguished and compared in [19] where a global, good assignment is achieved by successive exchange of load among neighbors. Gradient methods are described in [20] (a basic version) and [21] which is the study of various versions of such strategies. A comparison of load balancing strategies is given in [22] which includes Gradient Model (GM), Sender/Receiver Initiated Diffusion (SID/RID), Hierarchical Balancing Method (HBM) and Dimension Exchange Method (DEM). Other papers focus on certain algorithms or different comparisons: RID and DEM are presented in [23], Diffusion Algorithm Searching Unbalanced Domains (DASUD) and SID in [24], DE and diffusion algorithms in [19], Mesh/Cube/Tree Walking Algorithm (MWA/CWA/TWA) in [25] and Adaptive Contracting Within Neighborhood (ACWN) in [26]. Various algorithms including gradient and randomized methods are compared in [21] including numbers of task-hops and control messages to achieve load balance. The gradient method was proposed initially in [20]. Recent work on diffusive load balancing algorithms which minimizes the $l_2 = (\sum_{y,x,q} (t_{xy}^q)^2)^{1/2}$ communication metric to achieve load balance is presented in [27] and [28]. [29] does not deal with process migration but instead focuses on data redistribution in the PVM environment. A hydrodynamic load balancing approach is presented in [30]. Work transfer to balance the load and corresponding execution times of the HB, GDE (Generalized Dimension Exchange) and diffusive algorithms are compared in [31]. MWA, CWA and TWA are considered in [25] in the context of runtime incremental parallel
scheduling RIPS which is distinguished from dynamic scheduling mainly with respect to separate system and user computation phases and global load information in the former ([32], [33]).

Mapping strategies of divide-and-conquer trees considered in this work are presented in [34] and [35]. With respect to mapping particular problems considered in this paper, [36] presents an excellent overview of parallel $\alpha\beta$ strategies.

2.3 Implementations of Dynamic Load Balancing and Divide-and-Conquer Systems

There are many environments that incorporate dynamic load balancing features for parallel computing like: Condor ([37]), MPVM ([38]). Other similar systems include Dynamic PVM ([39]), Dynamite (based on Dynamic PVM) which provides an automatic load balancing tool for the PVM environment using process checkpointing ([40]), CHaRM ([41]), MpPVM ([42], [43]), tmPVM ([44]). There exist solutions implemented at the operating system level e.g.: Sprite, Chorus, Mach ([45]). Neither of them offers both high-level support for automatic data decomposition at runtime and dynamic process allocation/reallocation in case the system state changes.

With respect to the existing divide-and-conquer frameworks they either do not provide process migration or do not take multi-user environments into account. APERITIF (Automatic Parallelization of Divide and Conquer Algorithms, formerly APRIL, [46]) translates serial C programs to be run in parallel using PVM with no automatic compensation for irregularity. REAPAR (REcursive programs Automatically PARalleled, [47]) and Cilk ([48]) are thread-based approaches. Java is represented by ATLAS ([49]) and Satin ([50]) targeted for distributed memory utilizing work stealing. Frames ([51]) and Beeblebox ([52]) are other framework-based approaches.

3 Proposed Approach

The proposed DAMPVM/DAC framework ([53], [54]) is implemented on top of DAMPVM ([1]) and uses its dynamic features like process allocation/migration and estimation of forward execution times of processes at runtime. It allows to use external load balancing algorithms (like the one proposed in [55]) thanks to a special communication protocol. This makes it a perfect choice for efficient execution of irregular parallel applications in highly changeable environments. Dynamic process partitioning is used to create at least as many processes as the number of available processors in order to keep all of them busy. Dynamic process migration is invoked before dynamic process partitioning if enough tasks are already available to balance the load or if it is necessary to move some tasks because some nodes have become overloaded by other users ([54]).

Dynamic Allocation and Migration Parallel Virtual Machine (DAMPVM, [1], [3]) provides a programming environment that incorporates dynamic process allocation and heterogeneous migration. It has also been extended with fli tering node loads using low pass fli ters which enables not to activate costly migration procedures for high frequency high amplitude short-lived changes and fli ter them out or treat them as a certain average load instead. The DAMPVM environment consists of kernels, each with its own optional graphical interface window, running on each machine and monitoring node parameters and application processes running on it, exchanging information between each other and performing dynamic allocation of processes and process migration, if necessary. In its nature it is similar to PVM i.e. provides a library of communication functions analogous to those in PVM but also dynamic process allocation/migration ([11]) as well as incorporates many more application and system parameters in the decisions it makes with respect where to place tasks and how to balance them. System parameters include: $s_{pt}$ – the speed of node $N_i$ defned as the inverse of time needed to perform a selected set of instructions, for our consideration this set should contain the instructions used in application $A$; $othL_i(t)$ – the current load of node $N_i$ by processes that do not belong to application $A$ at moment $t$, expressed in percentages, since it is always calculated on node $N_i$ and the neighbors of $N_i$ are notified about its changes and keep their own copies; the copy for node $N_k$ is denoted as $othL_i^k(t)$, $estt_i(t)$ – the estimated forward execution time of all the processes of application $A$ currently running on node $N_i$ from moment $t$ until all the processes of application $A$ on node $N_i$ are completed. Communication parameters are defined by latency $L_i$ (startup time) and bandwidth defned as $\gamma_d(t)^{-1}$ ([56]). To preserve some locality of the task scheduling and migration algorithms, a neighbor graph
has been introduced to describe the system configuration. It defines whether a certain node is allowed to spawn a new task on another node or migrate existing tasks to that node. Application parameters include $instr_{ij}(t)$ that represents forward work for the $j$-th process on node $N_i$ and can be estimated by the number of elementary instructions for from moment $t$ until the process finishes, the number of remaining loops in the process etc. $PS_{ij}(t)$ denotes the size of the state of the $j$-th process on node $N_i$ at moment $t$. DAMPVM balances the following estimated times across all considered nodes: $\forall i \in \{1, 2, \ldots, n\}$

$$estt_i(t) = \frac{1}{sp_i(1-\frac{PS_{ij}(t)}{100})} \sum_{j=1}^{CP(i,t)} instr_{ij}(t)$$

where $CP(i,t)$ denotes the set of all the processes of application $A$ running on node $N_i$ at moment $t$. DAMPVM incorporates a plug-in for additional external dynamic load balancing algorithms. The system has been equipped with a protocol between its schedulers and load balancing kernels which enables to add many new dynamic load balancing algorithms.

DAMPVM/DAC uses DAMPVM as the layer which provides the following: interprocess communication, transparent process migration and allocation and enables process instrumentation. The latter one can be used to provide the runtime part of the system (DAMPVM kernels) with additional information about processes defined by the model above. It is then used for load balancing decisions inside DAMPVM ([1]). The new dynamic DAMPVM/DAC (Divide-And-Conquer) framework, implemented on top of DAMPVM, provides a high-level C++ programming skeleton to the programmer and can be parametrized in many ways. This allows the reuse of the skeleton code for many applications and eliminates errors in the implementation since the programmer can focus on the application itself. The main contribution of this work in this field is that it combines both process partitioning and dynamic process migration in order to achieve the same goal i.e. balance load in the network ([53], [54]). This allows to achieve high speed-ups and adapt to changes in the application’s data flow or the state of the network. DAMPVM/DAC is available at [http://www.ask.eti.pg.gda.pl/~pczarnul/DAMPVM.html](http://www.ask.eti.pg.gda.pl/~pczarnul/DAMPVM.html). It decomposes and maps irregular parallel applications to a highly changeable heterogeneous network of computers which is open for other users as well. Additional details are provided in Paragraph 6. DAMPVM/DAC dynamically sets process sizes at runtime even for irregular applications in order to achieve the best possible speedup. Apart from that nodes monitor how many tasks are running on them and what the current load is (measured separately for other users). Several dynamic functionalities are available for the DAMPVM/DAC scheme. A process can obtain at runtime: the current node in the tree being executed, the node which is the parent of the spawned subtree. This information can be used to transfer the global state of the process to spawned tasks. Processes executing subtrees can dynamically send asynchronous messages to other subtrees executed in parallel. This may be useful if intermediate results of certain processes can speed up the execution of tasks spawned before.

Fig. 1: Relations between DAC, DAMPVM and PVM
All the layers of the DAMPVM/DAC framework are presented in Figure 1 including the required underlying layers and provided functions.

## 4 Programming DAMPVM/DAC

This section introduces the high-level programming interface of the parallel DAMPVM/DAC scheme. Application instrumentation functions available in DAMPVM are used in the DAC library to provide an efficient implementation of the divide-and-conquer scheme including: dynamic process partitioning, dynamic process migration, idle node detection, hiding communication latency.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**void DaC_Init(int *, char *<strong>);</strong></td>
<td>initializes DAMPVM/DAC; executes PC_Init() and performs some DAC-specific initialization</td>
</tr>
<tr>
<td><strong>void DaC_Finish(void);</strong></td>
<td>terminates programs which use DAMPVM/DAC; executes PC_Finish()</td>
</tr>
<tr>
<td>**virtual long DaC_VectorSize(Object <em>vector_l, Object <em>vector_r);</em></em></td>
<td>returns the vector size (used by DAMPVM in the allocation algorithm); sometimes should be overwritten for better performance; the default returned value is (vector_l - vector_r)</td>
</tr>
<tr>
<td>**virtual void DaCInitialize(Object <em>vector_l, Object <em>vector_r);</em></em></td>
<td>executed only once in the beginning for each process, enables to do some initialization</td>
</tr>
<tr>
<td>**virtual bool DaCTerminate(Object <em>vector_l, Object <em>vector_r) = 0;</em></em></td>
<td>determines if the DaC strategy should still divide the vector; this must be supported by a programmer</td>
</tr>
<tr>
<td>**virtual void DaCLeafComputations(Object <em>vector_l, Object <em>vector_r);</em></em></td>
<td>the computations which are to be executed on the vector if further partitioning is forbidden by method DaCTerminate(); this function can be overwritten by a programmer if they wish to do so</td>
</tr>
<tr>
<td>**virtual void DaCPreComputations(Object <em>vector_l, Object <em>vector_r);</em></em></td>
<td>an optional method which enables to execute some code before the vector is further partitioned</td>
</tr>
<tr>
<td>**virtual int DaCHowManyNodes(Object <em>vector_l, Object <em>vector_r) = 0;</em></em></td>
<td>a mandatory method which returns the number of partitions the current subvector is divided into; this one must be overwritten by a programmer</td>
</tr>
<tr>
<td><strong>virtual Object</strong> * DaCDivide(Object <em>vector_l, Object <em>vector_r) = 0;</em></em></td>
<td>a mandatory method which partitions the vector into some number of smaller vectors; this one must be overwritten by a programmer; the pointer to the table of pointers to new subvectors should be returned</td>
</tr>
<tr>
<td><strong>virtual void DaCPostComputations(Object</strong> <em>new_vectors, Object * &amp;vector_l, Object * &amp;vector_r);</em>*</td>
<td>an optional method which enables to execute some code after the vector has been partitioned e.g. merging data from the subvectors; new_vectors – the pointer to the table of pointers to new subvectors; this function should finally perform all operations so that proper final values are delimited by vector_l and vector_r</td>
</tr>
<tr>
<td><strong>virtual void InitializeData(Object * &amp;vector_l, Object * &amp;vector_r);</strong></td>
<td>initializes data</td>
</tr>
<tr>
<td>**virtual void MasterReport(Object <em>vector_l, Object <em>vector_r);</em></em></td>
<td>enables the master process to report data e.g. send it to the standard output</td>
</tr>
<tr>
<td><strong>void Run(void);</strong></td>
<td>invokes the DAC algorithm</td>
</tr>
<tr>
<td>Method Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>void EnableMigration(void);</td>
<td>enables migration for the process which means it may be invoked if the DAMPVM/DAC scheme decides to do so</td>
</tr>
<tr>
<td>void DisableMigration(void);</td>
<td>disables migration for the process</td>
</tr>
<tr>
<td>void DaC_EnableDaC(void);</td>
<td>enables dynamic partitioning of the process calling this method</td>
</tr>
<tr>
<td>void DaC_DisableDaC(void);</td>
<td>disables dynamic partitioning of the process calling this method</td>
</tr>
<tr>
<td>void DaCActivation(void);</td>
<td>invokes dynamic process partitioning of the calling process (from the code) – normally it is called automatically by the DAC scheme if it decides that a new process is required</td>
</tr>
</tbody>
</table>

**Managing Tree Levels and Node Indexes**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int GetCurrentLevel(void);</td>
<td>returns the current level in the tree (a lower level means nodes closer to the root)</td>
</tr>
<tr>
<td>int GetCurrentNodeOnLevel(int nLevel);</td>
<td>returns the index of the currently executed node on the given level</td>
</tr>
<tr>
<td>int GetLastDynamicallySpawnedNodeOnLevel(int nLevel);</td>
<td>returns the last spawned node on the given level (useful for dynamic partitioning when using global variables)</td>
</tr>
<tr>
<td>int GetCurrentHighestDynamicPartitioningLevel(void);</td>
<td>returns the current highest dynamic partitioning level (for the current subtree) – the level for the next dynamic partitioning activity</td>
</tr>
<tr>
<td>void SetCurrentHighestDynamicPartitioningLevel(int nLevel);</td>
<td>sets the current highest dynamic partitioning level</td>
</tr>
<tr>
<td>void SetMinimumDynamicPartitioningLevel(int nLevel);</td>
<td>sets the minimum dynamic partitioning level</td>
</tr>
<tr>
<td>void SetMaximumDynamicPartitioningLevel(int nLevel);</td>
<td>sets the maximum dynamic partitioning level</td>
</tr>
<tr>
<td>int GetMaximumDynamicPartitioningLevel(void);</td>
<td>returns the maximum dynamic partitioning level</td>
</tr>
<tr>
<td>void SetMaximumDynamicPartitioningLevelLeftBranch(int nLevel);</td>
<td>sets the maximum dynamic partitioning level for the left outermost branch</td>
</tr>
<tr>
<td>int GetLastDynamicPartitioningLevel(void);</td>
<td>returns the last dynamic partitioning level</td>
</tr>
<tr>
<td>void SetDACMinimumNodeIndex(int nNodeIndex);</td>
<td>sets the minimum node index (refers to all levels) for which dynamic partitioning is allowed</td>
</tr>
</tbody>
</table>

** Managing Global State during Dynamic Process Partitioning**

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual void PackGlobalVariablesToChild(Object *vector_l, Object *vector_r, int nLevel);</td>
<td>packs global variables; nLevel means the level at which data should be packed</td>
</tr>
<tr>
<td>virtual void UnPackGlobalVariablesFromParent(Object *vector_l, Object *vector_r);</td>
<td>unpacks global variables received from the parent</td>
</tr>
<tr>
<td>virtual void PackGlobalVariablesToParent(Object *vector_l, Object *vector_r);</td>
<td>packs global variables to be sent as results to the parent</td>
</tr>
<tr>
<td>virtual void UnPackGlobalVariablesFromChild(Object *vector_l, Object *vector_r);</td>
<td>unpacks global variables (results) from a spawned process</td>
</tr>
</tbody>
</table>

**Asynchronous Update Messages Exchanged between Subtrees**
Many DAC-based examples are discussed in detail in Paragraph 7. In this paragraph a generic template for use with class DAC is presented. Basically, programming with the class does not require any parallel programming knowledge although it can prove useful and be incorporated into the code by setting parameters used during dynamic process partitioning. The programmer needs to create a class which inherits from class DAC, then instantiate an object of this class, set its parameters and run the object. Overridden methods of the class should provide functionalities typical of the divide-and-conquer paradigm such as partitioning the initial vector into subvectors, making a decision whether the current vector should be partitioned further, how to merge results etc. Available methods are given in Table 1. A basic framework is presented in Figure 3.

DAMPVM/DAC uses the notion of a vector which represents the initial data which is then recursively divided into subvectors and passed to successive recursive calls. vector\_l and vector\_r denote left and right boundaries of the current vector which can hold data of any type. If needed, it allows to use global variables and fetch data associated with the node of the tree currently being visited. The supported code is linked with a library of functions and executed with assistance of the runtime part of DAMPVM/DAC. Subtrees can be spawned as new processes, dynamically allocated and migrated to other nodes at runtime, if needed. Each node in the DAC tree receives a data vector delimited by left and right pointers Object *vector\_l and Object *vector\_r. Object is a template parameter for the abstract class DAC and a user-derived class from DAC should instantiate Object with a class/type suitable for its needs e.g. double for sorting vectors of double numbers. Vector (vector\_l, vector\_r) is either executed as a terminal node (if method DaC\_Terminate(vector\_l, vector\_r) returns true) or is divided further into subvectors by method DaC\_Divide(vector\_l, vector\_r) which returns a list of left and right pointers to the subvectors. In the first case method DaC\_LeafComputations(vector\_l, vector\_r) provides leaf computations and in the latter case DaC\_PreComputations(vector\_l, vector\_r). The procedure is repeated recursively. Upon return method DaC\_PostComputations (new vectors, vector\_l, vector\_r) provides code merging subvectors into the parent vector. This scheme allows arbitrary trees including various numbers of subvectors at each node and variable depths (unbalanced trees) depending on an application’s needs (e.g. computation accuracy may determine the depth). A simple example how to use the basic methods is shown for the simple DAMPVM/DAC implementation that computes \( \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \) recursively (Figure 4).

### Table 1: DAMPVM/DAC Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>void RegisterUpdateFunction(int (*func-</td>
<td>this function registers a global function given by a pro-</td>
</tr>
<tr>
<td>tion)(int));</td>
<td>grammer to be called when update messages arrive from</td>
</tr>
<tr>
<td>void SendUpdateMessage(int nLevel);</td>
<td>other subtrees</td>
</tr>
<tr>
<td>virtual void PackUpdateMessage(int nLevel);</td>
<td>this method sends update messages to all other processes</td>
</tr>
<tr>
<td>int ReturnChildSpawnLevel(PCTid pctid);</td>
<td>spawned at level nLevel if nLevel&gt;0; if nLevel==0</td>
</tr>
<tr>
<td></td>
<td>then the update message is sent to the parent; the method</td>
</tr>
<tr>
<td></td>
<td>can be invoked from any place in the code</td>
</tr>
<tr>
<td></td>
<td>packs necessary data for the update message – should be</td>
</tr>
<tr>
<td></td>
<td>overwritten by a programmer</td>
</tr>
<tr>
<td></td>
<td>returns the level at which the given child has been</td>
</tr>
<tr>
<td></td>
<td>spawned; returns DAC_CHILD_NOT_SPAWNED if it has not</td>
</tr>
<tr>
<td></td>
<td>been spawned</td>
</tr>
</tbody>
</table>

Fig. 2: DAMPVM/DAC Main Methods
```cpp
#include "DaC.H" /* DAMPM header file */
#include "myDAC.H" /* myDAC class definition */
MyDAC *mDCGoBangDAC; // a global pointer to the main object
void PackState() { packs the current state of the process for migration }
void UnPackState() { unpacks the current state of the process (after migration) }
int UpdateMessage(int mid) { // this is a message which updates
    PC_UPkInt(&nMessageDirection); // i.e. the node index and the level
    if (nMessageDirection==DAC_UPDATE_MESSAGE_TO_PARENT) PC_UPkPCTid(&ptid);
    unpack data;
}
int nSpawnedLevel = mDCGoBangDAC->ReturnChildSpawnLevel(ptid);
else update data; // I am a child of the calling process
    update the estimated work for this process due to the updates
    PC_Instructions(new estimated work for the process); return 0; // success
}
void MyDAC::PackUpdateMessage(int nLevel) { pack data for the update message };
void MyDAC::InitializeData(float * &tab_l, float * &tab_r) { // initializes data
    tab_l=(float)[fLevel[1][0]]; // initialize the vector
    tab_r=(float)[fLevel[1][2]]; // using some variables
}
void MyDAC::PackGlobalVariablesToChild(float *vector_l, float *vector_r, int nLevel) { // packs global variables; it is possible to get the data
    // associated with the particular node at which the tree is partitioned
    // i.e. the node index and the level
    pack data[GetLastDynamicallySpawnedNodeOnLevel(GetLastDynamicPartitioningLevel())];
}
void MyDAC::UnPackGlobalVariablesFromParent(float *vector_l, float *vector_r) { // unpack global values from the parent:
    initialize global variables using the unpacked values;
}
void MyDAC::PackGlobalVariablesToParent(float *vector_l, float *vector_r) {
    PC_pk*(&result)); // pack global result(s) to be sent to the parent
}
void MyDAC::UnPackGlobalVariablesFromChild(float *vector_l, float *vector_r) {
    // unpack global results from a child
    update the tree at the level the data was received and also global variables;
}
main(int argc, char **argv) {
    DaC_Init(&argc, &argv);
    mDCGoBangDAC=new MyDAC(&PC_PkFloat,&PC_UPkFloat); // it is necessary to
    // pass packing and unpacking functions as the input
    mDCGoBangDAC->SetCurrentHighestDynamicPartitioningLevel(2);
    mDCGoBangDAC->SetMinimumDynamicPartitioningLevel(1);
    mDCGoBangDAC->SetDACMinimumNodeIndex(2); // dynamic repartitioning allowed from
    // the third node on the current level
    mDCGoBangDAC->SetMaximumDynamicPartitioningLevel(1);
    mDCGoBangDAC->SetMaximumDynamicPartitioningLevelLeftBranch(2); // sets
    // the maximum dynamic partitioning level for the left outmost branch
    mDCGoBangDAC->EnableMigration(); // disable generally for this process
    mDCGoBangDAC->RegisterUpdateFunction(UpdateMessage); // registers a function
    // for updates of alpha and beta
    mDCGoBangDAC->Run(); // run the object
    DaC_Finish();
}
```

Fig. 3: A Basic DAC Framework
```c
double sum = 0; double fData[3]; // initial values of n, result and k
int MyDAC::Dac_HowManyNodes(double *vector_l, double *vector_r) { return 2; }
double ***MyDAC::DaC_Divide(double *tab_l, double *tab_r) {
    new_vectors[0] = *tab_l - 1; new_vectors[1] = 0; new_vectors[2] = *tab_r;
    tab_pointers[0] = new_vectors; tab_pointers[1] = new_vectors + 2;
    return tab_pointers;
}
void MyDAC::DaC_PostComputations(double **new_tab, double *&tab_l, double *&tab_r) {
    *(tab_l + 1) += (*new_tab)[1] + (*new_tab)[4];
}
bool MyDAC::DaC_Terminate(double *tab_l, double *tab_r) {
    return (((*tab_r) == 1) || ((*tab_l) <= (*tab_r)));
}
void MyDAC::DaC_LeafComputations(double *tab_l, double *tab_r) {
    if (((*tab_r) == 1) && (*tab_l) + 1 == (*tab_r)) /* leaf computations */
    if (((*tab_l) <= (*tab_r)) && (*tab_l + 1) == 1;
}
void MyDAC::MasterReport(double *vector_l, double *vector_r) {
    // report data e.g. send it to the standard output
    cout << *(vector_l + 1); // this should be our result
}
main(int argc, char **argv) {
    DaC_Init(&argc, &argv);
    if (PC_How_Started() != migrated) {
        fData[0] = 35; fData[1] = 0; fData[2] = 13; // n, result and k
    }
    PC_MigrationDisable();
    MyDAC mdcQSortDAC(PC_PkDouble, PC_UPkDouble);
    mdcQSortDAC.DisableMigration(); // disable generally for this process
    mdcQSortDAC.Run();
    DaC_Finish();
}
```

Fig. 4: Parallel (\(n^3\)) Implementation in DAMPVM/DAC

5 DAMPVM/DAC Tuning Mechanisms

5.1 Asynchronous Update Messages Exchanged between Subtrees

Although DAMPVM/DAC assumes that the order in which calls on a given level are executed does not change final results it may change the time required to analyze the tree as certain calls may use intermediate results produced by others. In \(\alpha \beta\) search the values of \(\alpha\) and \(\beta\) produced by previous calls can cut off large subtrees which results in shorter execution time ([57]). DAMPVM/DAC offers asynchronous messages which can be sent to parallel processes spawned before by the same parent (Figure 6). Thanks to a handler function registered by the programmer, a process can intercept an asynchronous message, update its data and thus narrow the search space by cutting off subtrees. An example which requires this mechanism is a parallel \(\alpha \beta\) search engine presented in Paragraph 7.1. Methods supported by the DAC scheme are shown in Table 1. A process can register a handler function by method RegisterUpdateFunction(int (*function)(int)) which is invoked when an update message from another process arrives. The handler function should unpack an integer which indicates whether the receiving process is the parent (constant \(DAC\_UPDATE\_MESSAGE\_TO\_PARENT\)) or a child (constant
DAC::UPDATE_MESSAGE_TO_CHILD) of the calling process. Figure 3 presents a framework of an application that shows how it can be done. It is possible to find out the level at which the calling child was spawned so that this information can be used in the updates. Naturally, there is a trade-off in this mechanism as update messages generate additional communication overhead but can pass important intermediate results to other processes at runtime and e.g. narrow their search space. This is the case in parallel chess, checkers or GoBang as explained in Paragraph 7.1.

5.2 Managing Tree Levels and Node Indexes and Partitioning Processes of Predefined Size

The implemented scheme enables to set maximum levels at which dynamic partitioning can be invoked (Figure 5). There is a general maximum partitioning level for all top-level branches except the outermost left one and a special maximum partitioning level only for the leftmost top-level branch. This prevents from spawning too short tasks with small subtrees. According to YBWC ([58]) in $\alpha \beta$ search subtrees should wait for the values of $\alpha$ and $\beta$ from the leftmost subtree. The special partitioning level for the leftmost top level branch is usually set to higher values than the general one for the others. This enables to partition the leftmost top level branch while the others are waiting. In later stages of computations the maximum level is lower as all top level branches can be spawned. In order to parallelize the execution of the leftmost subtree, the levels can be higher than for the other ones. Moreover a minimum index of a node which may be partitioned at a given level can be set. Setting it to 2 prevents from spawning tasks at a certain level if the leftmost node at this level is still being executed (Figure 5). Additionally, DAMPVM enables to partition only tasks larger than a predefined threshold. Tasks can report their sizes using method DAC::DaC::VectorSize() provided by the programmer. Although impossible to predict accurately for applications like the search this prevented small subtrees to be spawned as very short processes. This in turn prevented scenarios in which communication and synchronization costs were larger than parallelization gain.

5.3 Managing Global State during Dynamic Process Partitioning

Sometimes it is not enough to store the data associated with the given level as a vector. In particular, in games like chess it is necessary to store the board. DAMPVM/DAC includes special methods to support the management of global variables (Table 1). Often it can be more efficient to use for complex data structures. Table 1, the framework in Figure 3 and the internal DAMPVM/DAC implementation in Figure 7 show how it can be done. There are methods to pass data between parent and child processes and also to identify the particular node and the
template <class Object> // main recursive method
void DAC<Object>::
DaC(Object * &vector_l, Object * &vector_r) {
if (DAC has been requested for higher level than current) {
spawn a child; send the vector:
PackGlobalVariablesToChild(nPartLevel_vector_l, nPartLevel_vector_r,nPartLevel);
}
if (the highest depth) {
receive data from parent; unpack the vector:
UnPackGlobalVariablesFromParent(vector_l,vector_r);
DaC_Initialize(vector_l,vector_r);
}
if (DaC_Terminate(vector_l,vector_r))
DaC_LeafComputations(vector_l,vector_r);
else {
DaC_PreComputations(vector_l,vector_r);
 nHowManyNewVectors=
DaC_Divide(vector_l,vector_r);
if (more tasks needed) {
spawn tasks; send the vector:
PackGlobalVariablesToChild(nPartLevel_vector_l, nPartLevel_vector_r,nPartLevel);
inform DAMPVM that my size=
DaC_VectorSize(new_vectors[0],new_vectors[1]);
DaC(new_vectors[0],new_vectors[1]);
} else {
if (no tasks have been spawned)
enable migration for this process;
for(int nTask=0;nTask<nHowMany
NodesExecutedByThisTask;nTask++)
DaC(new_vectors[2*nTask],
 new_vectors[2*nTask+1]);
}
if (data has been sent to children) {
inform DAMPVM I may be idle waiting;
receive data; unpack the vector:
UnPackGlobalVariablesFromChild(new_vectors[child][vector_l],new_vectors[child][vector_r]);
}
DaC_PostComputations(new_vectors,vector_l,vector_r);
}
if (the highest level and I am not the root) {
pack the vector:
PackGlobalVariablesToParent(vector_l,vector_r);
send data to parent;
}
}
5.4 Detection of Idle Nodes through Automatic Process Instrumentation

Thanks to the capabilities incorporated into DAMPVM and thus hidden in the DAMPVM/DAC scheme the user does not have to worry whether a node becomes idle as a result of idle processes. A process executes all its non-spawned subbranches on a certain level and then waits for all spawned subtrees. Then it determines the final value of the whole subtree and continues with other subtrees.

However, it can become idle quite often. Since the execution time of a subtree may not be known in advance it is difficult to cope with such irregularities. Technically such a process calls DAMPVM function `PC_Wait()` which lowers its estimated number of instructions `instr(i)` to 0. This also updates the estimated execution time of the node this process is running on `est(t)`. Finally, after the process has received results from its children it automatically updates the number of instructions using function `PC_Instructions()` which updates the value of `est(t)` on the same node. All the described actions are done automatically and are hidden in the DAC scheme and thus are transparent to the user.

![Fig. 8: DAMPVM/DAC: Idle Node Detection](image)

5.5 Setting Estimated Work and Process Sizes at Runtime

It is important to notice that the DAMPVM instrumentation functions can still be called inside the DAMPVM/DAC code for more efficient instrumentation and thus process allocation and migration. As an example the framework shown in Figure 3 updates the size of the process in the message handler function (function `PC_Instructions()` due to the update information it has received. It changes the estimated amount of remaining instructions for this process (as modeled by parameter `instr` explained in Paragraph 3). Such modifications can tune e.g. the size of the tree estimated by a simple heuristic function like $2^{max \text{ level}}$ where `max level` denotes the depth of the tree. Similarly it is possible to set the size of the process which is used by DAMPVM kernels to determine whether the process should be migrated or not (the trade-off between load balance and migration costs).

6 Automatic Parallelization in DAMPVM/DAC

DAMPVM/DAC uses both dynamic process allocation and migration available in the DAMPVM layer (Figure 1) and implements dynamic process partitioning at the DAMPVM/DAC application/system level. This was done by additional modifications in DAMPVM kernels and special handler functions in DAMPVM/DAC applications. If a DAMPVM kernel decides to partition a process it sends a message which is intercepted by the process which spawns a child. As explained in [54] the system tries to use process migration before dynamic partitioning is invoked in order not to create additional tasks when enough tasks are available in the system. DAMPVM/DAC balances load “in advance” i.e. immediately when a considerable imbalance shows up. When the estimated execution time `est(t)` drops to a low value then dynamic partitioning is called.

Migration is heterogeneous since it is based on two functions for each process: packing and unpacking the current state (the values of important variables, tables etc. instead of machine registers or stack segments). A programmer should write the functions and the code in such a way so that the process can restart from the interrupted point after migration. The disadvantage of this method is a bigger programming effort. In return the
programmer gets heterogeneous and fast migration which is not available in many other systems. DAMPVM detects the presence of other users, the loads and speeds of the nodes as well as predicts the forward work for processes to achieve minimum execution time of an application. Two simple load balancing algorithms have been developed and analyzed ([55]) and more can be plug in very easily as explained in Paragraph 3.

DAMPVM kernels can be easily modified so that more than one process can be mapped to a node and thus hide communication latency. This may be an efficient solution for irregular applications as spawned tasks can be very short-lived and terminate quickly. If only one task is mapped to a node then idle time occurs before a new task is spawned/migrated on/to this node. DAMPVM/DAC can be configured so that it tries to execute at least a predefined number of tasks per node. This technique proved efficient for $\alpha\beta$ search engines described in Paragraph 7.1.

When a new task is spawned at a certain level, the execution in the parent process cannot proceed until results are obtained from the child process. However, DAMPVM/DAC can at least skip the nodes executed by spawned processes and execute all the nodes at this level before the results are gathered. It may be useful to initiate receive operations in the parent process while it still computes to overlap communication and computations rather than invoke receives when all communications are finished.

7 Partitioning and Tuning Divide-and-Conquer Applications in DAMPVM/DAC

The presented applications were implemented in DAMPVM/DAC, tuned as described below and executed on a LAN network on up to 16 SUN workstations. The results are shown in Figures 10 and 11 while the typical sizes of application trees in Table 2. Maximum partitioning levels and asynchronous messages were required only by the most complex example which was the $\alpha\beta$ search. All could be optimized by providing accurate estimation of computational costs of subtrees assigned to processes which were used by DAMPVM/DAC schedulers. Largest processes were partitioned first.

7.1 $\alpha\beta$ Search Engines

As an example, a GoBang game was implemented in which two players put crosses and circles in turn on the board. The one who has first put its five pieces vertically, horizontally or diagonally wins. [57] explains three problems that prevent parallel implementations of the $\alpha\beta$ scheme from reaching high speed-ups: synchronization overhead/idle time, parallelization overhead due to communication between processors, process creation etc. and the search overhead mentioned above. Since subtrees are spawned as separate processes/threads they need to be synchronized. When a process finishes and there is no available work to spawn idle time occurs which deteriorates performance. This is particularly visible in networks of workstations where communication latency is large. When subtrees are started in parallel too early are not able to use the $\alpha$ and $\beta$ values computed by other subbranches immediately although the asynchronous messages try to compensate for that. The experiments (Figures 10 and 11) showed that the maximum partitioning level of 3 for the leftmost subtree and 2 for the others were optimal. Sets of 2 and 1 resulted in idle time for larger numbers of processors and higher levels in too small tasks being spawned. In the latter case large synchronization overhead greater than parallelization gain was observed. The minimum node index was set to 2 in order to force YBWC. The DAMPVM scheduler has been modified so that it sends dynamic partitioning requests to tasks larger than a certain threshold (60 for maximum depth 8 for position in Figure 9). The size of a task is determined using method $\text{DAC::DaC::VectorSize()}$ which was set to $2^{\text{max depth}}$. It is not accurate for irregular trees as similar positions can lead equally well to a “mate”, “stalemate” or other complex positions. Nevertheless this change has improved performance significantly. The main problem is that the depth of the tree is low (Table 2) and dynamically spawned trees can be very small which causes large synchronization delays with the parent process. A simple move reordering ([59]) strategy was implemented:

1. Perform a quick shallow search sequentially (depth 6 for the position in Figure 9). No parallelization is done at this stage (DAC is disabled).
2. Put the best move at the beginning of the first level move list.
3. Execute the deeper search (of depth 8) in parallel.

[36] and [59] discuss iterative deepening and transposition tables. The above algorithm intentionally sacrifices some time by executing the shallow search. It compensates with shorter total execution time thanks to move reordering and using better \( \alpha \) and \( \beta \) from the best move from the shallow search for cut-offs. [36] describes Scout search with zero-size \( \alpha \beta \) windows in order to check whether a candidate move is better than the current best and its parallel version – Jamboree search. Move reordering can be applied to all moves at a given level.

7.2 Recursive Fibonacci and \( \binom{n}{k} \)

Although computing Fibonacci numbers recursively is inefficient it is regarded as a good benchmark for parallel divide-and-conquer systems since it is very fine-grained i.e. there are no computations in the leaves. Thus the Fibonacci example can describe the overhead of the parallel scheme compared to the best sequential implementation (Paragraph 8) very well. Both applications are easily parallelized by DAMPVM/DAC due to large and reasonably balanced trees. Since the trees have large depths in the order of 1-40 (Table 2) even subtrees which are dynamically spawned at higher levels are still large. This implies that vector sizes could be set even to fixed values. Imbalance is detected by DAMPVM/DAC schedulers in a later stage of computations and partitioning is invoked again. Communication and synchronization costs are low compared to computational costs as opposed to unpredictable and often small sizes of subtrees in \( \alpha \beta \) search. Figures 10 and 11 show results for \( F(43) \) and \( \binom{35}{13} \) respectively.

7.3 Finding Twin Primes

The goal is to find out how many twin prime numbers are in range \([a, b]\). The initial range \([a, b]\) (\([1000432, 50000540]\) for the results in Figures 10 and 11) is divided into two parts which are then divided recursively until their lengths have reached a predefined number of elements (10000 in Figures 10 and 11). This example is analogous to the adaptive integration example in this respect ([53]). However, the speed-up is a bit better for the prime example due to a better balanced tree compared to integrating irregular functions. For large enough ranges of large numbers the tree is more balanced than for the irregular integration shown in [53] which results in a slightly better speed-up (Figure 10). The size of a vector is equal to the number of elements which may be a good estimate in this case. Notice that this problem was not formulated recursively. Alternatively, the initial vector could be divided statically into as many subvectors as the number of available processors. However, if a task is migrated to another machine when one is recaptured by the owner a considerable imbalance occurs since tasks can not be partitioned. The recursive formulation adds overhead due to recursive calls (Paragraph 8) but maintains the dynamic partitioning capability which enables to produce smaller tasks and achieve a smaller imbalance.

7.4 Adaptive Integration and Image Recognition

These examples are analyzed in detail in [53] and [54] respectively. Comparing regular images (uniform patterns across the image, results for 4000x4000 pixels, 8-bit depth shown in Figures 10 and 11) to a smaller ROI (Region of Interest, 400x400, 8-bit depth in Figures 10 and 11) can be compared to the twin prime example with respect to the speed-up and the size of the tree (Table 2). The trees are reasonably large and balanced. Recognition of irregular images is similar to integration of irregular functions (the function takes different amounts of time for different subranges to be integrated, \( \int_0^{2\pi} \sin^2(x)dx + \int_{2\pi}^{4\pi} (x - 2\pi)dx \) in Figures 10 and 11) due to unbalanced trees. In the latter case, the tree is deeper (Table 2) so it can be partitioned to smaller processes and mapping achieves a smaller imbalance. The size of a vector is set to the number of pixels in the assigned part of the image or the analyzed range respectively. [54] shows that migration is not profitable for large trees which can be divided into reasonably large subtrees by dynamic partitioning (like integration).
Fig. 10: Speed-up for the Implemented Divide-and-Conquer Applications

Fig. 11: Execution Time of the Implemented Divide-and-Conquer Applications
8 DAMPVM/DAC Overhead and Optimizations

DAMPVM/DAC introduces additional overhead to the recursive mechanism including:

1. tracking dynamic partitioning requests from DAMPVM schedulers,
2. storing: the pointers, the number of vectors and indexes of analyzed nodes at various levels, highest dynamic partitioning levels for next DAC invocations,
3. setting process sizes at runtime: after new tasks are spawned, before and after data is gathered from processes spawned at the currently visited level.

The overhead can be evaluated by comparing the execution time of an application using DAMPVM/DAC and the best possible static implementation, both executed on one processor. Some applications execute long computations in leaves which makes the overhead insignificant compared to the execution time of even the best possible serial implementation. Others like Fibonacci are very fine-grained for which the overhead at each level is significant compared to computations in the algorithm. All the presented examples have been implemented without DAMPVM/DAC as well. The results are presented in Table 2. The worst (highest) \( \frac{t_1}{t_2} \) ratios were obtained for the Fibonacci and \( \binom{n}{k} \) examples. This is because there are practically no computations performed in the leaves, the trees are reasonably large and the overhead described above is considerable compared to real divide-and-conquer operations executed at each level of recursion. [50] compares the overhead of a similar Satin system where invocation records need to be registered to other implementations like Cilk ([48]) or ATLAS ([49]). Thresholding techniques similar to those described below are mentioned. DAMPVM/DAC shows the overhead very similar or lower than Satin.

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem Size</th>
<th>Exec. time without DAC (( t_2 ))</th>
<th>Exec. time with DAC (( t_1 ))</th>
<th>Ratio ( \frac{t_1}{t_2} )</th>
<th>Node degree</th>
<th>Tree depths tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive Integration</td>
<td>( \int_0^\infty \sin^2(x)dx )</td>
<td>111.6</td>
<td>115.8</td>
<td>1.04</td>
<td>2</td>
<td>( \approx 1-25 )</td>
</tr>
<tr>
<td>Alpha Search</td>
<td>shallow search depth=4, actual search depth=7</td>
<td>52</td>
<td>52</td>
<td>1</td>
<td>( \approx 5-30 )</td>
<td>&lt; 10</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>( F(37) )</td>
<td>70.5</td>
<td>105.2</td>
<td>1.46</td>
<td>2</td>
<td>( \approx 1-40 )</td>
</tr>
<tr>
<td>( \binom{100}{13} )</td>
<td>100.5</td>
<td>139.2</td>
<td>1.39</td>
<td>2</td>
<td>( \approx 1-30 )</td>
<td></td>
</tr>
<tr>
<td>Twin Primes</td>
<td>range [1000432, 5000540]</td>
<td>81.1</td>
<td>84.1</td>
<td>1.04</td>
<td>2</td>
<td>( \approx 10 )</td>
</tr>
<tr>
<td>Image Recognition</td>
<td>image: 1000x1000, ROI: 40x40, leaf limit: 60 pixels</td>
<td>112.8</td>
<td>118.9</td>
<td>1.05</td>
<td>4</td>
<td>( \approx 1-5 )</td>
</tr>
</tbody>
</table>

Table 2: DAMPVM/DAC Overhead and Typical Trees Tested

Migration in DAMPVM is based on the PVM 3.4 message handlers ([60]). Signals are caught as soon as any PVM function is called ([61]). Thus the current DAC implementation contains calls to \texttt{pvm_nrecv()} which in fact do not receive any messages. It was shown that no visible overhead is introduced when this function is called up to level 15 for narrow but very deep trees like the Fibonacci example and up to level 6 for wide and shallow trees like the GoBang example. These values do not restrict the dynamic partitioning capability since the subtrees below these levels are reasonably short and trees are always partitioned from the root so they do not reach these limits in practice.

9 Conclusions and Future Work

It was shown that the Fibonacci, \( \binom{n}{k} \) twin prime examples and irregular applications like integration of irregular functions, recognition of irregular images benefit from efficient automatic partitioning in DAMPVM/DAC due
to reasonably large trees and the ease of evaluating the computational costs of subtrees. On the other hand, in the \( \alpha/\beta \) search there is a trade-off between the reduction of execution time due to parallelization and the costs: both the search (reduced by asynchronous update messages) and synchronization/communication overhead. Experimentally derived maximum levels and the minimum size of a process for partitioning try to prevent synchronization costs higher than the parallelization gain. Future work will be focused on optimizations of the internal DAMPVM/DAC code (Figure 7) including communication between parent and child processes, hiding communication latency and modeling process sizes. At a higher level remote clusters running DAMPVM/DAC will be merged using tools like PHP or Java servlets.

References

4. Ohio Supercomputer Center, The Ohio State University: MPI Primer/Developing with LAM. (1996)


