Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters

JUDITH HIPPOLD[*] and GUDULA RÜNGER
Chemnitz University of Technology
Department of Computer Science
{juh, ruenger}@informatik.tu–chemnitz.de

Abstract

Clusters of SMPs (symmetric multiprocessors) are popular platforms for parallel programming since they provide large computational power for a reasonable price. For irregular application programs with dynamically changing computation and data access behavior a flexible programming model is needed to achieve efficiency. In this paper we propose Task Pool Teams as a hybrid parallel programming environment to realize irregular algorithms on clusters of SMPs. Task Pool Teams combine task pools on single cluster nodes by an explicit message passing layer. They offer load balance together with multi-threaded, asynchronous communication. Appropriate communication protocols and task pool implementations are provided and accessible by an easy to use application programmer interface. As application examples we present a branch & bound algorithm and the hierarchical radiosity algorithm.

1 Introduction

Irregular algorithms may have unknown access behavior to data structures, varying computational effort, or irregular dependences between computations resulting in a strongly input dependent program behavior. Examples for irregular applications include sparse problems like sparse Cholesky factorization, adaptive finite element methods, and algorithms creating hierarchical data structures like the hierarchical radiosity algorithm or branch & bound algorithms. Adaptive and hierarchical algorithms have been designed to reduce computation time and memory needs, however, parallel computers are still required to produce realistic results in reasonable time.

The characteristics of irregular algorithms may cause load imbalances and irregular communication pattern which often make an efficient parallel implementation difficult. Although irregular algorithms have several characteristics in common, their actual irregular behavior may differ tremendously. Many branch & bound algorithms show irregular computation structures with few data dependences between them, so that load balance is more important for a parallel realization than communication. Sparse or adaptive problems exhibit stronger data dependences and a parallel implementation has additionally to deal with communication to provide intermediate results. However, the communication requirements are moderate in nature since most remote data accesses are specified and allow optimizations, like collecting

[*]Supported by DFG, SFB393 *Numerical Simulation on Massively Parallel Computers*

data or loading data in advance, which decrease the irregular behavior of the communication process. In contrast, some hierarchical algorithms, e. g. the hierarchical radiosity algorithm, show a highly-dynamic computation behavior which is determined by the current entire set of data and where remote accesses have to be performed immediately preventing optimizations. This kind of algorithms requires the most flexible computation model since asynchronous irregular communication has to be realized together with load balancing issues.

This paper introduces *Task Pool Teams* (TPTs) a generalization of the task pool concept for implementing irregular algorithms on SMP clusters. Task Pool Teams are especially convenient for algorithms with asynchronous and unpredictable communication needs where the tasks strongly depend on each other and need frequent accesses to irregularly growing, local and remote data structures. A Task Pool Team combines several task pools each running on a specific SMP node by explicit message passing. Task pools are a common concept to realize dynamic load balance for irregular algorithms on shared memory platforms. For the use of task pools in the TPT context we have implemented a set of different pools and have enabled the migration of tasks between task pools. Explicit communication between task pools is used for irregularly occurring, remote data accesses. To handle different communication requirements different communication protocols can be used. The advantage of explicit communication is mainly the support of optimizations on application level which is especially useful for very irregular and unpredictable communication needs. The application programmer interface (API) of Task Pool Teams provides functions for using task pools as well as functions for multi-threaded and asynchronous communication between the pools.

The contribution of this paper is a hybrid parallel programming environment for highly-irregular problems with strong inter-dependences between computations. TPTs offer flexibility to realize asynchronous computation and communication structures together with load balance. Still the API makes Task Pool Teams easy to use for the application programmer. The resulting implementation of an irregular algorithm with Task Pool Teams is entirely realized on the application programmers level so that the characteristics of the specific algorithm can be exploited for an efficient realization. TPTs provide a variety of task pools, different communication protocols, and two-level load balancing mechanisms from which the application programmer can choose the appropriate one. The careful selection of a suitable task pool realization and load balancing level can improve performance.

The paper is structured as follows: Section 2 introduces the Task Pool Team programming environment. It briefly discusses task pools, presents the TPT concept, including communication schemes and communication mechanisms, and introduces the application programmer interface. Section 3 describes the load balancing mechanism provided by Task Pool Teams. Section 4 presents the implementation of two application examples, the hierarchical radiosity algorithm and a branch & bound algorithm, with TPTs and gives experimental results. Section 5 discusses related work and Section 6 concludes.

2 Task Pool Teams programming environment

This Section introduces Task Pool Teams an MPI and Pthreads based, hybrid programming environment. TPTs generalize task pools a well-known concept for realizing irregular algorithms on shared memory platforms. The next subsection gives a brief overview on task pools. Detailed information are presented in [22, 23, 25, 28].

2.1 Task pools

Application programs realized with task pools on SMPs are structured into a set of interacting tasks. Each task consists of a well-defined sequence of commands often captured in a function or procedure to be executed by a thread. The commands can include the dynamic creation of new tasks which can later be executed by a different thread.

A task pool is a shared data structure to store and manage the tasks created during a specific program run. All threads of a process have access to the task pool. They extract tasks from the task pool queue(s) for execution or insert tasks into the pool if the currently executed task creates new child tasks. The cooperation of tasks possibly executed by different threads is realized via the common shared address space of the process where the data of the program are stored. To guarantee conflict-free accesses to memory locations the execution of *lock* and *unlock* operations is necessary.

The entire task program can be executed by a fixed number of threads, also if the number of tasks is changing during program execution. This minimizes the overhead for thread creation and maps a varying number of tasks onto a fixed number of threads yielding a dynamic load balance for the execution.

There are several possibilities to implement task pools. They can differ for example in the number of queues or the access strategy to queues. Concerning the number of queues *central task pools* with only one task queue and *decentralized pools* with a separate queue for each thread are possible. Because the decentralized versions do not offer implicit load balance, *task stealing* mechanisms can be realized in addition. Access strategies for task queues are for example the *FIFO* (first-in first-out) or *LIFO* (last-in first-out) principle.

2.2 Task Pool Team approach

Programming clusters of SMPs using the task pool concept demands interaction between the pools on different cluster nodes in order to exchange data and to balance load. Multi-threaded and asynchronously running processes resulting from a task-oriented program structure require more flexible communication than offered by pure MPI. The particular requirements for the communication process are thread-safety, multi-threading, and asynchronism. Most current implementations of MPI2 do not provide the full functional range needed, e. g. they lack support for multi-threaded communication (*MPI_THREAD_MULTIPLE mode*) or for completely synchronization-free one-sided communication calls (*passive target*). Task Pool Teams build up a uniform interface providing the user with the required properties in order to be platform independent.

Task Pool Teams combine the task pools running on each single cluster node by mutual communication. At the beginning and at the end of a program run only the main threads of the cluster nodes are active. They can use MPI operations arbitrarily, for example to distribute data or to collect data from other cluster nodes. The program switches to multi-threaded mode when the main threads start the processing of tasks and switches back to single-threaded when they leave their pools.

During the multi-threaded mode communication between task pools is realized with the communication mechanism provided by TPTs. It includes a *communication thread* (CT) for each task pool. A separate communication thread is necessary to provide a communication partner for each remotely started operation. The threads working on the task pools (*worker threads*) cannot handle this communication demand because their asynchronous termination and irregular code execution may provoke deadlocks.

2.3 Communication in Task Pool Teams

The communication thread is responsible for handling the communication needs during the multi-threaded sections of the program. To avoid conflicting requirements for the CT, which might occur when sending requests interferes with the receipt of asynchronously arriving messages, the worker threads (WT) are enabled to send messages. Thus there are two distinct kinds of thread interactions to manage: the local synchronization of WTs and CT and the handling of remote communication needs. The resulting communication schemes are illustrated in Figure 1. We distinguish between *simple* and *complex communication* schemes to provide suitable schemes for different kinds of irregular applications.

Figure 1 a) illustrates the simple communication scheme. It can be used e. g. to exchange intermediate results without a preceding request by a worker thread. Simple communication consists of two steps: *(1) A worker thread of Node 0 sends calculated data to Node 1. (2) The communication thread of Node 1 receives and processes the data.*

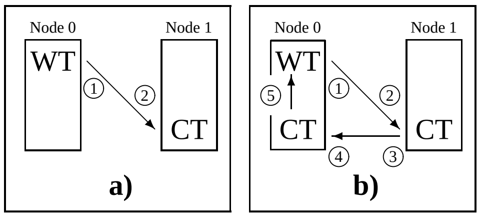
Complex communication occurs for example if intermediate results or data structures situated on a remote cluster node are needed by a worker thread of another cluster node, to continue calculations. For complex communication the following steps are performed as illustrated in Figure 1 b):
(1) A worker thread of Node 0 needs data situated in the address space of Node 1. Thus it sends a request and waits passively. (2) The communication thread of Node 1 receives the request, reacts accordingly, and (3) sends the data back. (4) The CT of Node 0 receives the message and (5) signals the waiting worker thread that the requested data are available.

```
Pseudo-code 1

while(1) {
        tag = message_check();
/* complex communication */
        if (tag == 1) {
            initial_receive();
            /*... get data ...*/
            final_send();
        }
        else if (tag == 2)
            final_receive();
/* simple communication */
        else if (tag == 3) {
            receive_data();
            /*... process data ...*/
        }        . . .
}
```

To express the communication needs of a specific application the programmer provides a function describing the communication pattern to be performed by the communication thread. Pseudo-code 1 illustrates the core structure of this function with examples for complex and simple communication. /*... ...*/ denotes algorithm-specific code. A pointer to that function is given as a parameter to the TPT initialization function tp_init (see Subsection 2.4.1, Function (i)).
TPTs provide the blocking function message_check to test for incoming messages. If there is a message available, it returns the tag of the message. According to this message tag the CT selects a specific action to perform. The application programmer chooses an arbitrary, odd, and unique integer as tag for a specific type of request. A communication thread receiving a message with an odd tag automatically increments this tag and uses it for the response message. Thus the pair $(2i+1, 2i+2), i \in I\!N$, labels a complex communication process of a specific type. The simple communication process needs only one, odd tag because there are no final_send and final_receive operations necessary.

a)
1. send_data
2. receive_data



b)
1. initial_send
2. initial_receive
3. final_send
4. final_receive
5. awake WT

Figure 1: Simplified illustration of a) simple and b) complex communication between two task pools only showing the involved worker thread and the communication threads.

2.4 Application programmer interface

This subsection presents the application programmer interface of Task Pool Teams and briefly explains the usage. We distinguish interface functions to use the pools designed for Task Pool Teams and functions to use the TPT communication mechanism. The usage within a specific application is illustrated in Subsection 4.2.1.

2.4.1 User interface for task pools

The interface to use task pools consists of five functions ((i)-(v)). Functions (i)-(iv) are performed by the main thread. Depending on the specific task pool implementation the main thread either becomes a worker thread after calling tp_run or waits passively until the processing of tasks is finished. When the task pool is empty, the main thread prepares the pool for re-use and returns from tp_run. There is no need to destroy the pool and to create a new one when it might be used again, which saves initialization time. If there is no exchange of messages between different task pools, the pools should run without a communication thread (WITHOUT_CT) for efficiency reasons. Function (v) is used by the main thread before calling tp_run to insert the initial tasks into the task pool. During the processing of tasks each thread working on the pool is able to use this function in order to insert dynamically created tasks.

(i) **void** tp_init (**unsigned** no_of_threads, **void** (* ct_routine)(), **long** sleep_dur, **unsigned** no_of_cpus) ;
allocates and initializes the task pool. no_of_threads denotes the total number of threads including the main thread and the communication thread. ct_routine points to the function performed by the CT. The parameters sleep_dur and no_of_cpus serve optimization purposes and denote the duration of inactivity (in μs) for the communication thread and the number of CPUs per SMP.

(ii) **void** tp_mig_init (**int** count, **void** (* task_routine)(), **int** size, **void** (* copy1)(), **void** (* copy2)()) ;
initializes task migration. count is the total number of different function definitions to be executed as tasks, task_routine points to the function definition to be initialized, and size is the size of its arguments in byte. copy1 and copy2 are pointers to user-provided functions for copying the arguments.

(iii) **void** tp_destroy ( ) ;
deallocates the task pool structure and destroys all threads except the main thread.

(iv) **void** tp_run (**unsigned** type) ;
starts the processing of tasks. Depending on the type (WITH_CT or WITHOUT_CT) the task pool runs with or without a communication thread.

(v) **void** tp_put (**void** (* task_routine)(), **void** *arguments, **unsigned** queue) ;
inserts the task task_routine with the arguments arguments into queue queue of the pool.

2.4.2 User interface for TPT communication

The interface functions for simple and complex communication are presented in the following. We distinguish between the cases: a) the size of incoming messages is not known and b) the size of incoming messages is known in advance. The provided functions encapsulate MPI and Pthread operations. The blocking function **int** message_check(); enables the CT to test for incoming messages as illustrated in Pseudo-code 1 of Subsection 2.3.

Simple communication

(1) **void** send_data(**void** *buffer, **int** count, **MPI_Datatype** type, **int** dest, **int** tag) ;
sends the data in buffer of length count and data type type to process dest. tag denotes the message.

(2) a) **void** *receive_data(**int** *source, **int** *count, **MPI_Datatype** type) ;
receives the data of data type type. After finishing this function, the sender and the message size are stored in source and count. A pointer to the received data is returned.
b) **void** receive_data_buf(**void** *buffer, **int** *source, **int** count, **MPI_Datatype** type) ;
receives the data of data type type and length count in the user provided buffer buffer. After finishing this function, the sender is stored in source.

Worker threads call function (1) and continue with their computations. Communication threads perform function (2) and process the received message.

Complex communication

(1) **void** *initial_send(**void** *buffer, **int** count, **MPI_Datatype** type, **int** dest, **int** tag, **unsigned** thread_id);
sends a data request to processor dest. The pointer buffer denotes the data of length count and data type type necessary to identify the requested data. tag denotes the message in order to initiate a specific action on the destination process. The function returns a pointer to the requested data.

(2) a) **void** *initial_receive(**int** *count, **MPI_Datatype** type) ;
receives the data request and returns a pointer to the received data. After finishing initial_receive, count contains the length of the received data of type type.
b) **void** initial_receive_buf(**void** *buffer, **int** count, **MPI_Datatype** type) ;
receives the data request. The user provides the buffer buffer of length count for receiving.

(3) **void** final_send(**void** *buffer, **int** count, **MPI_Datatype** type) ;
sends the requested data in buffer of length count and data type type back to the requesting process.

(4) a) **void** final_receive(**MPI_Datatype** type) ;
receives the requested data of data type type and awakes the waiting worker thread.
b) **void** final_receive_buf(**void** *buffer, **int** count, **MPI_Datatype** type) ;
receives the requested data of data type type and length count in a user provided buffer buffer and awakes the waiting worker thread.

Function (1) initiates a data request. The calling WT is blocked until the requested data are available. Functions (2) and (3) are performed by the remote communication thread which provides the data. The requested data are received by the local CT with Function (4) and are returned to the requesting worker thread by Function (1). The synchronization between requesting WT and local CT (step (5) of Figure 1) is hidden to the user and implemented within functions (1) and (4).

3 Dynamic load balancing

Task Pool Teams offer a two-level dynamic load balancing. On a single SMP cluster node balanced load is achieved by central task pools or decentralized pools with task stealing. To reduce imbalances between the task pools running on the different cluster nodes a *migration* mechanism for tasks is provided. Task migration may improve performance, however, there is a trade-off between the additional overhead to migrate tasks and the reduced execution time due to balanced load. The performance of task migration further depends on the properties of the specific application and the underlying platform. This section introduces the TPT task migration mechanism which is based on an internal synchronization hierarchy.

3.1 Internal synchronization

Irregular applications implemented with Task Pool Teams consist of different, asynchronously running program parts on different cluster nodes. To guarantee that the CTs handle the entire data requests, synchronization is necessary before switching from multi-threaded to single-threaded mode. Task Pool Teams ensure this by a hierarchy of *local* and *global synchronization* completely hidden to the user. Figure 2 illustrates this hierarchy on two cluster nodes for the most complex case where the main threads act as WTs.

Local synchronization takes place between the main thread and the other local worker threads and is necessary to avoid that the local CT is stopped by the main thread while worker threads are still waiting for non-local data. Figure 2 illustrates the phase of local synchronization by dashed lines. The first dashed line of each node indicates that the first WT has finished and the second line shows that all local WTs have finished their tasks. If all threads working on the local pool are ready, the main thread returns. Task Pool Teams realize several barriers for local synchronization. Pseudo-code 2 shows the most complex case where the main thread processes tasks together with the other worker threads.

Pseudo-code 2

Worker threads entering the barrier wait passively except the last thread which awakes the main thread and blocks, unless it is the main thread itself. Then it returns.

```
waiting_threads += 1;
if (waiting_threads<number_of_WT) wait();
else {
    if (main_thread) return;
    else {    signal_main_thread();
              wait();    }
}
```

Global synchronization guarantees that the CTs have handled the entire communication needs of remote worker threads before they are stopped. The basis for global synchronization is a preceding local synchronization step. The mechanism is illustrated by the Pseudo-code 3 and Figure 2 (dotted lines).

Pseudo-code 3

After leaving the task pool each main thread increments the synchronization counter and sends a synchronization message to each other task pool. Then it waits passively. The communication thread increments the local synchronization counter according to the number of received synchronization messages. It awakes its local main thread when the counter equals the total number of task pools. After that it waits passively for re-activation.

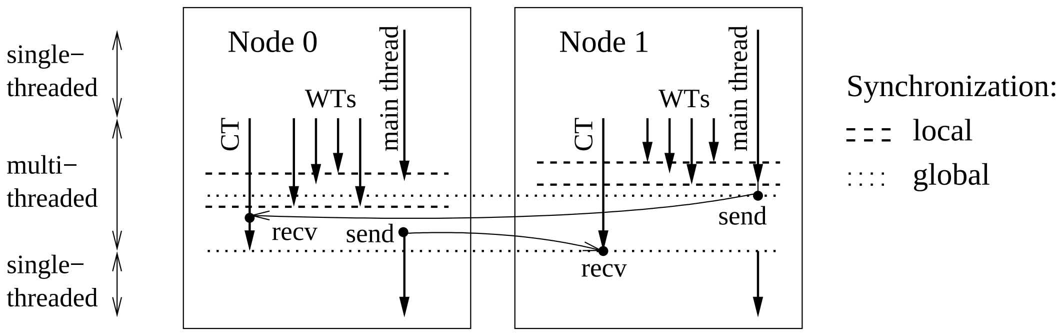


Figure 2: Synchronization hierarchy illustrated with two cluster nodes. Dashed lines label local synchronization between the worker threads and the main thread of a task pool after finishing the processing of tasks. Dotted lines show global synchronization between the different pools.

```
/* main thread */
while (pool_is_not_empty)
    { /* ... process tasks ... */ }
sync_count++;
foreach (remote_task_pool) {
    send_sync_message();
}
while (CT_run) wait();
```

```
/* communication thread */
while (sync_count != max_task_pools) {
    /* ... handle communication ... */
    if (sync_message) sync_count++;
}
CT_run = FALSE;
signal_main_thread();
while (!CT_run) wait();
```

3.2 Task migration

The task migration mechanism takes place between local and global internal synchronization. The migration interface function tp_mig_init provides Task Pool Teams with the necessary information about the different function definitions to be executed as tasks in the task pools (see Subsection 2.4.1, Function (ii)). Each different definition of the application needs to be initialized by a separate call to tp_mig_init. Pseudo-codes 4 and 5 show one possible realization for the exchange of tasks. The TPT complex communication mechanism is used for clear illustration although the migration of tasks does not take place on programmer's level but completely hidden using the basic concepts of complex communication.

Pseudo-code 4

```
while (!stop) {
        stop = 1;
        foreach (remote_task_pool) {
                task = initial_send();
                if (task != NULL) {
                        tp_put(task);
                        stop = 0;
}       }       }
```

After all tasks of the local task pool are processed the main thread sends a request for task migration to the other remote task pools. If all task pools are empty, the processing of tasks is completed and the main thread starts global synchronization. If there are migrated tasks, the main thread inserts them into the pool using tp_put*. This automatically starts the processing of tasks again.*

Pseudo-code 5

During migration communication threads have to handle two additional types of messages: requests for task migration and task data. If a communication thread receives a request, it extracts a task from the local pool and sends it back to the requesting cluster node. In case the local pool is empty, it submits this information. If a communication thread receives a task (or NULL*), it awakes the waiting local main thread which then returns from* initial_send.

```
while (1) {
    tag = message_check();
    if (tag == TASK_REQU) {
        initial_recv();
        task = NULL;
        if (pool_is_not_empty)
            task = get_task();
        final_send(task);
    }
    else if (tag == TASK_RECV)
        final_recv();
}
```

The proposed task migration concept is completely dynamic and automatically adapts to the actual program run. For Task Pool Teams different migration mechanisms are realized which follow the proposed concept but differ in the implementation details and so satisfy the needs of different applications.

4 Application examples and experiments

As application examples we chose the following two algorithms: the *hierarchical radiosity algorithm* (HRA) [6, 17] and a branch & bound algorithm solving the *traveling salesman problem* (TSP). This section introduces the implementation with Task Pool Teams and presents experimental results using the following parallel platforms:

SB1000	an SMP cluster of 4 SunBlade 1000 with 2 UltraSPARC3 processors (750 MHz) per node running Solaris and using an SCI network.
XEON	an SMP cluster of 16 PCs with 2 Intel Xeon processors (2.0 GHz) per node running Linux and using an SCI network.
HP9000	an SMP cluster of 3 HP9000-N4000 with 8 PA-8600 processors (550 MHz) per node running HP-UX and using Fast Ethernet.
CLiC	a Beowulf cluster consisting of 528 PCs with 800 MHz Intel Pentium III processors running Linux and using a Fast Ethernet network.

4.1 Hierarchical radiosity algorithm

The radiosity algorithm is an observer-independent global illumination algorithm from computer graphics to simulate diffuse light in three-dimensional scenes. The algorithm uses a geometric description (input polygons) of the scene with values of light emission and reflection coefficients. The input polygons are divided into smaller elements for which radiosity values (radiation per time and surface unit) are computed. Form factors describe the portion of light energy incident on an element from each other surface element. The computation of form factors is the most expensive part of the algorithm since it involves visibility tests and the computation of double integrals. The hierarchical approach couples the precision of computations with the exchange of energy between elements which reduces the number of form factors. This results in an uneven division of input polygons into smaller elements where the subdivision of elements depends on its size and the portion of energy incident from other elements. After finishing the algorithm the entire scene is represented by a set of quad-trees with at least one tree for

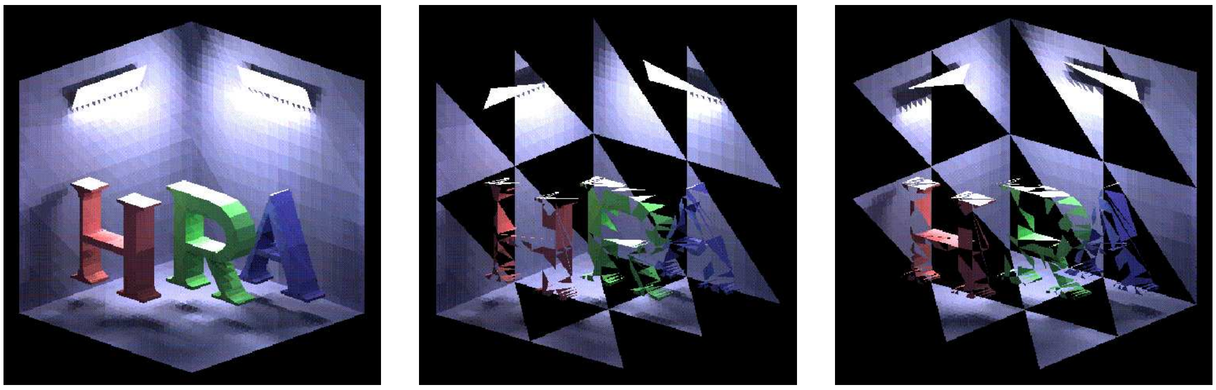


Figure 3: Left: resulting image for scene hra. Middle/right: scene hra computed in parallel on two cluster nodes using TPTs. Black, triangular areas denote patches computed on the remote cluster node.

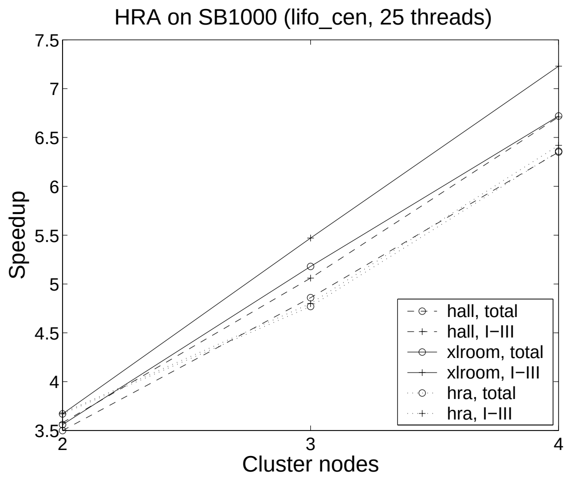
each input polygon. The leaf elements of all trees represent the surface to display but all levels of the quad-trees are required for computation.

4.1.1 Implementation with Task Pool Teams

As implementation basis we use the HRA of the SPLASH2 benchmark suite [33], modified by [22] and [25], and convert it for distributed memory and Task Pool Teams. The subdivision hierarchy is realized with irregularly structured quad-trees where trees are assigned to cluster nodes. To store the diverse interaction partners each element represented by a quad-tree node owns an interaction list which contains pointers to those elements providing portions of light energy. The algorithm consists of three phases. The phases I and III serve for initialization and smoothing of the resulting image. The phase II is the most time-consuming phase. It iteratively performs the computations on the quad-trees as described in the following.

The algorithm performs top-down passes over the quad-trees by creating tasks for each tree node and investigating the elements represented by the quad-tree nodes for refinement. The portions of energy incident from interacting elements are gathered and passed on to the children. If the current element is a leaf node of the quad-tree, the radiation is propagated to the top of the tree by a bottom up pass. During the top-down passes an element might interact with elements situated on different cluster nodes. This requires the exchange of energy values and possibly the subdivision of remote elements. Complex TPT communication is used for that purpose.

The performance of the HRA implementation is determined by sequential code portions and by the problem to eliminate load imbalances completely: During phase I the creation of quad-tree roots from input polygons is done. The number of quad-tree roots which represent one polygon is variable. Therefore a balanced initial distribution of work to cluster nodes according to the number of input polygons is not possible. Further imbalances emerge in phase II because of irregularly growing trees. Due to a strong dependence between tasks created for the nodes of a specific quad-tree data structure the migration of single tasks is cost-intensive. To reduce load imbalances and migration costs entire quad-trees are displaced between cluster nodes after each top-down-bottom-up pass rather than migrating single tasks. Thus the achievable balance is strongly input-dependent.



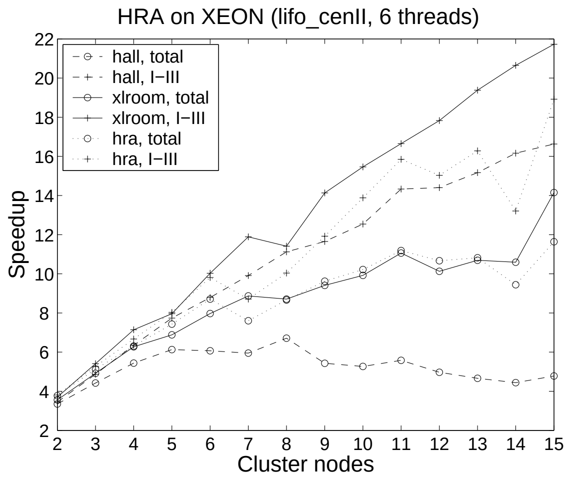


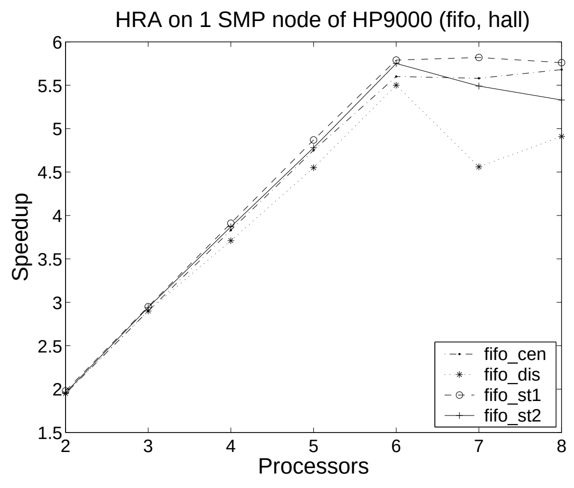
Figure 4: Speedups for scenes hall (dashed lines), xlroom (solid lines), and hra (dotted lines) on SB1000 (left) and XEON (right). + markers denote speedups for the phases I-III and o markers denote speedups for the entire algorithm.

4.1.2 Experimental results

For the HRA measurements with the input scenes hall (1157 initial polygons) and xlroom (2979 initial polygons) from [25] and hra (959 initial polygons, Figure 3) have been performed. The radiosity implementation running only the main thread with a lock free task pool implementation and one queue has been used to get sequential execution times.

Figure 4 shows speedups on SB1000 and XEON using the best-suited task pools. Each cluster node used runs one task pool and each pool employs the two available CPUs. For the largest scene xlroom (solid line) the speedups are slightly better than for hall (dashed line) and hra (dotted line) due to the higher workload per cluster node compared to the number of data exchanges. Lines with o markers denote speedups including time for initialization and redistribution. These speedup values are lower and scale less than those only regarding the time for phases I-III (+ markers). Reasons are sequentially calculated program parts whose portion on total execution time increases with growing number of cluster nodes. The behavior on HP9000 is similar. The phases I-III on HP9000 achieve for example values of 10.4 and 11.8 for hall and values of 14.2 and 17.3 for xlroom on 2 and 3 cluster nodes (pool tp_fifo_st1 with 8 threads). Task Pool Teams are also suitable for clusters with only one CPU per node because they support an overlapping of computation and communication: If more threads than real CPUs are assigned per cluster node, some worker threads are able to wait passively for remote data while the other threads utilize the full capacity of the node for computations. For example on CLiC speedups of 2.0 and 16.4 can be achieved for scene xlroom on 2 and 20 nodes (pool fifo_cenII with 3 threads).

Figure 5 shows speedups achievable with task pools on a single SMP of HP9000 and illustrates the influence of different task pool implementations on program execution time. In most cases the pools with LIFO access principle (right diagram) have slightly better speedups than the FIFO task pools (left diagram) which might be caused by cache effects. The pools without load balancing (fifo_dis, lifo_dis) have the highest execution times and show the strongest speedup decreases for large numbers of threads due to load imbalances. The central pools fifo_cen and lifo_cen have good performance but suffer from



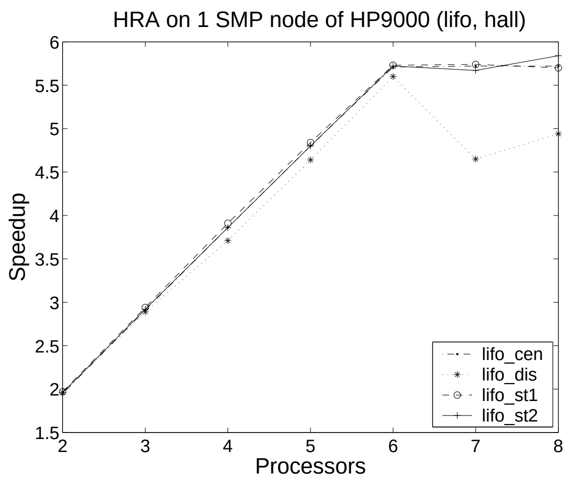


Figure 5: Speedups for scene hall on a single cluster node of HP9000 using task pools with FIFO (left) and LIFO (right) access principle.

access conflicts to the central queue. They can be outperformed by decentralized pools with task stealing, especially fifo_st1 and lifo_st1, which reduce the number of access conflicts by decentralized queues and avoid strong load imbalances by task stealing. Speedups reach a saturation point with about 6 threads for all task pool versions due to the increasing thread management overhead.

Figure 6 illustrates the influences of the communication thread and the total number of threads per pool on execution time. We present separate speedups for the phases of the HRA without communication thread (phases I+III) and with communication thread (phase II). The total number of threads includes the CT and is 3 for the left diagram and 6 for the right diagram. Thus there are 2 WT and 1 CT or 5 WT and 1 CT, respectively, running on the 2 CPUs of each cluster node. For phase II using the WITH_CT mode, a number of worker threads greater than the number of CPUs per SMP cluster node can improve performance slightly. Reasons are the reduced CPU time assigned to the CT and the overlapping of computation and communication. In contrast the phases running without communication thread achieve better speedups when the number of worker threads equals the number of CPUs per cluster node. The task pools fifo_cenII and lifo_cenII address this problem as the two peaks for phases I+III on the right of Figure 6 show. These pools use as many worker threads as CPUs per SMP when they are running in the WITHOUT_CT mode. Otherwise the user-defined number of threads is active.

TPTs provide the asynchronous communication mechanism needed for the scalable parallel realization of the HRA on clusters. The selection of a suitable task pool and appropriate input parameters speeds up the implementation. Regarding the entire algorithm the investigated platforms show mostly the best performance with the LIFO task pools which automatically adapt the number of active worker threads according to the requirements and use more threads than CPUs per cluster node.

4.2 Branch & bound algorithm

The traveling salesman problem is a well-known optimization problem which searches for the shortest round-trip of a salesman within a given set of cities. The symmetric TSP, where the paths from city *A* to

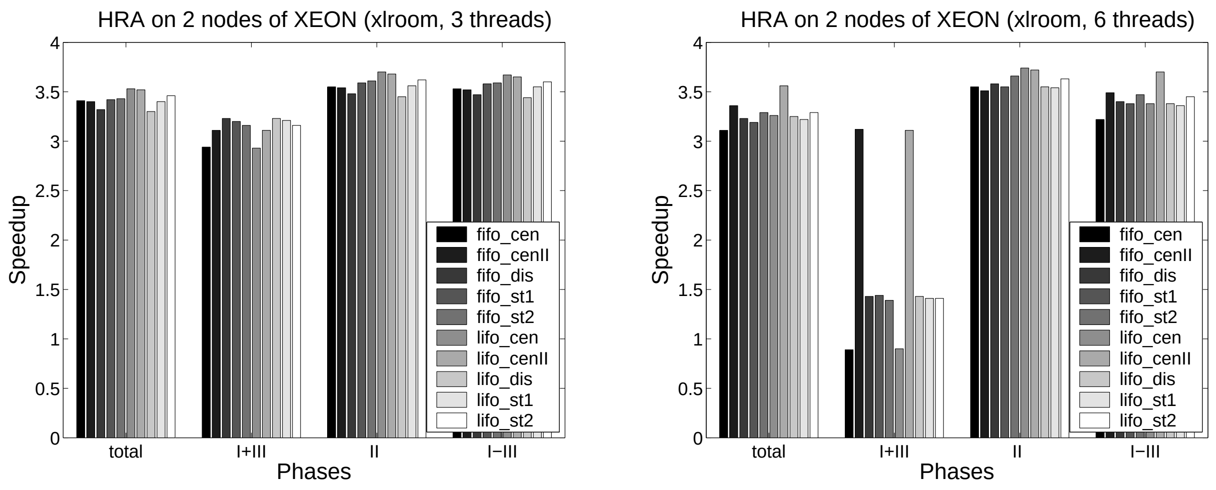


Figure 6: Speedups for the algorithmic phases of the HRA on 2 cluster nodes of XEON using 3 threads (left) and 6 threads (right) per cluster node. (total: entire algorithm, I+III: phases without CT, II: phase with CT, I-III: phases I,II, and III)

city *B* and city *B* to city *A* are identical, and the asymmetric TSP, where the paths from *A* to *B* and *B* to *A* differ in costs, can be distinguished. The implementation introduced in this paper solves the asymmetric variant and is based on a branch & bound strategy. We have realized a backtracking algorithm exploiting the enumeration principle which provides a suitable application to test Task Pool Teams rather than an optimized solution for the traveling salesman problem. The algorithm regards a treelike solution space and tries to find the solution stepwise by canceling incorrect steps.

4.2.1 Implementation with Task Pool Teams

The solution space is represented by a tree. This imaginary tree is assigned to cluster nodes by dividing it into disjoint subtrees. Multiple contiguous tree nodes form one task. The task granularity has main influence on the performance. Small tasks might increase execution time for the task pools using lock-mechanisms due to growing access conflicts. Large tasks might increase execution time for the pools using stealing and migration mechanisms due to lacks of tasks to steal. A trade-off has to be found for efficient execution. Pseudo-codes 6 and 7 give the coarse structure of the implementation and illustrate the ease of using the Task Pool Teams library.

Pseudo-code 6

The main program is performed by the main threads. They make the necessary initialization and insert the initial tasks into each pool. The call of tp_run *starts the processing of tasks. After all tasks have been finished the main threads return and destroy the pools.*

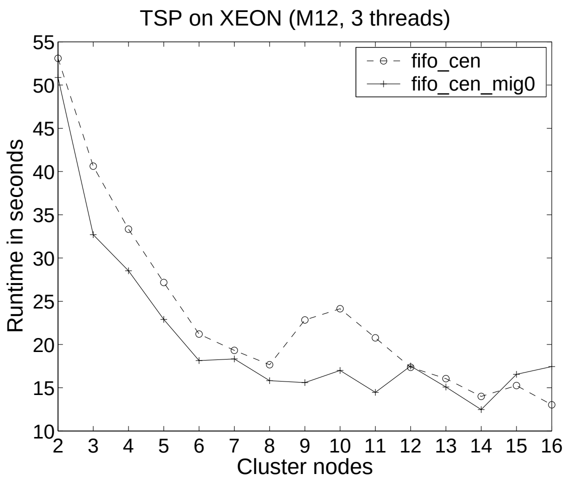
```
/* main program */
tp_init(tsp_ct_function);     tp_mig_init(tsp_task);      /* initialize the task pool and task migration */
/* ... distribute the search space among the cluster nodes ... */
while (not_max_initial_tasks) {                           /* insert the initial tasks into the pool */
    path = ... ;
    tp_put(tsp_task, path);
}
tp_run(WITH_CT);                                          /* start the processing of tasks */
tp_destroy();                                             /* destroy the task pool */
```

Pseudo-code 7

```
/* worker thread */                                  /* communication thread */
tsp_task(path) {                                     tsp_ct_function() {
    if (invalid_path(path)) return;                      while (1) {
    cost = determine_costs(path);                            tag = message_check();
    if (cost < min) {                                        if (tag == NEW_MIN) {
        if (minimal_tour(path)) {                                new_min = receive_data();
            min = cost;                                          if (new_min < min) {
            send_data(min, NEW_MIN);                                 min = new_min;
            return;                                              }
        }                                                    }
        foreach (city) {                                 }
            if (max_task_size())                     }
                tp_put(tsp_task, path+city);
            else tsp_task(path+city);
}   }   }
```

The processing of tsp_task *represents the search in the solution space and is performed by the worker threads. Depending on the chosen task pool implementation the subtrees spanning the solution space are searched in a depth or breadth-search-like order. Branches which do not contain a solution due to costs higher than the cost of the current minimal tour* min *or invalid paths are cut. If a new minimal tour is found, it is sent to all cluster nodes using simple communication. Otherwise a new city is added to the search path. Depending on the chosen task granularity a new* tsp_task *is created or the function representing the task is called recursively. The communication thread performs the function* tsp_ct_function*. It receives new minimums found on the remote cluster nodes and updates the local minimum if it is larger than the new one.*

Task Pool Teams enable the immediate exchange of calculated minimal tours among the different cluster nodes. Thus in addition to local cuts the number of tasks to compute can be significantly reduced by cutting remote invalid paths. Subtree cuts create strongly non-deterministic program runs which are determined by the sequence in which the tasks are processed.



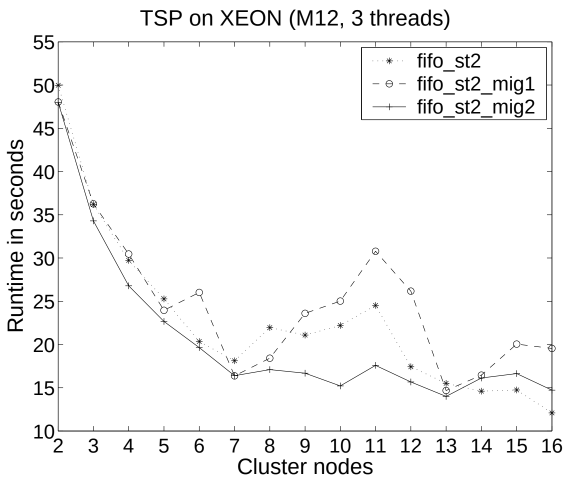


Figure 7: Comparison of runtimes for task pools with (_mig0, _mig1, _mig2) and without task migration on XEON using 3 threads per node and example M12.

4.2.2 Experimental results

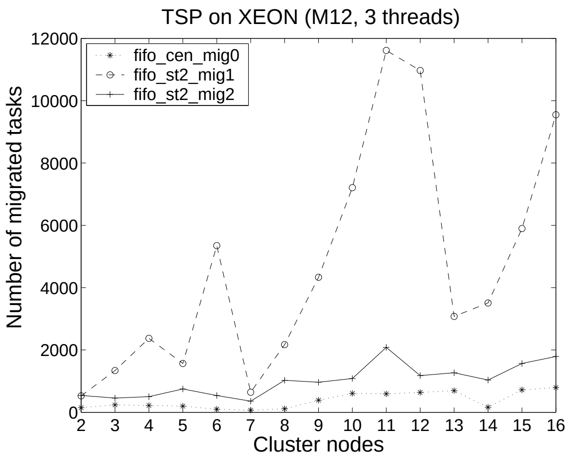
The input problem M12 used for the TSP is based on a 17 cities sample input for the asymmetric TSP created by B. Repetto which can be downloaded at [27]. For the measurements an average task size of 1000 search tree nodes has been chosen. Due to non-determinism different program runs for the same input problem create slightly different numbers of tasks. For the comparison of the different task pool versions an average runtime is used. The following experiments illustrate and compare the performance of the task migration mechanisms mig0-2 implemented for TPTs.

Figure 7 compares runtimes using task pools with and without task migration on XEON. The task pools fifo_cen_mig0 and fifo_st2_mig2 are faster than the corresponding versions without task migration when the number of cluster nodes is less than 13. For larger numbers of nodes the overhead for task migration becomes too high compared to the total execution time and the pools without task migration achieve in most cases better results.

On XEON the mig1 mechanism migrates some tasks multiple times between the cluster nodes before processing. Thus the overhead for task migration exceeds the performance gains and the task pool fifo_st2_mig1 has higher runtimes than fifo_st2. The left diagram of Figure 8 shows the resulting large number of task exchanges in comparison to the other task migration mechanisms. Reasons may be the small number of parallel working threads per cluster node which increase the probability that a newly received task is migrated again rather than extracted for processing by a worker thread.

On the right of Figure 8 the runtimes for input example M12 on HP9000 using 9 threads per cluster node are shown. In contrast to the measurements on 3 cluster nodes of XEON, fifo_st2_mig1 achieves better results than fifo_st2. A reason might be the 8 available CPUs per cluster node providing higher parallelism among the WTs than only 2 CPUs per node on XEON. The central task pool with task migration cannot outperform the corresponding version without task migration. This might be caused by the central queue and the resulting high numbers of access conflicts to remove a task for migration.

For the investigated example task pools with task migration can achieve better results than the corresponding versions without migration mechanism. The performance of the different pools using task



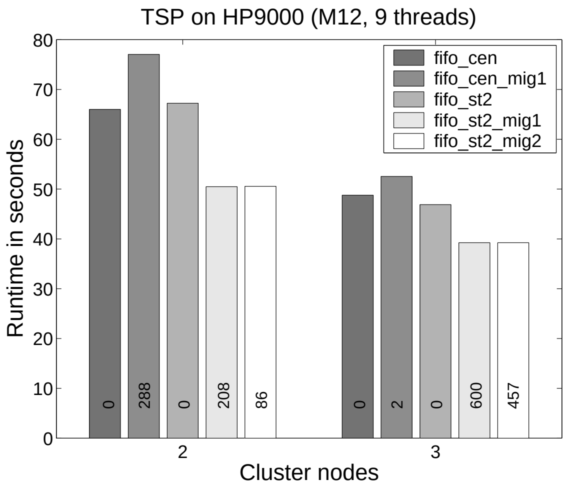


Figure 8: Left: total number of migrated tasks on XEON. Right: comparison of task pools with and without task migration on HP9000. The values inside the bars are the total number of migrated tasks.

migration is platform dependent. The main influence has the degree of parallelism offered by a single SMP cluster node of the target architecture. TPTs provide a task migration concept adaptable to the algorithm and the platform. This flexibility allows to select the best implementation. The explicit TPT communication further enables the immediate exchange of new minimums which narrows the search space significantly.

5 Related work

Programming models for SMP clusters can be classified into two main categories: *uniform* and *hybrid* [11]. In the uniform model the memory organization is hidden and the user sees either only shared memory (virtual shared memory) or only distributed memory. *Message-passing* supports the distributed memory view and is provided for example in libraries like MPI [9], MPI2 [10], or PVM [29]. Software distributed shared memory (DSM) systems, e. g. TreadMarks [1], MGS [24], or Orca [3], simulate a coherent shared memory and enable *shared-memory* programming for SMP clusters. [4, 15, 18, 21] support the shared memory programming library OpenMP [5] on top of DSM systems. Threads on distributed memory are provided by runtime environments like CableS [19], Chant [16], or Nexus [8]. Although a uniform model might simplify the parallel realization of applications it prevents to exploit the different characteristics of both memory organizations on application programmers level. Especially for irregular algorithms an explicit view can be useful to support implementation specific optimizations, e. g. the reduction of irregular communication or cost effective load balancing according to memory organization.

Hybrid programming models combine shared-memory programming with message-passing and may offer a more efficient implementation strategy for many application problems. Hybrid programming is supported for example by SIMPLE [2], NICAM [30], or [14] which combines MPI with the Nano-threads programming model (NPM) [26]. In contrast to the hybrid Task Pool Teams, SIMPLE and NICAM separate communication from computation phases. NPM is task-oriented but based on a parallelizing compiler thus targeting on applications with more regular parallelism.

The parallel extension of programming languages creates a software layer and provides a more abstract view on the architecture than offered by clear hybrid or uniform models. Multi-threaded extensions of C suitable for irregular applications are the prototypical implementation of Distributed Cilk [12] and Threaded-C of the EARTH system [31]. The Java dialect Titanium [34] and the C++ based Charm++ [20] are suitable for distributed memory, but not specifically designed for SMP cluster architectures. Based on Charm++, [13] discusses mechanisms to use SMP clusters efficiently by a uniform programming model. Task Pool Teams are situated on a lower level using the common library implementations of MPI and POSIX threads, thus releasing the user from learning extensive language modifications and concepts and enabling the application programmer to optimize the algorithm according to the underlying architecture.

Similar to Task Pool Teams the PMESC [7] library is designed for irregular task-parallel problems. PMESC provides pools for tasks on distributed memory and is not specifically developed for SMP clusters. Thus the library covers actual message passing and machine architecture so preventing algorithm specific optimizations offered e. g. by Task Pool Teams. As PMESC the LilyTask program framework [32] supports task parallelism using pools and is not specifically designed for SMP clusters. In contrast to TPTs, LilyTask adopts the BSP model which excludes remote data accesses during the processing of tasks.

6 Conclusion

We have introduced Task Pool Teams together with an easy to use interface which provides a functionality suitable for the parallelization of highly-irregular algorithms with strong data-intensive dependences on clusters of SMPs. Our model offers multi-level dynamic load balance which can be chosen independently and adapted to the algorithmic needs. Task Pool Teams provide asynchronous communication for the exchange of data during task processing and support overlapping of computation and communication.

The flexibility of the TPTs programming environment enables application specific optimizations, as to distribute data structures according to their access pattern or choosing suitable TPT parameters, like the appropriate task pool version or load balancing level. Exploiting algorithm-specific details helps to achieve an efficient implementation especially for problems with highly-irregular behavior. Even for some applications only the careful selection of the task pool, load balancing level, or communication strategy can show speedups. TPTs have been investigated on clusters of SMPs and PC clusters. The efficiency results are good but depend on platform specifics like the MPI implementation, thread management, or the communication network.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] D. A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[3] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[4] A. Basumallik, S. J. Min, and R. Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *Proc. of Intl. Workshop on OpenMP: Experiences and Implementations (WOMPEI'02), LNCS 2327*, pages 457–468, 2002.

[5] OpenMP Architecture Review Board. www.openmp.org, 2005.

[6] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. *ACM SIGGRAPH Computer Graphics*, 22(4):75–84, 1988.

[7] S. Crivelli and E. R. Jessup. The PMESC Programming Library for Distributed-Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, 57(3):295–321, 1999.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[10] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[11] W. W. Gropp and E. L. Lusk. A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters. In *Proc. of Conf. on Programming Models for Massively Parallel Computers (PMMP'95)*, pages 2–7, 1995.

[12] Supercomputing Technologies Group. *Cilk 5.1 Reference Manual*. MIT Labratory for Computer Science, 1997.

[13] A. Gürsoy and I. Cengiz. Mechanisms for Programming SMP Clusters. In *Proc. of Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1723–1729, 1999.

[14] P. E. Hadjidoukas, E. D. Polychronopoulos, and T. S. Papatheodorou. Integrating MPI and Nanothreads Programming Model. In *Proc. of Euromicro Workshop on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP'02)*, pages 309–316, 2002.

[15] P. E. Hadjidoukas, E. D. Polychronopoulos, and T. S. Papatheodorou. OpenMP Runtime Support for Clusters of Multiprocessors. In *Proc. of Intl. Workshop on OpenMP Applications and Tools (WOMPAT'03), LNCS 2716*, pages 180–194, 2003.

[16] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proc. of ACM/IEEE Conf. on Supercomputing (SC'94)*, pages 350–359, 1994.

[17] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. *ACM SIGGRAPH Computer Graphics*, 25(4):197–206, 1991.

[18] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[19] P. Jamieson and A. Bilas. CableS : Thread Control and Memory Management Extensions for Shared Virtual Memory Clusters. In *Proc. of Intl. Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 263–274, 2002.

[20] L. V. Kale and A. B. Sinha. Information Sharing Mechanisms in Parallel Programs. In *Proc. of Intl. Parallel Processing Symposium (IPPS'94)*, pages 461–468, 1994.

[21] M. B. S. Karlsson and S. W. Lee. A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory. In *Proc. of Intl. Conf. in High Performance Computing (HiPC'02), LNCS 2552*, pages 195–206, 2002.

[22] M. Korch and T. Rauber. Evaluation of Task Pools for the Implementation of Parallel Irregular Algorithms. *Proc. of ICPP'02 Workshops, CRTPC'02*, pages 597–604, 2002.

[23] M. Korch and T. Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[24] J. Kubiatowicz, A. Agarwal, and D. Yeung. MGS: A Multigrain Shared Memory System. In *Proc. of Intl. Symposium on Computer Architecture (ISCA'96)*, pages 44–55, 1996.

[25] A. Podehl, T. Rauber, and G. Rünger. A Shared-Memory Implementation of the Hierarchical Radiosity Method. *Theoretical Computer Science*, 196(1-2):215–240, 1998.

[26] C. Polychronopoulos. Nano-threads: Compiler Driven Multithreading. In *Proc. of Intl. Workshop on Compilers for Parallel Computing (CPC'93)*, 1993.

[27] TSPLIB Gerhard Reinelt. www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95, 2005.

[28] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.

[29] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[30] Y. Tanaka, M. Matsuda, M. Ando, K. Kubota, and M. Sato. COMPaS: A Pentium Pro PC-based SMP Cluster and Its Experience. In *Proc. of IPPS'98 Workshops, SPDP'98*, pages 486–497, 1998.

[31] G. Tremblay, C. J. Morrone, J. N. Amaral, and G. R. Gao. Implementation of the EARTH Programming Model on SMP Clusters: a Multi-threaded Language and Runtime System. *Concurrency and Computation: Practice and Experience*, 15(9):821–844, 2003.

[32] T. Wang, N. Di, J. Shen, and Y. Tang. LilyTask Programming Model and Its Implementations on SMP & Cluster. In *Proc. of IASTED Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'03)*, pages 301–306, 2003.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of Intl. Symposium on Computer Architecture (ISCA'95)*, pages 24–36, 1995.

[34] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.