Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Judith Hippold, Gudula Rünger

Task Pool Teams for Implementing Irregular Algorithms on Clusters of SMPs

Preprint SFB393/02-18

Judith Hippold, Gudula Rünger
TU Chemnitz
Fakultät für Informatik
D-09107 Chemnitz

`http://www.tu-chemnitz.de/informatik/HomePages/PI`

Preprint-Reihe des Chemnitzer SFB 393
ISSN 1619-7178 (Print) **ISSN 1619-7186** (Internet)

SFB393/02-18 **Oktober 2002**

Contents

Task Pool Teams for Implementing Irregular Algorithms on Clusters of SMPs

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

Abstract

The characteristics of irregular algorithms make a parallel implementation difficult, especially for PC clusters or clusters of SMPs. These characteristics may include an unpredictable access behavior to dynamically changing data structures or strong irregular coupling of computations. Problems are an unknown load distribution and expensive irregular communication patterns for data accesses and redistributions. Thus the parallel implementation of irregular algorithms on distributed memory machines and clusters requires a special organizational mechanism for a dynamic load balance while keeping the communication and administration overhead low.

In this paper we propose task pool teams for implementing irregular algorithms on clusters of PCs or SMPs. A task pool team combines multithreaded programming using task pools on single nodes with explicit message passing between different nodes. The dynamic load balance mechanism of task pools is generalized to a dynamic load balance scheme for all distributed nodes. We have implemented and compared several versions for task pool teams. As application example, we use the hierarchical radiosity algorithm, which is based on dynamically growing quadtree data structures annotated by varying interaction lists expressing the irregular coupling between the quadtrees. Experiments are performed on a PC cluster and a cluster of SMPs.

Keywords: task pool teams, distributed memory, irregular algorithms, hierarchical radiosity algorithms, cluster and cluster of SMPs

1 Introduction

Irregularity of algorithms may be caused by different characteristics including unpredictable accesses to data structures due to sparsity or dynamic changes, varying computational effort because of refinement or adaptivity, and irregular dependencies between computations. Examples are sparse linear algebra methods like sparse Cholesky factorization, grid-based codes with dynamic refinements like adaptive FEM, or hierarchical algorithms like the fast multipole or hierarchical radiosity algorithm. Although most hierarchical and adaptive algorithms have been invented to

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

save computation time while still getting a good solution, there is still need for a parallel implementation since the computation time for those methods can be quite large when realistic problems are considered. The computational characteristics of irregular algorithms can be different, but all irregular algorithms have in common that the actual program behavior of a specific program run strongly depends on the specific input data of the program. Thus, not much information is statically available and static planning of parallelism is difficult. Especially when an irregular algorithm has unpredictably evolving computational work, dynamic load balance is required to employ all processors evenly.

For shared memory platforms the concept of task pools can be used to realize dynamic load balance. The basic idea is to manage tasks in a special global data structure called task pool, see e.g. [4] or [13]. Each processor can take tasks from the pool and add new tasks to the pool until the entire computation is done. Task pool implementations for dynamic load balance have been presented in e.g. [17, 12]. Detailed investigations of different task pool versions have been presented in [11].

For distributed memory machines there is a close connection between dynamic load balance and communication since a redistribution of work at runtime can only be achieved with communication. For an efficient parallel implementation a trade-off between decreased runtime due to balanced load and increased runtime due to communication overhead has to be found. Furthermore the unpredictable access behavior forces communication for the exchange of data structures.

This paper introduces the realization of irregular algorithms on PC clusters or clusters of SMPs (symmetric multiprocessors) with a new approach, called *task pool teams*. A task pool team combines the concept of task pools for single nodes with explicit communication for non-local data accesses and redistribution. The integration of multithreaded programming and communication results in a two level scheme. Load balancing on individual nodes is achieved by task pools for shared memory nodes and interaction between different nodes is reached by a dynamic communication scheme using a specific communication thread. Explicit communication is also used for the execution of remote data accesses occurring irregularly.

We have implemented task pool teams on top of the Message Passing Interface (MPI) and POSIX Threads (Pthreads), although the concept is more general. The MPI standard is used for communication and Pthreads are used to implement task pools. A thread-based implementation of task pools offers the advantages of fast accesses to shared data structures, low thread creation time, and a distribution of threads of one process to several processors. MPI guarantees portability, because MPI implementations are available on a wide range of parallel machines. Thus, the resulting implementation of an irregular algorithm with task pool teams is entirely realized on the application programmer level. The advantage is that the mechanism of task pool teams can be used for an efficient parallel implementation while the application programmer can still exploit algorithm specific properties explicitly.

Our hybrid programming model is suitable for PC clusters or clusters of SMPs and for applications with arbitrary task graphs or arbitrary dynamic data structures. We illustrate und test the model with the hierarchical radiosity method, a global illumination method from computer graphics. This application has several of the properties of irregular algorithms mentioned above. Data are organized in different dynamically growing quadtrees and computations are guided by varying interactions between nodes of those quadtrees. The resulting parallel algorithm is tested on two platforms, a Beowulf cluster (CLiC) and a cluster of SMPs. We have implemented several variants for task pools and experimented with different approaches for the combination with communication to form task pool teams. The task pools differ in the internal administration using one or more task queues and in the access strategy to queues. The application implementations differ in the number of threads created. The number of threads influences the execution time of the application, also in the case of the PC cluster.

Special care was taken to design the communication mechanism between remote threads running on different cluster nodes. General problems are the absence of thread-safe communication in the implementation environment or a potential of deadlocks for blocking communication operations in irregular thread-based programs. We introduce communication protocols which successfully deal with those issues thus guaranteeing safe communication within a task pool team while providing efficiency. Several communication patterns are discussed and have been implemented and the most efficient one has been used in the final implementation.

The paper is structured as following: Section 2 presents concepts and realizations of task pools. Section 3 introduces the task pool team approach. The implementation of the traveling salesman problem with our programming model is illustrated in Section 4. Section 5 describes the hierarchical radiosity algorithm (HRA) and discusses special requirements and adaptations with regard to the task pool team realization. Experiments and measurements are presented in Section 6. Section 7 discusses related work and Section 8 concludes.

2 Task pools for shared memory

For the implementation of task pool teams we assume the following basic programming model on a cluster: There is one process for each cluster node, which can be a one-processor machine or an SMP with several processors. A process consists of a virtual address space and one or more threads of control which are executed by timesharing on the processors of the node. The interaction between processes running on different nodes with different address spaces is realized by explicit message passing. The task pool team approach considers task pools for clusters with one task pool for each cluster node. A task pool manages the computation work of the corresponding node.

This section presents the task pool concept for a single cluster node. The actual combination of task pools with message passing and specific strategies for implementing task pool teams for entire clusters are introduced in the next Section 3.

2.1 General approach

Application programs realized with task pools 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 single thread of the process. The commands can include the creation of child tasks which can later be executed by a different thread of the same process. All tasks of a program form a graph of tasks with arrows expressing dependencies between tasks. Although the task creation and execution is coded within the implementation of the algorithm, each run of a specific program may actually create a different task graph, especially for irregular algorithms.

A task pool is a shared data structure to store and manage the tasks created for one specific program. All threads of the process executing the program have access to the task pool. They extract tasks from the pool for execution and insert tasks into the task pool if the currently executed task creates new child tasks. Corresponding access functions for the insertion of tasks into the structure and for removing tasks are provided. 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 realize the correct program behavior the execution of *lock* and *unlock* operations is needed to guarantee a conflict-free access to memory locations. Programming environments for shared address space usually provide such lock and unlock operations.

The entire task program can be executed by a fixed number of threads, also if the task graph of the program has a quite irregular structure and the number of tasks is varying during program execution. So for each process we create an arbitrary but fixed number of threads. Because the number of threads remains constant during runtime, the overhead for thread creation is minimized. Moreover this has the advantage that a varying number of tasks is mapped to a fixed number of threads yielding a dynamic load balance for the execution. The actual success of this load balancing strategy may depend on implementation details.

There are several possibilities for the internal organization of task pools and the storage of tasks. Often the tasks are kept in queues and task pools differ in the number of queues and the access strategy to queues. Concerning the number of queues the main cases are:

- Central task pool: Only one task queue holding tasks exists and all threads of the process access this queue to remove or insert tasks. To avoid access conflicts each access to the task queue has to be protected by a lock operation. Threads remove tasks from the queue for execution when they are ready with previous work and so the central queue offers good dynamic load balance. But frequent accesses to the queue, e.g. for many small tasks, may lead to sequentialization due to the lock protection.

- Decentralized task pool: (often also called distributed task pool) Each thread has its own queue from which only this thread can remove tasks and insert tasks. The advantage is that accesses to the queues do not have to be protected. On the other hand the static initialization of task queues may lead to imbalances if the initial tasks in the queues create an unequal number of new tasks, which is usually unknown in advance in the case of irregular applications. Special heuristics might be used to fill the queues at the beginning of the computation.

- Decentralized task pool with task stealing: This task pool variant of a decentralized task pool allows threads to access queues of other threads if its own queue is empty. This is called *task stealing* [16]. Task stealing avoids load imbalances but requires a locking mechanism for guaranteeing exclusive access to queues.

More variations of those main classes of task queue organizations within task pools are proposed and investigated in [11]. Another implementation decision concerns the order of inserting tasks into and extracting tasks from task queues. We distinguish the FIFO (first-in first-out) and LIFO (last-in first-out) access strategies for queues and have implemented those variations for different central and decentralized task pools which we describe in the next subsection.

2.2 Specific task pool implementations

We have implemented the following set of task pools:

(1) **tp_fifocen** is a central task pool with FIFO access strategy. The central queue is protected by a lock mechanism.

(2) **tp_lifocen** is a central task pool with LIFO access strategy. The central queue is protected by a lock mechanism.

(3) **tp_fifo** is a decentralized task pool with FIFO access strategy. No lock mechanism is necessary.

(4) **tp_lifo** is a decentralized task pool with LIFO access strategy. No lock mechanism is necessary.

(5) **tp_fifost** is a decentralized task pool with FIFO access strategy and task stealing. The stealing mechanism is the following: If the private queue is empty a thread tries to steal tasks by locking and investigating the queues of other threads. The thread visits the queues of the other threads one after another. If the thread is successful in stealing one task, it executes the stolen task and then visits the next queue. If the thread could not steal a task, it immediately visits the next queue. The thread blocks if the thread is not able to steal any task from another queue.

(6) **tp_lifost** is a decentralized task pool with LIFO access strategy and task stealing. The stealing mechanism is the same as for **tp_fifost**.

(7) **tp_fifost2** is a decentralized task pool with FIFO access strategy and task stealing. The stealing mechanism is the following: Task stealing is initiated if the number of tasks in the private queue drops below a predefined threshold value. In contrast to the stealing strategy in **tp_fifost** the stealing process is split into locking and stealing: First a thread tries to lock the queue of another thread. After a thread has successfully locked a foreign queue it can only extract a task if more than a certain predefined number of tasks is available. This avoids the stealing from almost empty queues which would force the owner thread of that queue to steal a task too. The thread trying to steal a task visits the queues of other threads one after another until it can successfully steal a task. Otherwise it continues to execute the remaining tasks in its own queue.

(8) **tp_lifost2** is a decentralized task pool with LIFO access strategy and task stealing. The stealing mechanism is the same as for **tp_fifost2**.

Our task pool implementations provide a user defined number of task pool threads only able to execute tasks. The main thread can perform arbitrary code like creating initial tasks, working on the task pool, and leaving the task pool to prepare data for the reactivation of the pool. The advantage especially for clusters with single processor nodes is that no additional costs emerge from the synchronization of threads on code which was designed to be executed by a single thread. Furthermore the expensive creation of new threads, in case that the task pool is used again, is avoided because all task pool threads wait passively for new tasks.

2.3 Application programmer interface for task pools

For the programming with a task pool the user API provides functions for initializing and destroying the task structure, inserting tasks, and extracting tasks. Those functions are used within an application program and the additional effort for the application programmer is to formulate the functions or procedures as tasks. The following set of access functions is provided:

(1) Allocate and initialize the task pool structure:
void tp_init (**unsigned** number_of_threads) ;
The parameter number_of_threads denotes the total number of threads working at the task pool.

(2) Reinitialize the task pool:
void tp_reinit () ;

(3) Destroy the task pool:
void tp_destroy () ;

(4) Insert initial tasks:
void tp_initial_put (**void** (* taskroutine)(), arg_type *arguments) ;
The parameter taskroutine is a pointer to the function representing a task. The arguments for that function are given by the pointer arguments.

(5) Insert tasks dynamically:
void tp_put (**void** (* taskroutine)(), arg_type *arguments, **unsigned** thread_id) ;
The parameter thread_id has to contain the identifier of the thread inserting the task. The other parameters are identical to the parameters of tp_initial_put.

(6) Extract tasks for processing:
void tp_get (**unsigned** thread_id) ;
The parameter thread_id denotes the identifier of the thread removing a task.

The application programmer explicitly guides the creation of tasks by using those functions. The strategy of the chosen task pool variant is known, but the actual mechanism is hidden to the user so that the programmer can concentrate on the task structure of the program. The task structure, e.g. the granularity of tasks or the dependencies between tasks, might influence the efficiency for some task pool versions.

3 Task pool teams

In this section, we present the combination of several task pools by mutual explicit message passing to build task pool teams for clusters of PCs or SMPs. Mutual communication between cluster nodes is realized with MPI. The combination of such different programming models for implementing irregular algorithms requires some general considerations about the compatibility of both models and the communication structure of applications. At first we investigate the communication behavior of task-pool-based distributed algorithms and the resulting requirements to form task pool teams. The subsequent section discusses some problems of MPI implementations concerning the combination with threads. Then we present our approaches of communication schemes needed in task pool teams. At last some implementation details are given.

3.1 Communication behavior of task-pool-based distributed algorithms

Depending on the specific application there are different situations which may require communication. These communication situations can be divided into two main classes:

- administrational communication emerging from the need to balance load and synchronize cluster nodes and
- application specific communication to exchange input data, (intermediate) results and other data structures for calculation.

Due to the behavior of irregular algorithms the actual communication is hardly predictable in most cases. Moreover, in a task-pool-based implementation the threads of different cluster nodes work asynchronously on different parts of the code so that the counterpart of communication operations, which is needed for the communication in the SPMD style of the MPI programming, might be missing when calling a communication operation. In order to solve those problems we use a separate *communication thread* for each task pool, which is responsible for communication. The communication is then realized using a specific protocol for the interaction of threads executing tasks (called worker threads), the corresponding communication thread, and the worker and communication threads of other task pools in the task pool team.

3.2 Thread-safe communication

A thread-safe design and implementation of the Message Passing Interface guarantees that the threads of a process can call MPI operations simultaneously without mutual interference or influence. Although the MPI standard was designed thread-safe most implementations are not thread-safe. The extension MPI-2 of the MPI standard provides functions to support multithreaded programs. There are different levels of thread safety ranging from only one user thread to multiple user threads which can perform MPI operations simultaneously. Currently, implementations often support only the single-threaded mode. For that reason we protect each MPI operation by a lock. This ensures that MPI calls are only made by one thread at a time, however, the combination of blocking MPI operations and access protection can lead to deadlock. Therefore nonblocking communication operations are used when necessary.

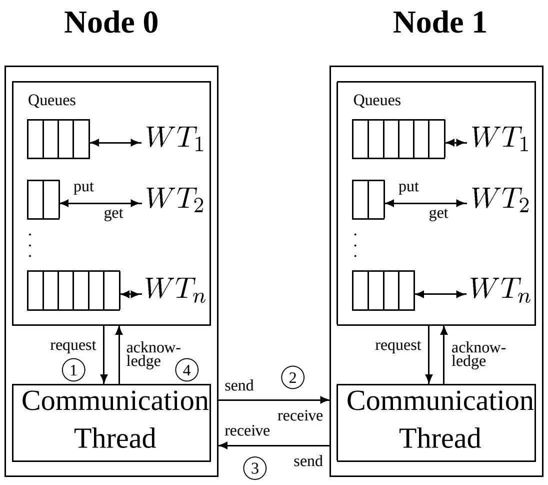


Figure 1: Illustration of the communication scheme for a task pool team with 2 task pools running on a cluster with 2 nodes. Each node employs n worker threads (WT). All worker threads $WT_1, \ldots, WT_n$ use the communication thread for communication with the other node. The communication protocol with the steps (1), (2), (3), and (4) is described in the text.

3.3 Communication scheme with separate communication thread

To integrate a communication thread the basic programming model of task pools for shared memory nodes is modified. The modified task pool of each node consists of a fixed number of $n + 1$ threads: n worker threads responsible for processing tasks, and a separate communication thread. Figure 1 illustrates a task pool team of two decentralized task pools with n worker threads $WT_1, \ldots, WT_n$, their corresponding task queues, and one separate communication thread. The figure also shows the communication scheme. The basic communication protocol for the configuration given in Figure 1 consists of the following steps:

(1) If a worker thread of Node 0 needs data situated in the address space of Node 1 it signals its communication thread and waits passively.

(2) The communication thread of Node 0 sends the request to Node 1 and the corresponding communication thread receives the message.

(3) According to the message type the communication thread of Node 1 reacts and sends data back.

(4) The communication thread of Node 0 receives the data and awakes the requesting thread for continuing its work.

In this approach the communication thread is designed to handle the entire communication needs. Slightly modified schemes are possible and are introduced in the next subsection.

3.4 Modified communication protocol

A modification of the communication protocol is to split the steps of communication between worker threads and communication thread of the same node. Figure 2 illustrates different possible communication schemes.

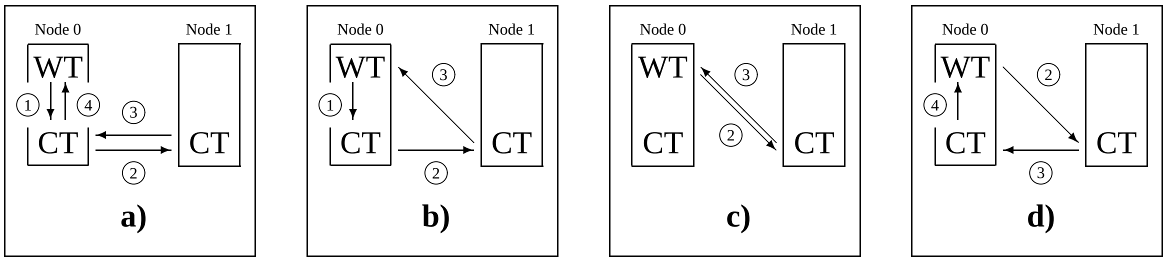


Figure 2: Simplified illustration of possible communication schemes only showing one worker thread (WT) of Node 0 involved in the communication and the communication threads (CT) of Node 0 and Node 1.

The first communication scheme a) is identical with the scheme proposed in Figure 1. The circled numbers characterize the different steps to execute. Figure b) shows a direct receive of requested data by the requesting thread. This forces the worker thread to wait actively for the message. Figure c) reduces the communication to step 2 and 3 and also requires active waiting of the requesting worker thread. Communication scheme d) avoids the additional synchronization of scheme a) and active waiting of schemes b) and c) by introducing a direct request from the worker thread requiring remote data. In communication scheme d), a thread that needs non-local data sends a direct request to the corresponding node and waits passively for reply. The communication thread of the remote node receives the request and sends the data back. The communication thread of the local node receives the data and signals the waiting worker thread that non-local data are available. Experiments have shown that this scheme leads to the best performance.

3.5 Implementation details

Programs for task pool teams are written in an SPMD style and each cluster node runs the same SPMD program. The program for a node has a task-oriented structure and uses a task pool for processing the tasks as described in Section 2. Additionally, the program code contains MPI communication operations. In the following we give detailed information about the communication and synchronization of worker and communication threads according to the described protocol d). During the processing of tasks worker threads can call MPI operations initiating a communication process (Figure 2, step 2). There are two main steps which have to be performed. At first the sending of the request and secondly the waiting for the reply. The following code fragment illustrates the first step:

```
tag = create_tag(message_type, thread_id);

pthread_mutex_lock(communication_lock);
MPI_Isend(buffer, count, type, destination, tag, communicator, request);
pthread_mutex_unlock(communication_lock);
```

Tags are integer values used by the MPI communication operations to identify a message. We use them for the unique identification of worker threads within the communication scheme and for the transmission of the message type. The lower bits of the tag contain the identifier of the thread initializing the communication process. The identification of worker threads is necessary because the communication thread of the same process has to distinguish the incoming data in order to assign them to worker threads. The higher bits of the tag represent the type of the message. The message type indicates the communication thread which action it has to perform. The application programmer has to define a unique type for each specific action.
Due to the missing thread-safety the actual send operation has to be protected by a lock (`pthread_mutex_lock, pthread_mutex_unlock`). `MPI_Isend` is a non-blocking operation for sending the data located in `buffer` of length `count` and type `type` to the process `destination`. (The `communicator` denotes the set of processors able to receive the message and `request` serves for the identification of the operation.) After sending the request the worker thread waits passively for reply:

```
pthread_mutex_lock(wait_lock[thread_id]);
while(!received[thread_id])
        pthread_cond_wait(wait_cond[thread_id], wait_lock[thread_id]);
received[thread_id] = 0;
pthread_mutex_unlock(wait_lock[thread_id]);

/* ... read the requested data out of buffer[thread_id] ...  */
```

The Pthread library provides `pthread_cond_wait` for passive waiting. This function uses the condition `received[thread_id]`. As long as this condition is not true the worker thread blocks. If the requested data are available the worker thread is awaked by the communication thread which has set the condition variable true. That means the worker thread leaves the loop and alters the condition to false for the next use. The entire mechanism has to be protected by a lock to prevent the concurrent modification of the condition variable by the worker and the communication thread. The requested data are stored in the buffer denoted by `buffer[thread_id]`.
Since there are multiple worker threads capable of sending requests the variables `wait_lock, wait_cond, received` as well as the buffer `buffer` for the requested data have to be available for each worker thread exclusively. Therefore we chose arrays for each variable indexed by the worker thread identifier `thread_id`.

The communication thread receives incoming messages (Figure 2, steps 2 and 3) and performs the corresponding actions. It works on a function with a predetermined structure. The following code fragment gives an overview of that function:

```
int receive = 0;
MPI_Status status;

while(1) {
   pthread_mutex_lock(communication_lock);
   MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
              &receive, &status);
   pthread_mutex_unlock(communication_lock);

   if(receive != 0) {
       thread_id = extract_id(status.MPI_TAG);
       message_type = extract_type(status.MPI_TAG);
       switch (message_type) {
             case TYPE_1: /* ... */
             case TYPE_2: /* ... */
             /* ... */
             case TYPE_n: /* ... */
       }
   }
}
```

`MPI_Iprobe` checks for incoming messages of any origin (`MPI_ANY_SOURCE`) with any tag (`MPI_ANY_TAG`) within a set of all processes (`MPI_COMM_WORLD`). The communication thread has to distinguish between two classes of messages. On the one hand messages which request data (Figure 2, step 2) and on the other hand messages which contain requested data (Figure 2, step 3). The first class requires the sending of the requested data back to the origin of the received message. The second one requires the completion of the communication process:

```
pthread_mutex_lock(communication_lock);
MPI_Recv(buffer[thread_id], count, type, status.MPI_SOURCE,
         status.MPI_TAG, MPI_COMM_WORLD, &status);
pthread_mutex_unlock(communication_lock);

pthread_mutex_lock(wait_lock[thread_id]);
received[thread_id] = 1;
pthread_mutex_unlock(wait_lock[thread_id]);
pthread_cond_signal(wait_cond[thread_id]);
```

At first the message is received in the buffer `buffer` associated with the worker thread identifier `thread_id`. The variable `status` contains necessary information like the origin and the tag of the message to receive. After that the condition is set true and the worker thread denoted by `thread_id` is awaked by the function `pthread_cond_signal` (Figure 2, step 4).

4 The traveling salesman problem

We chose the traveling salesman problem (TSP) as a less complex irregular example in order to illustrate the conversion of a sequential program into our hybrid programming model. We have concentrated on the conversion principle rather than creating the most efficient implementation.

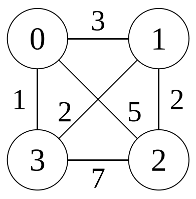
The TSP is a minimization problem which finds the shortest circular path in a weighted, undirected, and connected graph $G = (V, E)$ with a set of edges E and a set of nodes V. A graph is:

undirected if $e_1 = e_2$ for each $e_1, e_2 \in E$ with $e_1 = (i, j)$, $e_2 = (j, i)$ and $i, j \in V$.

connected if the edge $e = (v_1, v_2)$ exists for each node $v_1, v_2 \in V$, $v_1 \neq v_2$.

weighted if a cost function $c : E \rightarrow I\!N$ exists associating each $e \in E$ with a natural number.

A circular path is a set of edges which links all nodes of G in a circle containing each node only once. The cost of a circular path is the sum of edge weights. The circular path with minimal cost is the shortest circular path. Figure 3 illustrates the TSP for a graph with four nodes.



$$V = \{0,1,2,3\}$$
$$E = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$$
$$cost\,matrix = \begin{pmatrix} \infty & 3 & 5 & 1 \\ 3 & \infty & 2 & 2 \\ 5 & 2 & \infty & 7 \\ 1 & 2 & 7 & \infty \end{pmatrix}$$
$$possible\,circular\,paths = \{(0,1,2,3,0),(0,2,3,1,0),(0,2,1,3,0)\}$$
$$shortest\,circular\,path = (0,2,1,3,0)$$

Figure 3: The traveling salesman problem for 4 nodes

The TSP is NP-complete. That means most likely there is no algorithm to resolve the problem within polynomial time. The simplest way to determine a solution is enumeration. A graph with n nodes has $(n-1)!/2$ possible paths to test. Due to this rapidly growing number of possibilities with an increasing number of nodes only small problems can be solved with this method in adequate time.

4.1 Sequential implementation

We have implemented a backtracking algorithm (based on the enumeration principle) which serves as starting point for the development of the distributed task-oriented variant. Backtracking algorithms regard a treelike solution space and try to find the solution stepwise. Each incorrect step will be cancelled.
Figure 4 shows the solution space of the TSP for a graph with three nodes. Our algorithm searches the tree by depth search and stores local minima. In order to reduce the number of tree nodes to visit, the algorithm cuts the branches which contain edges connecting a node of the graph with itself. Furthermore all paths with a cost exceeding the current minimum or containing multiple circles are rejected. The following C-code fragment illustrates the functionality of the sequential algorithm:

```
int n;     /* number of graph nodes */
int path[n], cost, node;
int res_minimum = MAXINT, res_vector[n];   /* solution */

search(int *path, int cost, int node) {

   determine_path(path, node);
   if(valid_path(path)) {
        determine_cost(path, cost);
        if(cost < res_minimum) {
             if(check_for_circular_path(path))
                  copy_to_result(path, cost);
             else
                  for(node=0; node<n; node++)
                       search(path, cost, node);
        }
   }
}
```

Beginning from the root of the tree spanning of the solution space the function `search` is performed recursively. It determines the path (`determine_path`) and the cost (`determine_cost`) up to the current node. If this path is valid, that means each visited graph node occurs only once (valid_path), and the current cost is less than the current minimum we can check for a circular path. The function check_for_circular_path performs that by testing if each graph node occurs only once and in a single circle. A positive test indicates that a new minimal circular path has been found. A negative test forces the resumption of the search by investigating the children (if existing) of the current tree node.

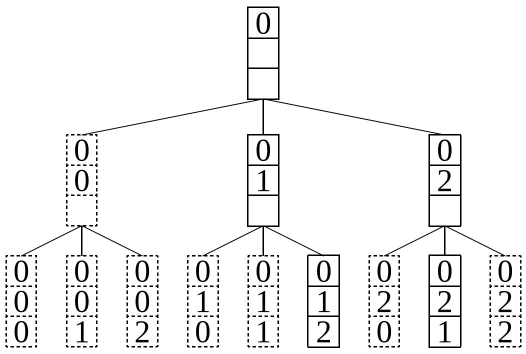


Figure 4: Solution space for a graph with 3 nodes. The dashed boxes mark rejected paths. There are two identical solutions: (0,1,2,0) and (0,2,1,0).

4.2 Parallel implementation

We have implemented a parallel version for distributed memory by allocating subtrees to cluster nodes. The function search is used without changes. The exchange of final results is realised with MPI operations. The runtime of this implementation is determined by two main factors: on the one hand the missing load balance and on the other hand the missing exchange of local minima between cluster nodes during the computation. Both problems can be solved with our hybrid programming model. We decided to formulate the function search as task and change the name to search_task. Thus the resulting code looks like as follows:

```
int n;
int res_minimum = MAXINT, res_vector[n];

typedef struct argument {
   int path[n];
   int cost;
   int node;
} arg_t;

search_task(arg_t *arg_old) {

arg_t *arg = (arg_t *)malloc(sizeof(arg_t);

   arg->path = determine_path(arg_old->path, arg_old->node);
   if(valid_path(arg->path)) {
        arg->cost = determine_cost(arg->path, arg_old->cost);
        if(arg->cost < res_minimum) {
             if(check_for_circular_path(arg->path)) {
                  lock;
                  copy_to_result(arg->path, arg->cost);
                  send_to_all(arg->cost);
                  unlock;
```

```
                }
                else
                     for(arg->node=0; arg->node<n; arg->node++)
                         tp_put(search_task, arg);
           }
      }
}
```

The general structure of the function is preserved. The recursive part is substituted by a task-oriented structure. That means the original parameters for `search` like `cost`, `path`, and `node` are wrapped in the structure `argument` and the memory for that structure has to be allocated (`malloc`) for each function call of `search_task`. `tp_put` inserts the new task (the function and its arguments) in the task pool. Because there are no data requests the communication scheme is simplified. The function `send_to_all` performs the sending of locally calculated minima to all other cluster nodes. The communication thread receives the data and makes them available for the worker threads. The process of copying calculated results (`copy_to_result`) to the solution variables has to be protected by a lock because several threads can perform this process concurrently.

The granularity of tasks has main influence on runtime. Frequent inserting and removing of tasks caused by small tasks can increase runtime. For this TSP example it is beneficial to take branches or parts of branches of the search tree to form one task. The parallel execution combined with the exchange of locally calculated minima reduces the number of branches to search enormously (compared with the sequential implementation). For that reason the speedup values are extremely large and we do not present them.

5 The hierarchical radiosity algorithm

The hierarchical radiosity algorithm (HRA) represents a complex irregular algorithm requiring dynamic load balancing for achieving an efficient parallelization and thus is suitable for testing task pool teams. The next subsections give a short introduction to the HRA and provide information about application-specific adaptations to reduce communication.

5.1 The hierarchical radiosity algorithm

The basic 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 patches or elements for which radiosity values[1] are computed. The radiation power of the overall system is modeled by the equation

$$\underbrace{B_i A_i}_{1} = \underbrace{E_i A_i}_{2} + \underbrace{\rho_i \underbrace{\sum_{j=1}^{n} \underbrace{F_{ji}}_{3a} \underbrace{B_j A_j}_{3b}}_{3}}_{4}, \qquad i = 1, ..., n \tag{1}$$

[1]radiosity = radiation per time and surface unit.

1	radiation power of surface element i	3	incident radiation power from other elements
2	emission of surface element i	4	diffuse reflection of radiation power incident from other elements
3a	form factor		
3b	radiation power of surface element j		

In this system B_i is the radiosity value for element i, A_i is the surface area of element i, and E_i is the emission per time and surface unit of element i, $i = 1, ..., n$. The parameter n is the number of surface elements and determines the refinement level of the division of the surface polygons. A large n causes a high quality of the scene and increases the computational effort considerably. The form factor F_{ji} describes the portion of light energy incident on an element from another 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. Equation 1 has to be solved with a direct or an iterative method.

In order to decrease the costs the number of form factors is reduced with a hierarchical approach. The computation of form factors is based on the basic law for the transmission of radiation. That means the energy exchanged between elements is decreasing quadratically with the distance of these elements. This fact makes it possible to perform less exact computations for remote surface elements while getting good realistic results. The hierarchical approach results in an uneven division of input polygons into patches or elements as illustrated in Figure 5. After finishing the algorithm the entire scene is represented by a set of quadtrees with one tree for each input polygon. The leaf elements of all trees represent the surfaces to display but all levels of the quadtrees are required for computation.

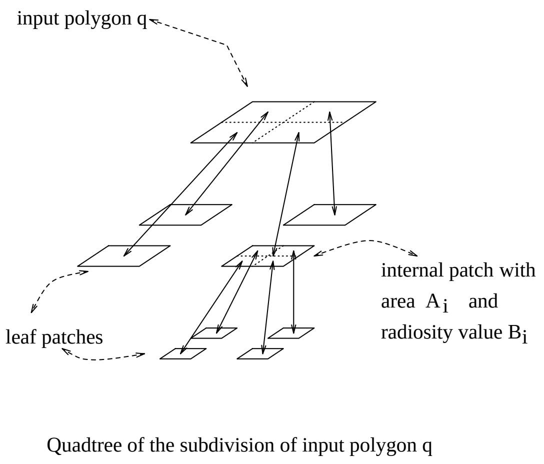

input polygon q
internal patch with area A_i and radiosity value B_i
leaf patches
Quadtree of the subdivision of input polygon q

Figure 5: Uneven subdivision of input polygons and representation as quadtree

The subdivision of a surface element depends on its size and on the portion of energy incident from other elements. The refinement process stops if a minimal size is reached or the form factors for two interacting elements are small enough to create a realistic scene. Due to the fact that distant elements exchange less energy than elements situated close to each other the interaction of those elements takes place on higher levels of the tree. Neighboring elements interact at leaf level. To store the diverse interaction partners each patch or element represented by a quadtree node owns an interaction list which contains pointers to those elements providing portions of light energy. The interaction lists are changed dynamically due to the dynamic refinement of patches depending on the specific geometry and energy situation. Figure 6 illustrates the two possible cases of refinement: If the area of element p is larger than the area of element q, then p is subdivided, if not already done, into four smaller elements with new interaction lists containing pointers to element

q. The old interaction to q is invalid. The right side of Figure 6 shows the second case where the old interaction to q has to be deleted and four new interactions are inserted in the interaction list of p.

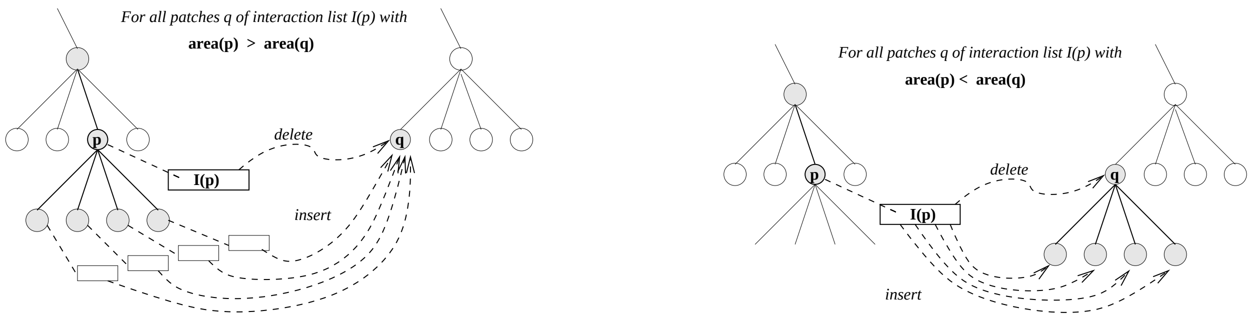

For all patches q of interaction list I(p) with
area(p) > area(q)
p
q
I(p)
delete
insert
For all patches q of interaction list I(p) with
area(p) < area(q)
p
q
I(p)
delete
insert

Figure 6: Subdivision of surface elements and required changes in the interaction list I(p).

5.2 The radiosity implementation of the SPLASH-2 benchmark suite

The SPLASH-2 benchmark suite contains an implementation of the hierarchical radiosity algorithm with task pools for shared memory. The HRA computes the radiosity values by an iterative method for solving the system of equations with top-down and bottom-up passes over the quadtrees accessing data according to the interaction lists. We started with the pure thread-based program version modified by [12] and further modified for Pthreads within the scope of [11]. Further adaptations were necessary for the use of our specific task pools and for the integration of the entire communication structure for the task pool teams. These adaptations are introduced in subsection 5.3.

The HRA creates tasks of five different types which are executed in three phases. The first phase performs initializations like the building of a BSP tree for visibility computations and the creation of quadtree roots and their interactions. The last phase deals with the smoothing of the scene to improve the quality. The most complex phase is the second one in which the system of equations is solved iteratively by the execution of ray and visibility tasks. During the processing of the ray tasks for each quadtree root more tasks are recursively created for all child nodes until the leaf levels of the quadtrees are reached. Then the radiosity values are recalculated for inner nodes and the root nodes of the quadtrees in a bottom-up pass over the tree.

In particular the following steps are performed within a ray task. At first the refinement of elements is done by a pass over the interacting elements. If elements of the interaction list are refined new interactions are inserted in the list. That means the visibility factors are not yet calculated for those new elements. For that reason a ray task creates a visibility task. A visibility task computes the missing visibility factors and afterwards it performs the same actions as a ray task. This process is recursive and stops when no refinements take place anymore within a task. If all visibility factors are calculated the radiation can be gathered and passed on to the children. If the leaf level is reached the bottom up pass of the tree starts.

5.3 Application-specific optimizations

We have also modified the HRA implementation of the SPLASH-2 benchmark suite for the usage of task pool teams. In order to make the algorithm more efficient several optimizations and techniques have been examined:

1. dummy data structures for remote data (software cache),
2. initial distribution of data,
3. detecting redundant communication,
4. combining requests.

Although the improvements are application-specific, the basic ideas can be applied to other irregular algorithms.

5.3.1 Dummy data structures for remote data

In the distributed algorithm the patches or elements and the corresponding quadtrees are distributed over the different address spaces of the cluster nodes. Accordingly, the information in the interaction lists might point to remote data. Thus, the computation of radiosity values can cause irregular communication since radiosity values of other elements stored in non-local address spaces have to be made available by message passing. Because non-local data access with message-passing takes usually much more time than local memory access and in most cases the same element is needed again in subsequent calculations we use a balanced binary tree to store non-local information which allows easy and fast access by any thread of the process. A node of the tree consists of a data structure with the type `dummy_tree_type`.

```
typedef struct dummy_tree {
        long global_pointer;
        Element *local_pointer;
        short balance;
        struct dummy_tree *left, * right;
} dummy_tree_type;

dummy_tree_type *search_dummy(long global_pointer, int rank);
void insert_dummy(long global_pointer, int rank);
```

Each node contains information necessary to manage the tree, like pointers to the left and the right son and a balance value for reorganization and information of the element represented by the tree node like `global_pointer`, which denotes the actual memory address of the element, and `local_pointer`, which points to the data for this element at the local address space.
Because of the distributed address space of non-local processes memory addresses are not unique. So a separate tree exists for each remote process. The function `insert_dummy` allows the insertion of dummies in a tree identified by `rank`. `global_pointer` serves as key to determine the correct position in the tree. The function `search_dummy` returns a pointer to a dummy element identified by the key `global_pointer`. The correctness of data is guaranteed by refreshing data in each step of the iterative solution method. Our experiments have shown that this software cache approach leads to efficiency gains.

5.3.2 Initial distribution of data

Although the access behavior for irregular algorithms is hardly predictable, in some cases useful information can be extracted from input data. For the HRA it is possible to group the input polygons according to its mutual visibility. If two polygons are mutually invisible there is no exchange of energy between them and an assignment to different nodes is advantageous. We have investigated the runtime of the HRA with several simple distribution strategies which actually lead to lower communication. Unfortunately, this reduction of communication was associated with a heavy imbalance of computational work. Thus the best runtime results are achieved with a regular cyclic assignment of initial polygons to the physical processors, although the number of communications was not minimized.

5.3.3 Detecting redundant communication

Each worker thread computes radiosity values for elements using data from interacting local and non-local elements. In most cases non-local data are available in the dummy data structures. However, the refinement of a non-local element caused by ray tasks represents an exceptional case because the refined elements are only present in the memory of the owner process. If two threads of the same process initiated the refinement process for the same non-local element concurrently, redundant communication would occur. We have implemented a mechanism which initiates a communication of only one thread and blocks any other thread requiring the data until the specific data are available. The underlying structure is a small array for each task pool containing all currently requested elements. This mechanism requires the extension of the dummy data structure by four pointers to the child dummies of that element. If a non-local element is not refined (or the data are not yet available) the pointers are NIL. The following code fragment shows the refinement mechanism for remote elements. To simplify the code we have omitted the lock and unlock functions.

```
/* ... */
if(register_element(dummy->global_pointer, array[rank])) {
    while(dummy->child_1 == NULL)
        pthread_cond_wait(cond[rank], lock[rank]);
    return;
}
else {
    /* ... send the request and wait passively ... */
    /* update the pointers dummy->child_1 .. dummy->child_4 */

    remove_element(dummy->global_pointer, array[rank]);
    pthread_cond_broadcast(cond[rank]);
}
return;
```

Before initiating a communication each thread checks the array denoted by `array[rank]` for the needed element with the function `register_element`. If the element is already registered (`register_element` returns 1) the thread blocks. Otherwise it adds the needed element to the array and starts the communication process by a request. The thread actually sending the request deletes the element from the array with the function `remove_element` and awakes all waiting threads (`pthread_cond_broadcast`) after the requested data are available. There are at most as many entries as worker threads per task pool and so the search for elements in those arrays is cheap, especially when compared with communication.

5.3.4 Combining requests

In order to reduce communication, requests can be collected and send as a single message. The HRA offers two obvious situations for the combination of requests:

a.) The preliminary radiosity values for elements computed at one node within one iteration step are immediately accessible for local threads. New values of non-local elements are exchanged between nodes at the end of an iteration step rather than immediately. This algorithmic change results in slight changes of the convergence rate.
b.) During the computation of the radiosity values of an element each interacting element is investigated for locality. If there are non-local interacting elements they are accessed separately. The access to interacting non-local elements situated in the address space of the same process can be combined because all interacting non-local elements are already known at the beginning of an iteration step.

6 Experimental results

This section presents some of our experimental results with the hierarchical radiosity algorithm. As example scenes we use a.) ”largeroom” (532 initial polygons), b.) ”hall” (1157 initial polygons), and c.) ”xlroom” (2979 initial polygons). The models ”hall” and ”xlroom” were generated by [12]. ”Largeroom” belongs to the SPLASH-2 benchmark suite [20]. We have investigated the task pool teams on two architectures: the Chemnitzer Linux Cluster (CLiC), a Beowulf cluster consisting of 528 Intel PentiumIII processors with 800 MHz and running Linux. CLiC has a Fast Ethernet network. SB1000 is a small 4x2 SMP cluster of SunBlade 1000 with 750 MHz UltraSPARC3 processors. The operating system is Solaris. SB1000 uses SCI.

Figures 7 and 8 show the results of a program run of the HRA on two processors. The final illuminated scenes illustrate the distribution of work between different cluster nodes and corresponding task pools.

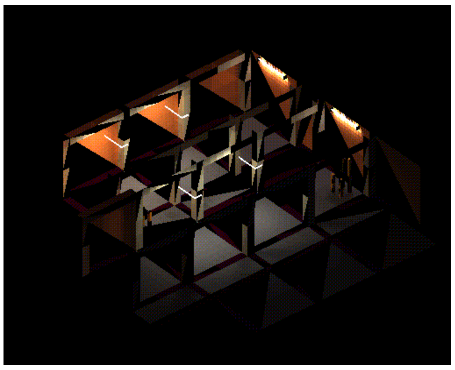
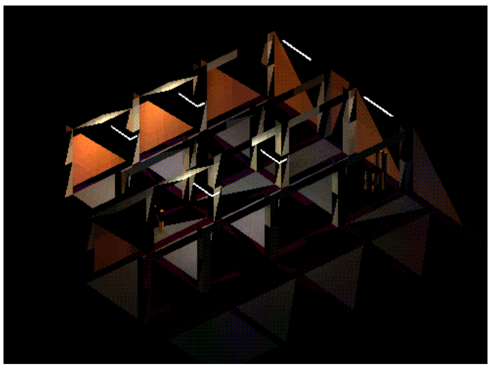



Figure 7: Results of the distributed computation of the HRA for the scene ”hall” on two nodes

Figure 9 gives runtimes depending on the number of threads and the number of processors (of CLiC and SB1000) of the HRA using the scene ”xlroom”. The figure shows that there are dependences of the runtime on the number of processors as well as on the number of threads. Therefore we present all runtime and speedup measurements with regard to threads and processors. The speedup values have been computed using the implementation without communication thread and with only one worker thread.

Although CLiC has only one CPU per cluster node the usage of only two threads per node (one worker thread and one communication thread) is not reasonable since an increasing number of

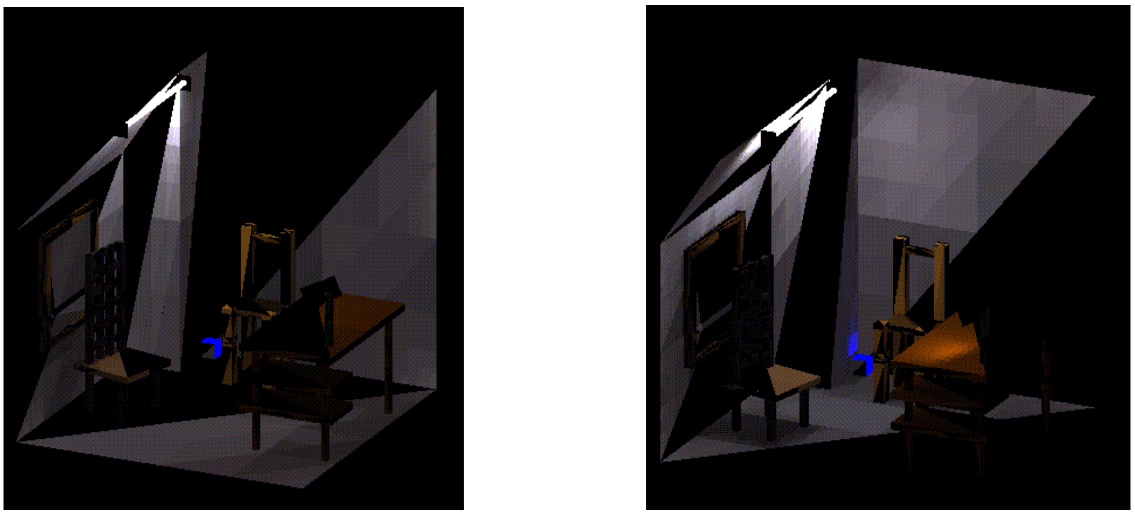


Figure 8: Results of the distributed computation of the HRA for the scene ”xlroom” on two nodes

threads leads to better speedups as the Figures 10 and 11 show. This fact is caused by the competition between threads for processing time. For a large number of threads per process the allocated time for the communication thread (which mostly performs unsuccessful checks for incoming messages) is reduced. Also, the wasted time due to blocking worker threads is minimized. Our experiments have shown that the efficiency converges for an increasing number of threads. If the number of worker threads is too high a decreasing of speedup values is possible, because incoming messages cannot be received fast enough. Figure 12 shows the speedup values depending on the number of threads for the smallest model ”largeroom” on CLiC. Especially for three and four processors the speedup results are less regular than for the models ”hall” and ”xlroom” (see Figures 10, 11). This is caused by the small number of initial polygons for each cluster node compared with the irregular communication requirements.

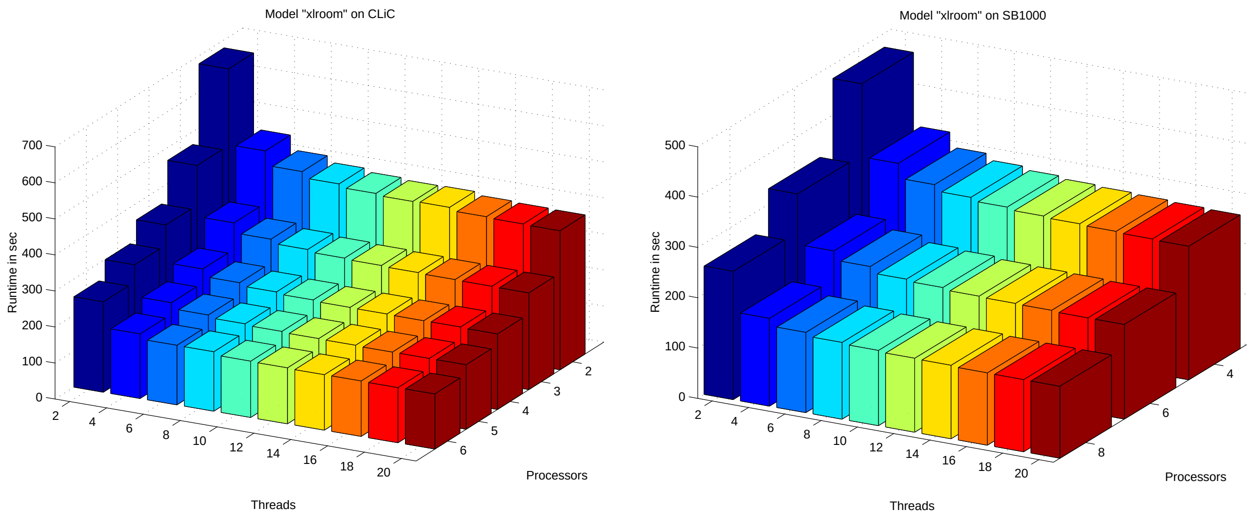


Figure 9: Runtime for model ”xlroom” on CLiC and on SB1000

Because all threads have to compete for two processors of each cluster node the dependence between speedup and number of threads on the SB1000 is similar to the results of CLiC. Figures 13, 14 and 15 illustrate this fact for different task pool versions and for different numbers of processors simulating the models ”xlroom”, ”hall”, and ”largeroom”.

The task pool teams computing the model ”largeroom” achieve different runtimes when using different task pool variants. In most cases computing the model ”largeroom”, the central task pool or the distributed task pool with the second stealing mechanism st2 (task stealing with threshold values) has better speedups than the other distributed task pools. This fact results from a better load balance between the threads of one cluster node. The distributed task pools tp_fifo and tp_lifo (without task stealing) do not balance load very well and the stealing mechanism st (stealing without threshold) wastes too much time when trying to steal from almost empty queues. For the larger models ”hall” and ”xlroom” the runtimes for different task pools differ only slightly. It seems that load imbalances within one cluster node hardly affect the total runtime because of the long over-all runtime compared with the idle time caused by load imbalances.

The effect of the task pool versions is illustrated again in Figures 16, 17 and 18 for a fixed number of 20 threads per task pool. For the smallest model ”largeroom” the best speedups can be achieved with the st2 task pools. The speedup values are 1.7 (2), 2.5 (3), 2.7 (4), 3.4 (5), 3.7 (6) for CLiC and 3.7 (4), 5.4 (6), 5.9 (8) for SB1000. (The numbers in brackets are the number of physical processors.) For ”hall” the best speedups can be achieved on both platforms with the task pool tp_fifost2: 1.7 (2), 2.5 (3), 3.2 (4), 3.7 (5), 4.4 (6) for CLiC and 4.0 (4), 5.8 (6), 7.4 (8) for SB1000. The central task pool tp_fifocen is best-suited for the largest model ”xlroom”. The speedup values are 1.7 (2), 2.5 (3), 3.2 (4), 3.7 (5), 4.4 (6) for CLiC and 3.8 (4), 5.4 (6), 7.1 (8) for SB1000. Due to the large number of tasks in proportion to the communication the speedup values of ”hall” and ”xlroom” are better than the values for the small model ”largeroom”. Similarly an increased number of processors changes the proportion in favor of communication.

7 Related work

There are approaches which deal with the efficient parallelization of irregular algorithms for distributed memory focusing on runtime optimizations and especially on communication behavior. The PARTI [18] and the CHAOS library [15] are developed to support the implementation of irregular problems by runtime procedures on distributed memory. Optimizations are carried out especially for the runtime reduction of nested loops over distributed arrays with the inspector/executor model. This model determines the necessary array elements in a preprocessing step before the actual loop computation and provides them by predefined communication operations. This approach requires compiler support and does not realize dynamic load balance.

Especially for problems with irregular grid-based data structures load balance can be achieved by partitioning into blocks. The aim is to obtain blocks of approximately identical size with minimal interdependence. This problem known as the graph partitioning problem is NP-complete. There are many algorithms to find solutions for that problem, which differ in quality of the produced partitions and the consumed time. The MeTis system [10] implements partitioning algorithms for sparse matrix ordering and partitioning of unstructured graphs. The CHACO [9] software realizes several partitioning algorithms. The usage of partitioning algorithms is not useful if the time for the calculation of partitions and for repartitioning exceeds the time savings gained. Especially for algorithms like the HRA this approach seems to be unsuitable, because there is no static information about the development of the data structures.

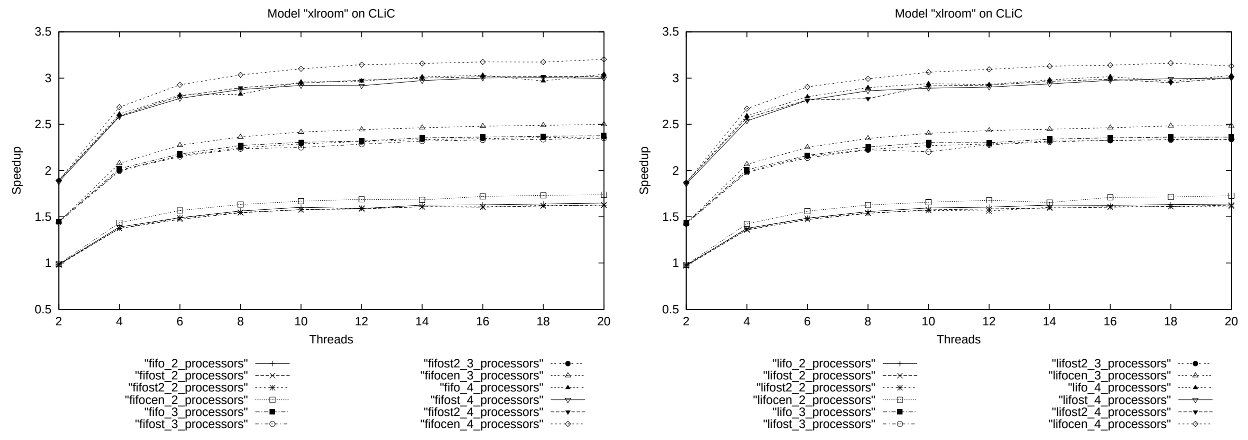


Figure 10: Speedups for model "xlroom" and task pools with FIFO access (left) and LIFO access (right) on CLiC

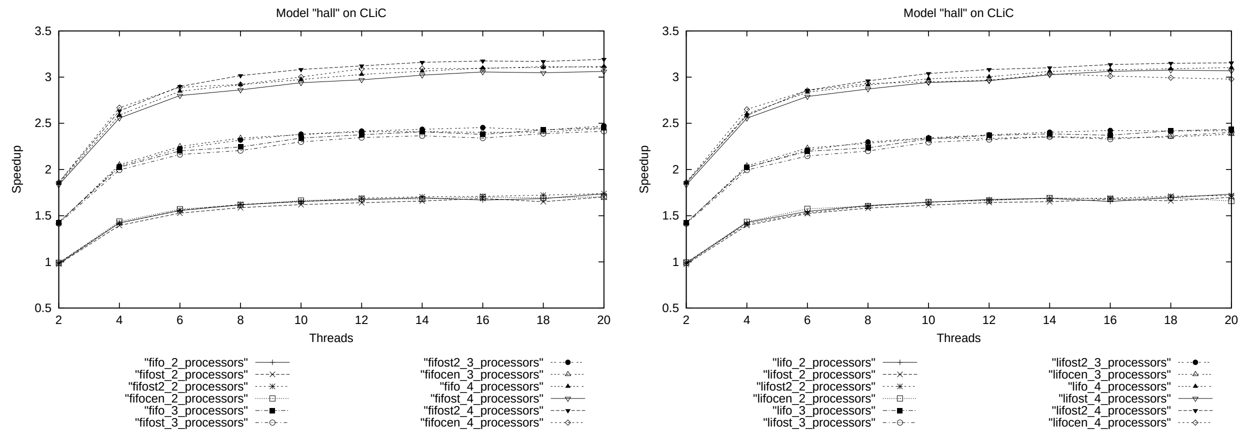


Figure 11: Speedups for model "hall" and task pools with FIFO access (left) and LIFO access (right) on CLiC

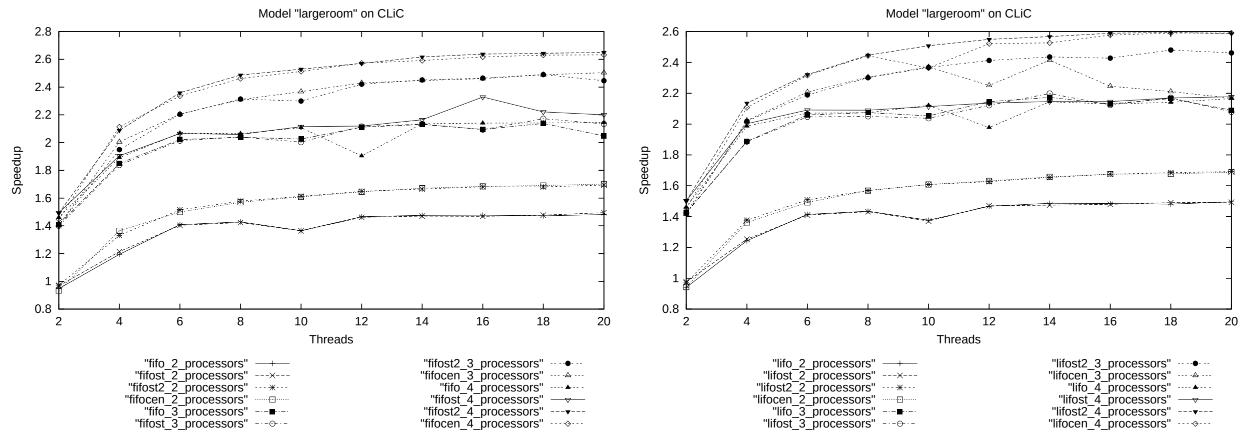


Figure 12: Speedups for model "largeroom" and task pools with FIFO access (left) and LIFO access (right) on CLiC

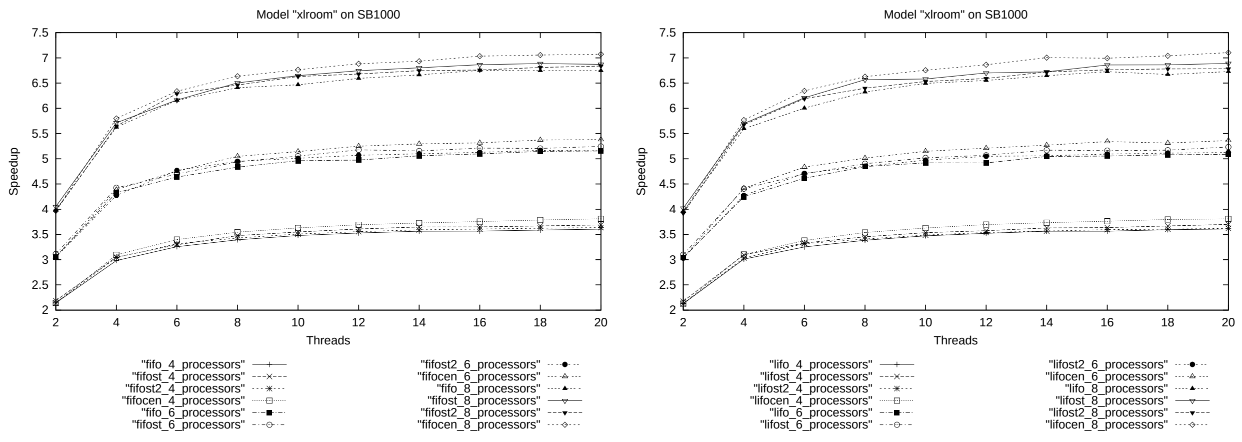


Figure 13: Speedups for model "xlroom" and task pools with FIFO access (left) and LIFO access (right) on SB1000

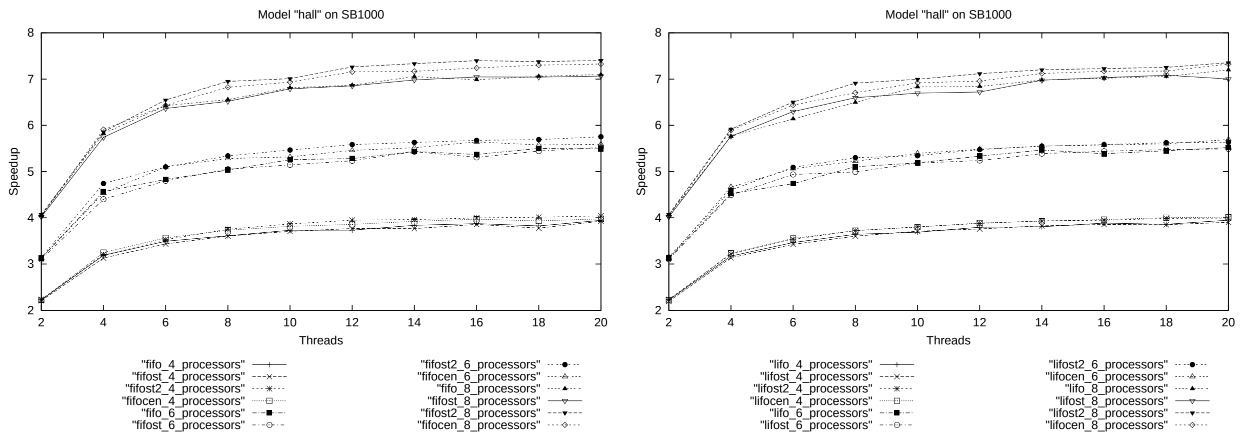


Figure 14: Speedups for model "hall" and task pools with FIFO access (left) and LIFO access (right) on SB1000

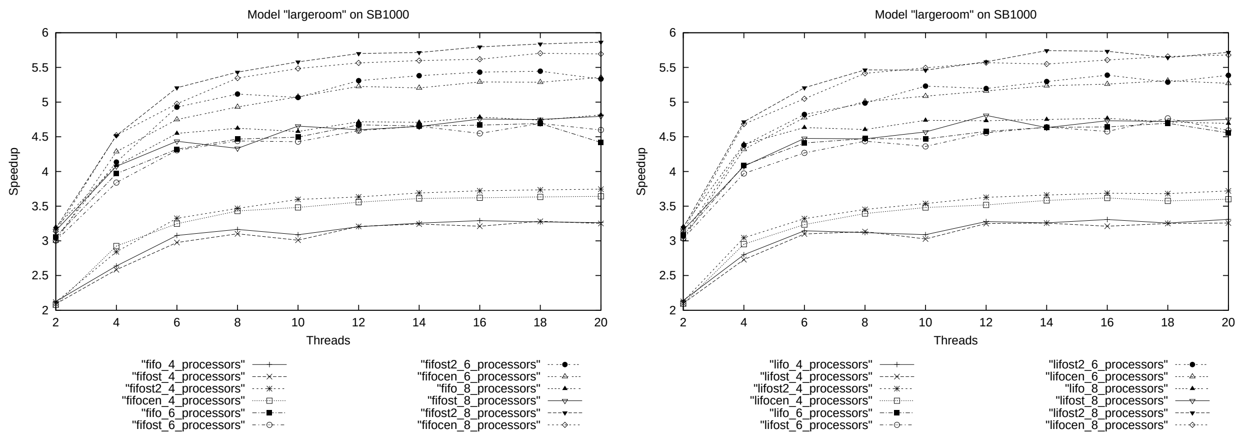


Figure 15: Speedups for model "largeroom" and task pools with FIFO access (left) and LIFO access (right) on SB1000

Several systems provide implicit support for irregular applications, for example: Titanium [21], PETSc [3] and TreadMarks [1]. Titanium includes a Java-based language and compiler for SPMD programs. TreadMarks is a programming environment optimized for irregular data accesses. Both systems map a global address space to distributed memory. PETSc provides data types for sparse and dense structures and uses the inspector/executor model to decrease time for communication. These systems hide the mixed memory organization from the programmer. They were not designed to create dynamic load balance.

There are several approaches for hybrid programming of clusters. Most of them try to benefit from the special architecture of SMP clusters by overlapping communication and computation. They can only be used for applications which allow a strict separation of computation and communication phases. [2] introduces SIMPLE which provides user level primitives for programming SMP clusters in addition to MPI operations and POSIX threads. SIMPLE allows MPI for communication within the application and Pthreads for task and data parallelism. Threads can only be used for data without dependence. Therefore the usage for irregular algorithms is restricted. NICAM [19] is a user level communication layer for SMP clusters which allows overlapping of communication and computation in iterative data parallel applications by using threads and message passing.

An application specific example which uses threads and remote memory operations on an SMP cluster is presented in [14]. This approach introduces the parallelization of sparse Cholesky factorization using threads for the parallel computation of blocks of the factor matrix and remote memory operations for synchronization of remote data access.

There are some packages providing threads on distributed memory. In contrast, our approach is entirely situated within the application programmer level in order to provide a systematic programming approach to the programmer without hiding important details and implicit load balance. Nexus is a runtime environment for irregular, heterogeneous, and task-parallel applications [5, 6]. The focus is more on the realization of the combination on lower levels. Chant [7, 8] presents threads capable of direct communication on distributed memory. This library uses lightweight thread libraries and communication libraries available at the system used.

8 Conclusion

The parallel implementation of irregular algorithms on clusters of PCs or SMPs requires special dynamic load balancing and organization of irregular accesses. For this purpose we have presented task pool teams, a generalized task pool approach solving both problems. Task pool teams combine multithreaded programming with explicit communication on application programmer level. The efficiency results are good on both platforms but depend on the platform and the specific input data. We have chosen the hierarchical radiosity algorithm as test program since this application program is a very complex program combining many characteristics of irregular algorithms. The experiments on CLiC and SB1000 have shown that the approach is suitable for efficient parallelization.

The task pool team approach is entirely embedded in the application programmers level so that characteristics of the application program can be exploited. The user interface is an easy-to-use task pool API for SMP nodes and standard MPI SPMD programming with remote accesses. But still the application programmer has the view on the physically hybrid system and can influence the behavior of the task pool team by picking different task pool versions or starting different numbers of threads per pool.

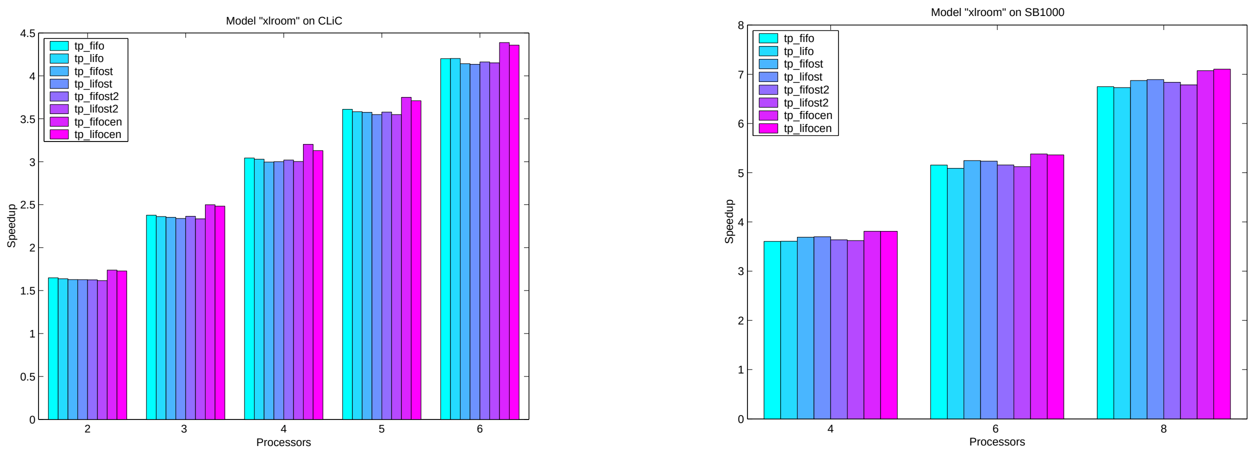


Figure 16: Speedups with 20 threads for model ”xlroom” on CLiC and on SB1000

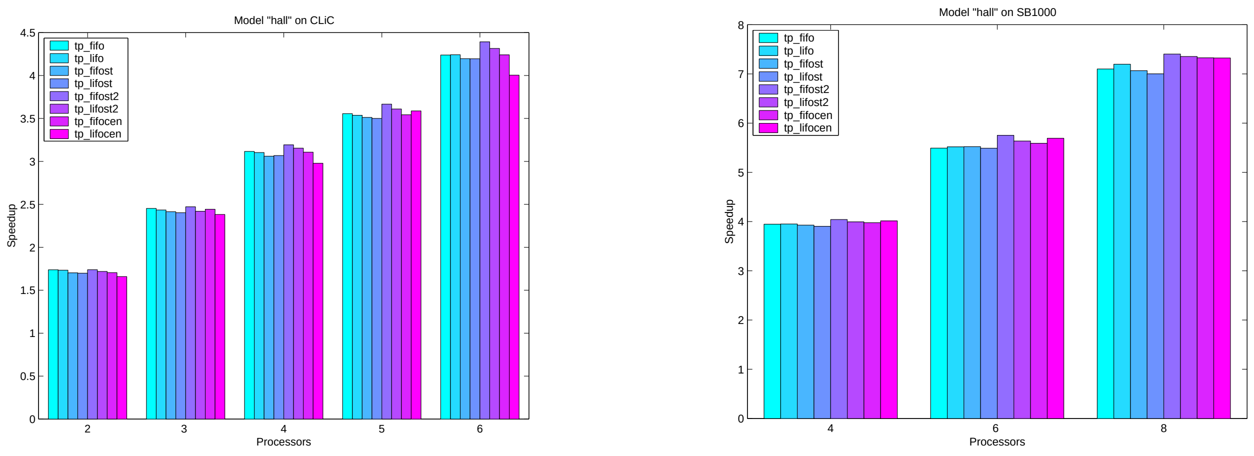


Figure 17: Speedups with 20 threads for model ”hall” on CLiC and on SB1000

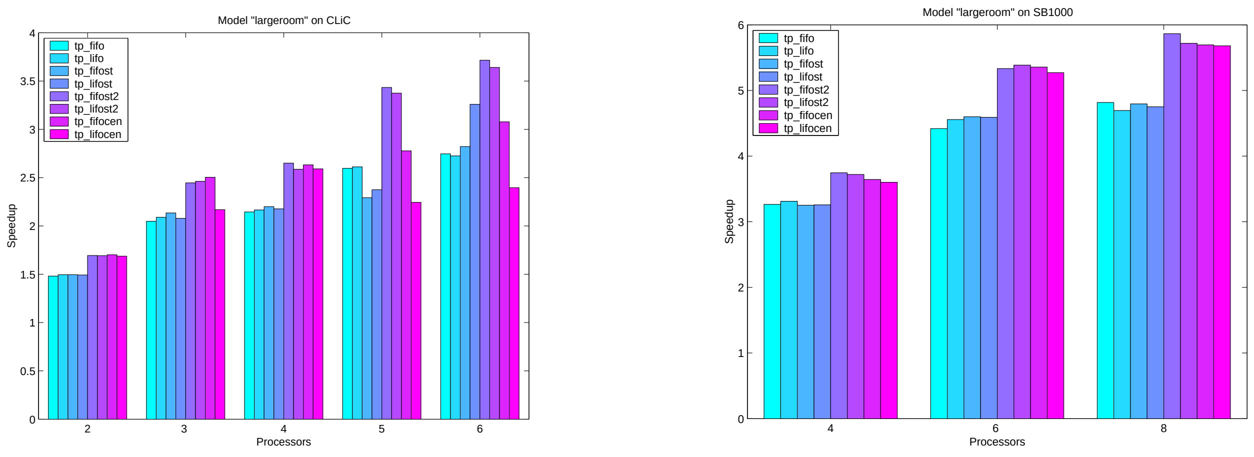


Figure 18: Speedups with 20 threads for model ”largeroom” on CLiC and on SB1000

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] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and eds. H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[4] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[5] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Task-Parallel Runtime System. In *Proc. 1st Intl. Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

[6] 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.

[7] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing '94*, pages 350–359, 1994.

[8] M. Haines, P. Mehrotra, and D. Cronk. *Chant: Lightweight Threads in a Distributed Memory Environment*. Technical report, ICASE, 1995.

[9] B. Hendrickson and R. Leland. *The Chaco User's Guide, Version 2.0*. Sandia National Laboratories, Technical Report SAND94-2692, 1994.

[10] G. Karypis and V. Kumar. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Departement of Computer Science, University of Minnesota, Technical Report, 1995.

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

[12] 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.

[13] T. Rauber and G. Rünger. *Parallele und Verteilte Programmierung*. Springer, 2000.

[14] S. Satoh, K. Kusano, Y. Tanaka, M. Matsuda, and M. Sato. Parallelization of Sparse Cholesky Factorization on an SMP Cluster. In *Proc. HPCN Europe 1999, LNCS 1593*, pages 211–220, 1999.

[15] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. H. Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proceedings of Supercomputing '94*, pages 97–106, 1994.

[16] J. P. Singh. Parallel Hierarchical N-Body Methods and their Implication for Multiprocessors, Ph.D. Thesis, Stanford University, 1993.

[17] 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.

[18] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI Primitives for Unstructured and Block Structured Problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992.

[19] Y. Tanaka. Performance Improvement by Overlapping Computation and Communication on SMP Clusters. *In Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1:275–282, 1998.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[21] 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. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.