Exploiting Task-Level Parallelism Using pTask

Tarek S. Abdelrahman and Sum Huynh
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 3G4

Abstract

This paper presents pTask— a system that allows users to automatically exploit dynamic task-level parallelism in sequential array-based C programs. The system employs compiler analysis to extract data usage information from the program, then uses this information at run-time to dynamically exploit concurrency and to enforce data-dependences. Experimental results using a prototype of the system show scaling performance and low overhead on a 32-processor KSR1 multiprocessor for a number of applications.

1 Introduction

Compiling sequential programs into parallel programs has been an active area of research for the past few years. Much of this research has focused on loop-level parallelism and on the efficient execution of data-parallel programs. An alternative model of parallelism is that of a collection of asynchronous cooperating tasks, or *task-level* parallelism. Such a model is attractive for a number of reasons. Some parallel applications are best expressed as a collection of coarse-grain tasks [12]. Large-scale and multidisciplinary parallel applications are likely to exploit both loop-level and task-level parallelism to achieve desired levels of performance [2, 6]. In addition, a recent study [13] suggests that some data-parallel applications perform better when expressed as task-parallel programs.

In this paper, we describe the design and implementation of pTask, a system for automatically detecting and exploiting dynamic task-level parallelism in sequential array-based C programs. pTask uses compiler analysis to statically determine data usage information for tasks, then uses this information at run-time to extract available concurrency and to enforce data dependence constraints. pTask targets shared memory systems; a prototype is operational on the KSR1 multiprocessor.

The remainder of this paper is organized as follows. An overview of pTask is given in Section 2, describing its model of execution and how tasks communicate. The compile-time analysis performed by pTask is described in Section 3. The run-time system of pTask is presented in Section 4. Experimental results are presented and analyzed in Section 5. Related work is reviewed in Section 6. Conclusions are given in Section 7.

2 Overview and program execution

pTask is a compiler and a run-time system that executes a sequential array-based C program, automatically creating, synchronizing and scheduling task-level parallelism. A task corresponds to an asynchronous procedure invocation. To specify a procedure to be invoked asynchronously, the keyword `TASK` is prepended to the procedure header. We refer to such procedure as a *task-procedure* and to a program containing task-procedures as a pTask program.

pTask executes a sequential C program as follows. The main routine is considered the main task and it starts executing sequentially. For each task-procedure invocation, an independent thread of control (i.e., a task) is created to asynchronously execute the body of the task-procedure. This task may run on a separate processor, concurrently with the task that created it and with other tasks in the system. It may in turn create *child* tasks (or *subtasks*) by invoking task-procedures. In general, the program can be viewed as a set of tasks executing concurrently, with each task sequentially

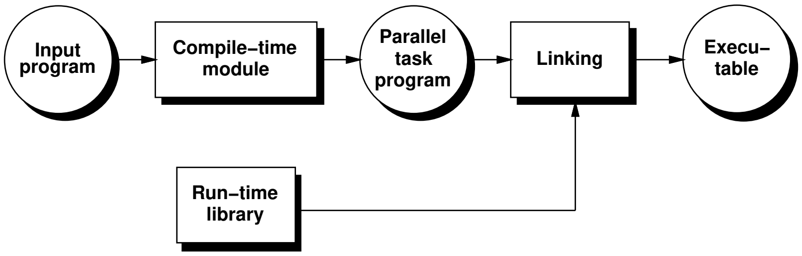


Figure 1: The pTask system.

executing the body of its associated task-procedure, and creating more tasks whenever it invokes task-procedures. A task terminates when it reaches the end of the task-procedure. The program terminates when all tasks terminate.

Tasks in pTask communicate in two ways. First, the actual parameters of a task-procedure invocation become input to the new task, making it possible for a (parent) task to pass values to its child tasks. Second, tasks may communicate by accessing (reading/writing) data in the shared memory. In a C program, global variables are accessible by all procedures, and thus are shared by all tasks. In addition to global variables, tasks may also share automatic variables or dynamically allocated data since a procedure may pass references or pointers to this data to other procedures. In general, tasks share data if they access the same data at some address in shared memory.

The flow of data in the sequential execution of the program results in data dependences among tasks. Hence, synchronization of tasks is necessary to preserve program correctness. pTask preserves a program's sequential semantics by enforcing serial execution order for tasks performing conflicting operations on the same data. That is, tasks that write the same shared data must be executed sequentially in the program's serial execution order. Similarly, serial execution order is preserved between a task that writes data and another task that reads the same data. Of course, tasks accessing different data or only reading the same data may execute concurrently.

pTask is composed of two main components: a compile-time module and a run-time system. The compile-time module compiles the input sequential program into a parallel task program which contains calls to the run-time library to create, schedule, execute, and synchronize tasks. The run-time system carries out the instructions inserted by the compile-time module. The parallel task program is then linked with the run-time library to produce an executable program. This is shown in Figure 1.

3 The compile-time module

The compile-time module performs three main functions. It summaries array accesses in statements; it translates procedure invocations into task creations; and it inserts synchronization code.

3.1 Data access summaries

The compile-time module summarizes the array accesses of a statement in the form of *access regions* [1]. An access region describes the range of elements accessed in an array and the type of access, which is either read or write. A range of accesses is defined for each dimension of the array by the lowest and highest indices of the elements accessed in that dimension. Depending on the subscripts of the array reference, these indices may be constants, variables, or expressions. The region of an array accessed is the cross product of the range in each dimension of the array. Hence, the region is the smallest rectangular space enclosing all elements accessed. Figure 2 shows example regions of an $m \times m$ array A.

The *union* of two or more regions is the smallest region that encloses these regions. Region union is useful for computing data access summaries for a sequence of statements in a program. The *intersection* of two or more regions is the subregion in which these regions overlap. Region intersection is used to determine if two statements access the same data. Two regions are said to be in *conflict* if their intersection is not empty and at least one of them has a write access type.

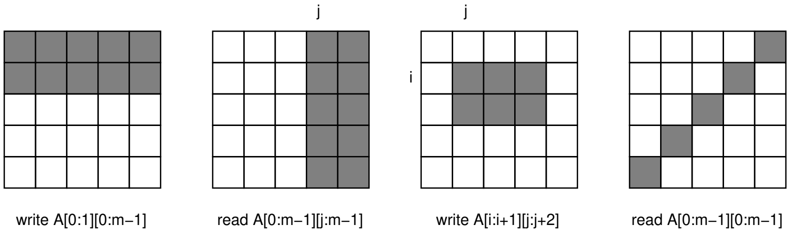


Figure 2: Example data access summaries in an $m \times m$ array.

3.2 Task-procedure conversion

The compile-time module converts procedure invocations into calls to the run-time system to create tasks. Each call passes to the run-time system a pointer to the procedure, the actual arguments of the procedure, a descriptor of the regions accessed by the procedure invocation, and an identifier for the task.

3.3 Synchronization

A task may access a region only if such access does not violate the execution semantics of the sequential program. Hence, tasks in pTask are synchronized according to the following rule: *a task can access a region if the region is not in conflict with regions of the tasks that execute and complete earlier (than the task) as procedures in the sequential program.*

The run-time system provides a function called `region_check` which, given one or more regions of a task, determines whether the task can access the region(s), and if necessary causes the task to wait until it can access the region(s). The run-time system also provides a function called `region_signal` which, given one or more regions of a task, allows tasks waiting on the region(s) to proceed. The implementation of these functions is described in Section 4. The compile-time module obtains regions of all statements in the program, and inserts calls to these functions where synchronization is required. In general, at each point where synchronization is required, a task would call a `region_check` on region(s) before accessing these region(s), and calls a `region_signal` after having accessed the region(s).

In a program, synchronization is necessary at four specific locations: at entry of task-procedures, within the body of procedures, at exit points of program blocks, and at program exit points. The following sections provide the motivations and the methods for synchronization at these locations.

3.3.1 Task-entry synchronization

The invocation of a task-procedure results in a new task which executes the body of the task-procedure asynchronously. Synchronization is required to ensure the new task accesses its regions only if such accesses do not violate the synchronization rule. For example, the program in Figure 3 creates a task `t1`, followed by task `t2`. Task `t2` must not proceed to access `sum` until the variable is written by task `t1`. This type of synchronization is referred to as *task-entry* synchronization.

Task-entry synchronization is achieved by having the compile-time module identify the first access to each variable within the task-procedure and precede each first access with a `region_check` on those regions of the task associated with the variable. Similarly, the compile-time module identifies the last access to each variable within the task-procedure and follows each last access with a `region_signal`. Thus, a task may execute the body of the associated task-procedure as soon as it is created, but it must check for the regions of each variable before the first access to the variable. This approach enables the system to better exploit concurrency since regions of a variable are available to waiting tasks as soon as earlier tasks finish accessing the variable. Also, a task is allowed to execute until it attempts to access regions that it must wait for.

3.3.2 Task-body synchronization

Task-entry synchronization ensures that a task proceeds to execute only if all existing dependence constraints are satisfied. However, an executing task may create child tasks and continue to access

```
int sum;
main() {
    t1();
    t2();
}

/* TASK() */
void t2 () {
    ··· = sum;
}
```

```
/* TASK() */
void t1 () {
    sum = ···;
}
```

Figure 3: Task-entry synchronization.

```
/* TASK() */
void t1 (int *b) {
    t2 (b);
    /* statement s */
    b[0] = ··· ;
}

/* TASK() */
void t2 (int *b) {
    b[0] = ···;
}
```

Figure 4: Task-body synchronization.

```
f0 () {
    int a[N], b[N], c[N];
    t1 (a);
    f1 (b);
    f2 ();
    t2 (c);
}
```

```
f1 (int *p) {
    t3 (p);
}

f2 () {
    int d[N];
    t4 (d);
}
```

```
/* TASK() */
void t1 (int *p) {
    /* access p */
}

/* TASK() */
void t2 (int *p) {
    t5 (p);
}
```

Figure 5: Block-exit synchronization. Task-procedures `t3`, `t4` and `t5` are identical to `t1`.

data that is also accessed by the child tasks, and hence additional synchronization is required to enforce these new dependences. For example, in the program shown in Figure 4 synchronization is required in task-procedure `t1` to ensure that statement `s` is executed only after its child task `t2` has completed; otherwise the dependence between task `t2` and statement `s` is violated. This type of synchronization is referred to as *task-body* synchronization.

To implement this form of synchronization, the compile-time module collects adjacent statements that do not create subtasks (either directly or indirectly) into groups. A statement that creates a task is considered a group by itself. The compile-time module obtains regions of each group, and surrounds each group with a `region_check` and a `region_signal`. Consequently, a task can execute a statement concurrently with its subtasks if the regions of the statement and the regions of the subtasks are not in conflict.

3.3.3 Block-exit synchronization

Synchronization is required at exit points of a program block because a block may directly or indirectly create subtasks that access its local (i.e., automatic) variables. For such blocks, the run-time system must ensure that the task executing the block leaves the block only after the subtasks have finished accessing the local variables; otherwise these variables would be de-allocated, causing the subtasks to access invalid data. For example, synchronization is needed to ensure that the task executing procedure `f2` in Figure 5 returns only after task `t4` has terminated, because `t4` accesses array `d`.

In order to provide this form of synchronization, an executing task is suspended until all subtasks that access its local variables have completed. The insertion of synchronization code is complicated by the fact that a task may access variables that are automatically allocated elsewhere in the program. In general, a task may access data allocated in a program block of any ancestor task. For instance, in Figure 5, task `t4` accesses the data allocated in procedure `f2` of its parent, while task `t5` accesses the data allocated in procedure `f0`, its grandparent. The remainder of this section details how pTask provides this type of synchronization.

We denote a program block that directly or indirectly creates tasks that access its local variables as a *local-access* block. Exit points of local-access blocks are preceded with code causing execution to wait for all tasks created from within. In Figure 5, the compile-time module must insert code to cause the task executing procedure `f0` to wait for task `t4` at the end of the local-access block in procedure `f2` and for tasks `t1`, `t2` and `t3` at the end of the local-access block in procedure `f0`.

In order for a task `A` to wait for a set of subtasks created from within a particular local-access block, it must know the number of subtasks created in that local-access block. Similarly, in order for these subtasks to signal task `A` to continue, each of the subtasks must know in which of task `A`'s local-access blocks it was created. This is facilitated by having each task maintain an integer called the `block_number` which is initialized to zero at the beginning of the task-procedure. At the top of each local-access block, the compile-time module inserts code to increment the task's `block_number`. Before the exit points of each local-access block, code is inserted to decrement the task's `block_number`. When a task is created, the current `block_number` of its parent is passed to it. Also the parent keeps count of the number of subtasks created at a particular `block_number`. Consequently, a task knows in which of its parent task's `block_number` it was created, and it also knows the number of subtasks it created in a particular `block_number`. To force a task to wait for the subtasks created from within a local-access block, the compile-time module simply precedes the decrement of the `block_number` with code to cause the task to wait for the subtasks having the current `block_number`.

3.3.4 Program-exit synchronization

A sequential program terminates upon invoking an `exit`. In a pTask-generated program, since any task may invoke exit, synchronization is required to ensure proper program termination. Regardless which of the tasks invokes exit, the parallel task program must terminate. On most parallel systems, if the main thread invokes exit then all threads in the program terminate, but if any other thread invokes exit then only that thread terminates. In pTask, the task that invokes exit (referred as the exit-task) must terminate, and in addition, states of all other tasks, such as its subtasks, its dependent tasks, and tasks already running in the system, must be determined.

pTask handles program-exit synchronization by having the compile-time module replace every exit call by a `sys_exit` call to the run-time system. This procedure:

1. Causes the calling task (i.e., the exit-task) to wait for its subtasks. This is necessary because in the sequential execution, these subtasks would have been executed before exit is invoked.
2. Calls exit to terminate the program if the calling task is the main task. Since every task waits for its subtasks and the main task is the ancestor of all tasks, this implies the program terminates when all tasks have terminated. In other words, this results normal program termination.
3. Signals the main task to indicate that an exit has been invoked if the calling task is not the main task. Some running tasks may be prematurely terminated and hence the program does not terminate normally. Such behavior is still valid because the prematurely terminated tasks must had been accessing different data from that accessed by the exit-task; whether they get to run to completion or not does not change the intended sequential behavior of the program.

4 The run-time system

The run-time system provides an environment to create, execute and synchronize tasks. It is initialized and taken down by calls inserted by the compile-time module at the entry and exit points of the main procedure, respectively. The system consists of a global task queue, and a set of p workers where p is the number of available processors. A worker is a thread running on a separate processor.

Once a task is created, it is added to the global task queue. Workers remove tasks from the global task queue and execute them. Each worker also has a local task queue which is read by the worker itself and can be written to by other workers. If a worker receives a task that must be executed by another worker, or that can be more efficiently executed by another worker then it adds the task directly to the other worker's local task queue. As will be explained later in this section, a suspending task must be resumed on the same worker that was executing it, and it may be more efficient to execute a task on the worker that most recently accessed the regions of the task. Tasks on local task queues are given higher priority. Hence, a worker always removes tasks from its local queue before tasks from the global queue. Figure 6 illustrates the run-time system.

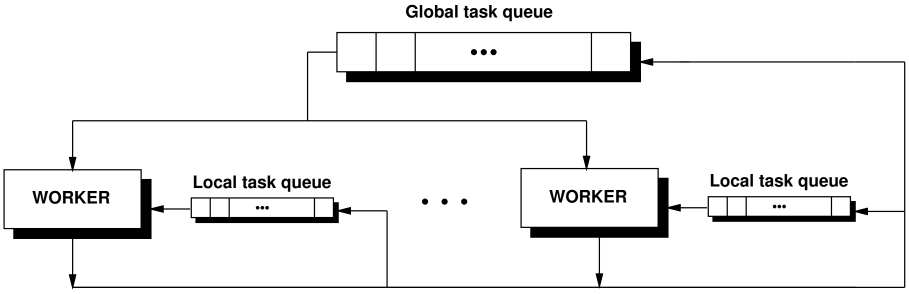


Figure 6: The run-time system.

4.1 Task synchronization

Tasks are synchronized based on data regions they access. A task `T` can access a region if the region is not in conflict with regions of tasks that execute and complete as procedures before `T` in the sequential program. Such tasks are referred to as *earlier* tasks. The order in which tasks execute as procedures in the sequential program is referred to as the *serial execution order*.

Since tasks are created concurrently as task procedures are invoked, the order in which tasks are created at run-time is not necessarily the serial execution order. The run-time system maintains a set of data structures referred to as *region-lists* for the purpose of maintaining and enforcing serial execution order.

4.1.1 Region-lists

Dynamically created and maintained by the run-time system, a region-list is a list of all regions of a variable currently being accessed by tasks in the program. Each node on the list contains the descriptor of the region, the identifier of the task that accesses the region, and a list of pointers to tasks waiting for this region. For simplicity, we refer these nodes as regions on the region-lists. Each region-list is maintained such that the order of the regions on the list reflects the serial order in which tasks access the corresponding variable. That is if tasks `t1` and `t2` access a variable `A`, and task `t1` accesses `A` earlier than task `t2` according to the serial execution order, then on the region-list of `A`, the regions of task `t1` are in front of those of task `t2`. Figure 7 shows an example program and the associated region lists. All tasks are assumed to access subregions of an array variable `A`.

Three operations are associated with region-lists: `region_reserve`, `region_unreserve`, and `region_check`. The first two operations insert and remove regions to and from region-lists, respectively. The third operation determines whether a region on a list is in conflict with regions of the earlier tasks.

4.1.2 Region_reserve

Given a region of a task, this operation atomically inserts the region onto the region-list of the associated variable. The region is inserted such that its position on the list reflects the serial data access order of tasks.

In a sequential program, when a procedure invokes subprocedures, the subprocedures execute one at a time, each to completion, in the order in which they are invoked. If these subprocedures access the same data, then they would access the data in the order in which they are invoked, and a subprocedure always completes its access to the data before returning to the calling procedure. In the framework of pTask where a procedure invocation corresponds to a task, this implies sibling tasks access the data in the order in which they are invoked, and a task always completes its access to the data before its parent continues to access the same data. Thus, regions of a task should be inserted in front of the regions of its parent, and among the regions of its siblings, reflecting the order in which it and its siblings are invoked as procedures in the sequential program. Since a task creates child tasks sequentially, reserving a region of a task essentially amounts to inserting the region immediately in front of the regions of its parent– provided the parent task reserves all regions of a child task when it creates the task, and reserves regions of its child tasks in the order in which they are created. This is the case as described in Section 4.3.1. This ensures that for any two tasks, `t1` and `t2`, where `t1`

```
main () {          /* TASK() */        /* TASK() */        /* TASK() */          /* TASK() */
    t1 ();         void t1 () {        void t2 () {        void t1_1 () {        void t2_1 () {
    t2 ();             t1_1 ();            t2_1 ();            ...                   ...
}                  }                   }                   }                     }
```

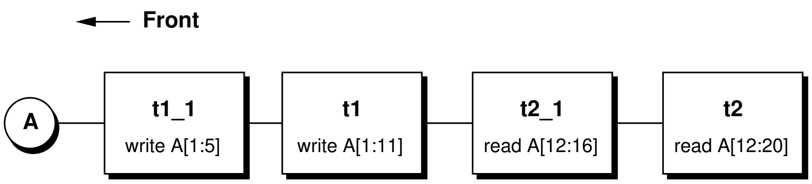


Figure 7: An example program and associated region list.

is earlier than `t2`, and both access some common variable then, on the region-list of the variable, regions of `t1` are always in front of those of `t2`. A formal proof can be found in [7].

4.1.3 Region_unreserve

Given a region on a region-list, this operation atomically removes the region from the region-list. If there are tasks waiting for this region then `region_unreserve` resumes these tasks. The process of task resumption is explained in Section 4.3.3. The `region_signal` operation in the previous section directly translates into this operation.

4.1.4 Region_check

While the previous two operations are used to maintain task serial data access order, this operation is used to detect inter-task data dependencies. Given a region on a region-list, this operation determines whether the region is in conflict with those of the earlier tasks. This operation involves traversing the associated region-list starting from the given region toward the front of the list, checking the region against those on the path until the head of the list is reached or a conflicting region is found; two regions are in conflict if their intersection is not empty and at least one has the write access type. In the case of a conflict, `region_check` returns the conflicting region. For example, given the region-list in Figure 7, a `region_check` on the region of task `t2` would return no conflicting region, while a `region_check` on the region of task `t1` would return the region of task `t1_1`.

4.2 Enforcing the synchronization rule

The run-time system maintains sequential execution semantics. That is, given any two arbitrary tasks `t1` and `t2` such that `t1` is earlier than `t2`, and they access some common data, the run-time system will always detect and enforce the data dependencies that exist between these two tasks. This can be easily seen because:

1. The `region_reserve` operation will always insert the regions of `t1` ahead of the regions of `t2` on region-lists to indicate that task `t1` is earlier to task `t2`, regardless of the order of their creations.

2. `t1` and `t2` each execute a `region_check` before accessing the data and a `region_signal` after accessing the data. If a region of task `t2` is in conflict with a region of task `t1` then `region_check` returns the conflicting region and task `t2` will be forced to wait until task `t1` issues a `region_signal` on the conflicting region. When `region_check` on the regions of task `t2` returns no conflicting region, then the run-time system allows `t2` to access its regions.

4.3 Task activities

4.3.1 Task creation

A `create_task` call in the parallel program performs three functions. First, it creates a data structure representing the new task. Contained within this structure are the task identifier, the parent task identifier, the pointer to the associated task-procedure, the task-context, and the regions of the task-procedure invocation. Next, it establishes the serial data access order for the new task by reserving *all* regions of the task using the `region_reserve` operation. Finally, it adds the new task to the global task queue.

4.3.2 Task execution

A worker executes a task as soon as it receives the task. During the execution of the task, if a `region_check` returns no conflicting region, the execution is continued. Otherwise, the worker adds a pointer to the task to the conflicting region's list of waiting tasks, saves the task's execution context, and fetches another task.

4.3.3 Task resumption

Once a task completes its accesses to a set of regions, a `region_unreserve` is called for each of these regions. For each unreserved region, the worker obtains the list of tasks waiting on the region and adds each of these tasks to the local task queue of the worker that previously executed and suspended the task. This worker would eventually restore the task's execution context and continue with its execution. This restriction simplifies the design of the run-time system by not allowing task contexts to migrate.

In the case where a task must wait for its child tasks to terminate, the child task which terminates last adds the parent task back to the local task queue of the worker on which the parent task was executing before being suspended. This worker would eventually resume the parent task.

4.4 Scheduling for data locality

The default scheduling strategy is to execute a task on the first idle worker that becomes available. This strategy maintains the load among the processors fairly balanced and incurs little run-time overhead. However, it does not take into account the fact that on most scalable shared-memory multiprocessors, the time it takes a processor to access data depends on the location of the data with respect to the processor. The farther away data is from a processor, the longer it takes the processor to access the data. Thus, this default strategy may result in poor data locality. On the other hand, restricting a task to execute only on the processor closest to the data needed by the task is likely to result in poor load balance.

The run-time system provides a locality-conscious scheduling strategy which exploits data locality without sacrificing load balance. Given a task, this strategy assigns the task to the *idle* worker that has highest data affinity for the task. This scheduling strategy incurs more run-time overhead since task-processor data affinity must be determined at run-time.

4.4.1 Accessed region-lists

For each region-list, the run-time system maintains a corresponding list of unreserved regions. The `region_unreserve` operation appends removed regions from a region-list to the corresponding unreserved list. These unreserved lists are referred to as *accessed-lists* because regions on these lists are regions that have already been accessed by tasks. Each node on the accessed-list has the identifier of the worker which executed the task that accessed the region. For simplicity, we refer these nodes also as regions on the accessed-lists. Since regions are appended onto these lists in roughly the order in which they were accessed, regions at the rear of an accessed-list are more recently accessed compared to those at the front.

```
for each region (called LESSR_R for less recent) from
    the rear to the front of the associated accessed-list
        LESSR_ISIZE = number of elements R and LESSR_R intersect
        if (LESSR_R is the region at the rear)
            LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
        else if (LESSR_ISIZE > LATEST_ISIZE)
            if (LATEST_R is of read type)
                LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
            else
                if (WORKERID of LESSR_R == WORKERID of LATEST_R
                    or LESSR_ISIZE - LATEST_ISIZE > LATEST_ISIZE)
                        LATEST_R = LESSR_R, LATEST_ISIZE = LESSR_ISIZE
                endif
            endif
        endif
        if (LATEST_ISIZE = size of R)
            return WORKERID of LATEST_R, and LATEST_ISIZE
        endif
endfor
return WORKERID of LATEST_R, and LATEST_ISIZE
```

Figure 8: Task-worker affinity algorithm.

4.4.2 Measuring task-worker data affinity

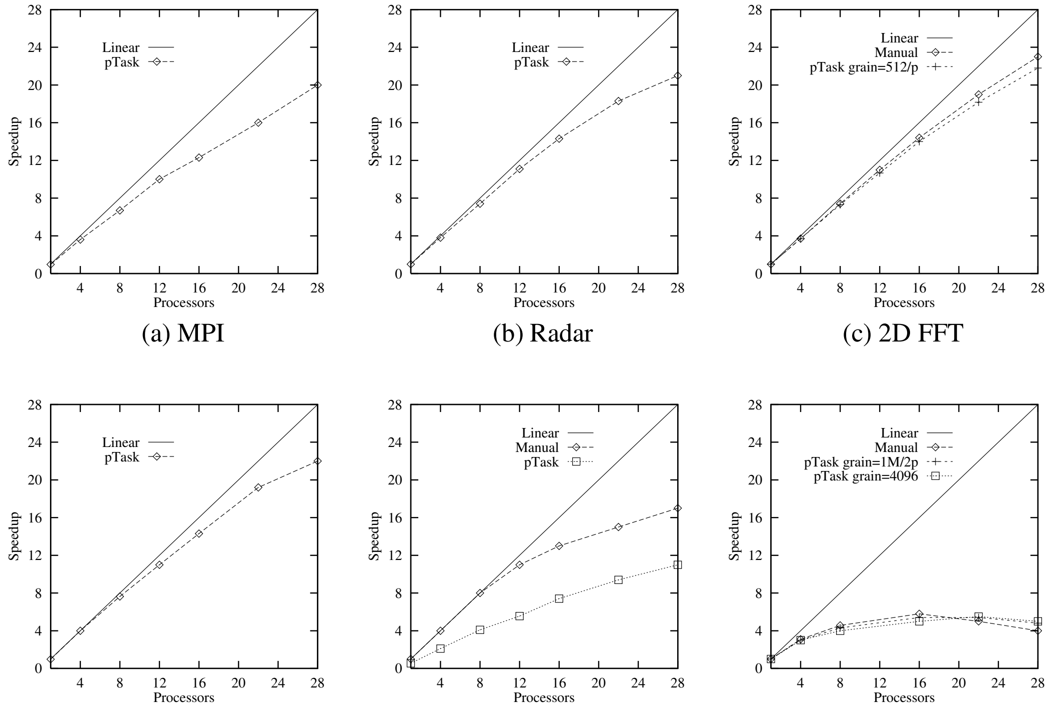
Accessed-lists are used to measure task-worker data affinity. Since a task may access many regions that were accessed by tasks executed on different workers, there may be more than one worker having data affinity for the task. Given a task that is ready for execution, for each of the task's regions, the worker determines the set of workers having data affinity for the region and the extent of affinity that each of these workers has for the region. The worker computes data affinity for all regions for each of these workers and assigns the task to the idle worker having the highest data affinity.

The algorithm shown in Figure 8 determines the worker that has the highest data affinity for a given region. Given a region, a worker locates the latest copy of the region by traversing the associated accessed-list from the rear toward the front, intersecting the given region with those on the path. We use the amount of data in the intersection to determine how much of the given region was accessed by a worker. We also use the accessed-regions' access types to determine whether the elements accessed are the most up-to-date values. The algorithm takes into account the fact that an accessed-region may only intersect part of the given region, and the intersection of the last (as positioned on the accessed-list) accessed-region of write access type contains latest elements of the given region. The algorithm stops when the front of the accessed-list is reached, or when an intersection that is the latest copy of the given region is found. The worker that accessed most of the latest copy is the one having the highest data affinity for the given region.

5 Experimental results

We have implemented a prototype of pTask using the Sigma II toolkit [5], which is a compiler infrastructure for building source-to-source translators and performance analysis tools. Sigma II is used to parse the input program, to calculate regions, and to generate the output code. Presently, Sigma II cannot analyze regions of `while` or `do-while` loops, of recursive procedures, and of array accesses that use pointers. Hence, `while` and `do-while` loops must be re-coded into `for` loops; calls to recursive procedures must be annotated [7]; and array data must be accessed by array indexing.

In this section we report on the performance of applications using the prototype on a 32-processor KSR1 system. We compare the performance of the pTask-generated programs to that of manually parallelized versions of the programs when possible. We use the speedup, defined as the ratio of the execution time of the sequential program to the execution time of the parallel program, to measure performance improvements. We use the overhead ratio, defined as the ratio of the time spent in the run-time system during parallel execution to sequential execution time, to measure the extent of the overhead introduced by pTask.



(a) MPI (b) Radar (c) 2D FFT

(d) Search (e) TSP (f) Mergesort

Figure 9: Speedup of applications using pTask.

Multidimensional polynomial interpolation (MPI) [11] interpolates an n-dimensional function. The program makes n calls to a procedure that performs a 1-dimensional interpolations, and then uses the same procedure to perform a final interpolation using the results computed by the previous set of calls. We declare this procedure as a task procedure, allowing n tasks to execute concurrently, followed by one task that performs the final interpolation. The performance of the pTask-generated program for $n = 256$ is shown in Figure 9(a). The overhead is below 5% of sequential execution time.

The narrowband tracking radar benchmark [3] (radar for brief) was developed by researchers at MIT Lincoln Labs to identify targets from a sequence of radar returns. The program inputs data from a single sensor along a number of independent channels. Over each channel, the program receives a 512 complex vectors each of length 10 every 5 milliseconds. The vectors fill a 512×10 input matrix, which is then processed in the following way: (1) the matrix is corner-turned into a 512×10 matrix; (2) 10 independent 512-point FFTs are applied to each row of the matrix; (3) the resulting complex matrix is reduced to a real 40×10 submatrix; and finally, (4) the elements of the submatrix are threshold based on the sum of the submatrix elements. The procedures that receive the data, fill the matrix from the input vectors, corner-turn the matrix and perform the FFT are declared as task-procedures. The performance of the pTask-generated program is shown in Figure 9(b) for 32 channels. The run-time overhead is quite low and this is reflected in a speedup that is close to linear.

The 2-dimensional FFT program consists of two passes. The first pass performs independent 1-dimensional FFTs on the rows of the input matrix. The second pass performs independent 1-dimensional FFTs on the columns of the matrix. A task is created to perform a 1-dimensional FFT on a set of rows or columns. The system exploits parallelism by executing tasks in each pass concurrently. Figure 9(c) shows the speedup of the parallel program using a 512×512 input matrix. The pTask curve shows the speedup when the granularity of a task is $512/p$ rows or columns, where p is the number of processors. The run-time overhead is 0.15% of sequential execution time. The performance is almost identical to that of a manually parallelized data-parallel version.

Search [8] is a program from Stanford University that uses a Monte-Carlo method to simulate the elastic scattering of electrons from an electron beam into various solids. The main computation performed by the program simulates six different solids for 10 initial values of beam energy. Each solid-energy simulation tracks 5,000 electron trajectories. With pTask we declare the procedure that

Application	p=16	p=22	p=28
Radar	3%	5%	5%
Mergesort	16%	13%	21%

Table 1: Impact of locality-conscious scheduling on Radar and Mergesort.

performs one solid-energy simulation as a task-procedure. Tasks that simulate different solid-energy points can execute concurrently. The performance of the pTask-generated program is shown in Figure 9(d). The speedup for this application is close to linear. There is negligible run-time overhead because the sizes of tasks in this application are relatively large.

The Travelling Salesman Problem (TSP) program is an example of an application which cannot be expressed in a data-parallel form. The sequential program solves a TSP instance using a branch-and-bound algorithm. Parallelism is exploited by concurrently traversing branches in the branch-and-bound tree; a task is created for each call to a procedure that traverses a fixed number of paths in the tree, and then updates the current bound. Figure 9(e) shows the speedup of the parallel program solving a 11-city problem. Although the performance of the automatically generated parallel program is worse than that of a manually parallelized version, it scales with the number of processors. The overhead of the run-time system is about 5% of computation time.

The speedup of a mergesort program (1 M integers), is shown in Figure 9(f). The automatically generated program performs almost identical to a manually parallelized version. The speedup is low, but it is close to what is attainable from divide-and-conquer parallelism [9].

The impact of the locality-conscious scheduling strategy is summarized in Table 1. It shows the improvement in performance obtained by using the locality-conscious strategy as opposed to the simple default strategy. It indicates performance benefits for radar and mergesort. There is no benefit for the other four applications, but there is no performance degradation either. The lack of benefit for these applications stems from the lack of data reuse in each application, making a locality-conscious strategy unnecessary. The overhead of scheduling is too small to degrade performance.

6 Related work

Dongarra and Sorenson [4] describe `Schedule`, in which programmers explicitly partition their programs into tasks, and analyze these tasks to determine inter-task dependences. A task is a subroutine call, and programmers specify task dependencies by supplying the system with a unique identifier for each task, the number of prerequisite tasks, and the number of dependent tasks and their identifiers.

Yang and Gerasoulis [14] implement `Pyrros`, in which only static task-parallelism is allowed and programmers are required to provide estimates of task execution times. The system generates the output program and the scheduling of tasks to processors.

Lam and Rinard [8] describe the `Jade` language in which programmers augment sequential programs with high-level data usage constructs. In `Jade`, programmers specify tasks and each task's side effects. The run-time system detects concurrency and enforces data dependences. In this respect, pTask is similar to `Jade`, particularly in the run-time system. However, in pTask programmers only identify tasks; the compiler automatically determines task side effects and inserts synchronization.

Gross, O'Hallaron, and Subholk [6] design and implement a compiler based on HPF called `Fx`. It extends HPF to allow programmers to specify task parallelism. `Fx` defines a task as a procedure invocation, but restricts parallelism to programmer-defined parallel sections. Furthermore, it requires programmers to supply directives that define input/output parameters of each task, and the mapping of tasks onto processors.

Markatos and LeBlanc [10] describe an affinity conscious scheduling algorithm for parallel loops. Assuming an HPF-like data distribution for arrays accessed in a parallel loop, the iterations of the loop are dynamically assigned to processors to both enhance locality and balance the workload.

7 Conclusions

In this paper, we have described the design and implementation of pTask, a system that automatically exploits task-level parallelism in sequential `C` programs. The system executes a program in parallel while dynamically detecting and enforcing dependences among tasks to maintain sequential execution semantics. The system is implemented on a KSR1 multiprocessor and the performance of applications validate our approach– the system incurs low overhead at run-time and results in good performance improvements. A scheduling strategy which takes into consideration data locality was presented and shown to improve the performance of applications in which data reuse exists.

References

[1] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: the data access descriptor," *Journal of Parallel and Distributed Computing,* vol. 9, no. 6, pp. 154–170, 1990.

[2] B. Chapman, H. Zima and P. Mehrota, "Extending HPF for advanced data-parallel applications," *IEEE Parallel and Distributed Technology,* vol. 2, no. 3, pp. 59–70, 1994.

[3] P. Dinda, et al., The CMU task parallel program suite, Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March, 1994.

[4] J. Dongarra and D. Sorenson, "A portable environment for developing parallel FORTRAN programs," *Parallel Computing,* vol. 5, no. 2, pp. 175–186, 1987.

[5] D. Gannon, et al., "SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems," in Programming Environments for Parallel Computing, N. Topham and T. Bemmerl (eds.), pp. 17–36, Elsevier Science Publishers, 1992.

[6] T. Gross, D. O'Hallaron and J. Subholk, "Task parallelism in High Performance Fortran framework," *IEEE Parallel and Distributed Technology,* vol. 2, no. 3, pp. 16–26, 1994.

[7] S. Huynh, Exploiting task-level parallelism automatically using pTask, M.A.Sc. Thesis, University of Toronto, Toronto, 1996.

[8] M. Lam and M. Rinard, "Coarse-grain parallel programming in Jade," *Proc. ACM PPOPP*, pp. 94–105, 1991.

[9] T. Lewis and H. El-Rewini, *Introduction to Parallel Computing,* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[10] E. Markatos and T. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *Proc. Supercomputing,* pp. 104–113, 1992.

[11] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical recipes in C: the art of scientific computing,* Cambridge University Press, 1988.

[12] H. Printz, et al., "Automatic mapping of large scale signal processing systems to a parallel machine," *Proc. SPIE Symp. on Real-Time Signal Processing,* pp. 2–16, 1989.

[13] J. Subholk, et al., "Exploiting task and data parallelism on a multicomputer," *Proc. ACM PPOPP*, pp. 13–22, 1993.

[14] T. Yang and A. Gerasoulis, "PYRROS: Static task scheduling and code generation for message passing multiprocessors," *Int'l Conf. on Supercomputing,* pp. 428–437, 1992.