T

Task Graph Scheduling

Yves Robert
Ecole Normale Supérieure de Lyon, France

Synonyms

DAG scheduling; Workflow scheduling

Definition

Task Graph Scheduling is the activity that consists in mapping a task graph onto a target platform. The task graph represents the application: Nodes denote computational tasks, and edges model precedence constraints between tasks. For each task, an assignment (choose the processor that will execute the task) and a schedule (decide when to start the execution) are determined. The goal is to obtain an efficient execution of the application, which translates into optimizing some objective function, most usually the total execution time.

Discussion

Introduction

Task Graph Scheduling is the activity that consists in mapping a task graph onto a target platform. The task graph is given as input to the scheduler. Hence, scheduling algorithms are completely independent of models and methods used to derive task graphs. However, it is insightful to start with a discussion on how these task graphs are constructed.

Consider an application that is decomposed into a set of computational entities, called *tasks*. These tasks are linked by *precedence constraints*. For instance, if some task T produces some data that is used (read) by another tasks T', then the execution of T' cannot start before the completion of T. It is therefore natural to represent the application as a *task graph*: The task graph is a DAG (Directed Acyclic Graph), whose nodes are the tasks and whose edges are the precedence constraints between tasks.

The decomposition of the application into tasks is given to the scheduler as input. Note that the task graph may be directly provided by the user, but it can also be determined by some parallelizing compiler from the application program. Consider the following algorithm to solve the linear system $Ax = b$, where A is an $n \times n$ nonsingular lower triangular matrix and b is a vector with n components:

for $i = 1$ ***to*** n **do**
  Task $T_{i,i}$: $x_i \leftarrow b_i / a_{i,i}$
  for $j = i + 1$ **to** N **do**
    Task $T_{i,j}$: $b_j \leftarrow b_j - a_{j,i} \times x_i$
  end
end

For a given value of i, $1 \leqslant i \leqslant n$, each task $T_{i,*}$ represents some computations executed during the i-th iteration of the external loop. The computation of x_i is performed first (task $T_{i,i}$). Then, each component b_j of vector b such that $j > i$ is updated (task $T_{i,j}$). In the original sequential program, there is a total precedence order between tasks. Write $T <_{seq} T'$ if task T is executed before task T' in the sequential code. Then:

$$T_{1,1} <_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \cdots <_{seq} T_{1,n} <_{seq} T_{2,2} <_{seq} T_{2,3} <_{seq} \cdots <_{seq} T_{n,n}.$$

However, there are independent tasks that can be executed in parallel. Intuitively, independent tasks are tasks whose execution orders can be interchanged without modifying the result of the program execution. A necessary condition for tasks to be independent is that they do not update the same variable. They can read the same value, but they cannot write into the same memory location (otherwise there would be a race condition and the result would be nondeterministic). For instance,

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,

tasks $T_{1,2}$ and $T_{1,3}$ both read x_1 but modify distinct components of b, hence they are independent.

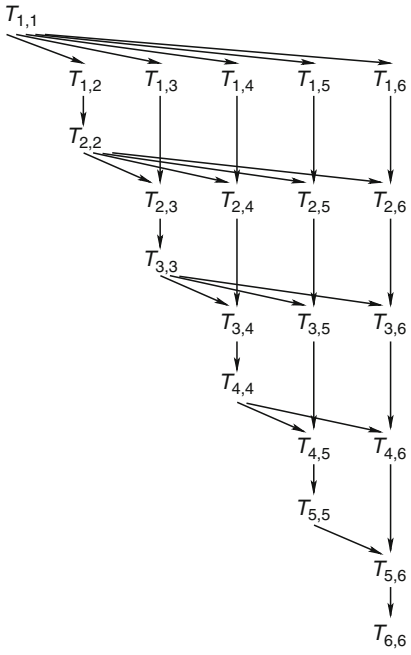
This notion of independence can be expressed more formally. Each task T has an input set $\mathrm{In}(T)$ (read values) and an output set $\mathrm{Out}(T)$ (written values). In the example, $\mathrm{In}(T_{i,i}) = \{b_i, a_{i,i}\}$ and $\mathrm{Out}(T_{i,i}) = \{x_i\}$. For $j > i$, $\mathrm{In}(T_{i,j}) = \{b_j, a_{j,i}, x_i\}$ and $\mathrm{Out}(T_{i,j}) = \{b_j\}$. Two tasks T and T' are not independent (write $T \perp T'$) if they share some written variable:

$$T \perp T' \Leftrightarrow \begin{cases} & \mathrm{In}(T) \cap \mathrm{Out}(T') & \neq \emptyset \\ \text{or} & \mathrm{Out}(T) \cap \mathrm{In}(T') & \neq \emptyset \\ \text{or} & \mathrm{Out}(T) \cap \mathrm{Out}(T') & \neq \emptyset \end{cases}$$

For instance, tasks $T_{1,1}$ and $T_{1,2}$ are not independent because $\mathrm{Out}(T_{1,1}) \cap \mathrm{In}(T_{1,2}) = \{x_1\}$; therefore $T_{1,1} \perp T_{1,2}$. Similarly, $\mathrm{Out}(T_{1,3}) \cap \mathrm{Out}(T_{2,3}) = \{b_3\}$, and hence $T_{1,3}$ and $T_{2,3}$ are not independent; hence $T_{1,3} \perp T_{2,3}$.

Given the dependence relation $\perp$, a partial order $\prec$ can be extracted from the total order $<_{seq}$ induced by the sequential execution of the program. If two tasks T and T' are dependent, that is, $T \perp T'$, they are ordered according to the sequential execution: $T \prec T'$ if both $T \perp T'$ and $T <_{seq} T'$. The precedence relation $\prec$ represents the dependences that must be satisfied to preserve the semantics of the original program; if $T \prec T'$, then T was executed before T' in the sequential code, and it has to be executed before T' even if there are infinitely many resources, because T and T' share a written variable. In terms of order relations, $\prec$ is defined more accurately, as the transitive closure of the intersection of $\perp$ and $<_{seq}$, and captures the intrinsic sequentiality of the original program. Note that transitive closure is needed to track dependence chains. In the example, $T_{2,4} \perp T_{4,4}$ and $T_{4,4} \perp T_{4,5}$, hence a path of dependences from $T_{2,4}$ to $T_{4,5}$, while $T_{2,4} \perp T_{4,5}$ does not hold.

A directed graph is drawn to represent the dependence constraints that need to be enforced. The vertices of the graph denote the tasks, while the edges express the dependence constraints. An edge $e : T \to T'$ in the graph means that the execution of T' must begin only after the end of the execution of T, whatever the number of available processors. Transitivity edges are not drawn, as they represent redundant information; only predecessor edges are shown. T is a predecessor of T' if



Task Graph Scheduling. Fig. 1 Task graph for the triangular system ($n = 6$)

$T \prec T'$ and if there is no task T'' in between, that is, such that $T \prec T''$ and $T'' \prec T'$. In the example, predecessor relationships are as follows (see Fig. 1):

- $T_{i,i} \prec T_{i,j}$ for $1 \leqslant i < j \leqslant n$
  (the computation of x_i must be done before updating b_j at step i of the outer loop).
- $T_{i,j} \prec T_{i+1,j}$ for $1 \leqslant i < j \leqslant n$
  (updating b_j at step i of the outer loop is done before reading it at step $i + 1$).

In summary, this example shows how an application program can be decomposed into a task graph, either manually by the user, or with the help of a parallelizing compiler.

Fundamental Results

Traditional scheduling assumes that the target platform is a set of p identical processors, and that no communication cost is paid. Fundamental results are presented in this section, but only two proofs are provided, that of Theorem 1, an easy result on the efficiency of a schedule, and that of Theorem 4, Graham's bound on list scheduling.

Definitions

Definition 1 *A task graph is a directed acyclic vertex-weighted graph $G = (V, E, w)$, where:*

- *The set V of vertices represents the tasks (note that V is finite).*
- *The set E of edges represents precedence constraints between tasks:*
  $e = (u, v) \in E$ if and only if $u \prec v$.
- *The weight function $w : V \longrightarrow \mathbb{N}^*$ gives the weight (or duration) of each task. Task weights are assumed to be positive integers.*

For the triangular system (Fig. 1), it can be assumed that all tasks have equal weight: $w(T_{i,j}) = 1$ for $1 \leqslant i \leqslant j \leqslant n$. On a contrary, a division could be considered as more costly than a multiply-add, leading to a larger weight for diagonal tasks $T_{i,i}$.

A schedule σ of a task graph is a function that assigns a start time to each task.

Definition 2 *A schedule of a task graph $G = (V, E, w)$ is a function $\sigma : V \longrightarrow \mathbb{N}^*$ such that $\sigma(u) + w(u) \leqslant \sigma(v)$ whenever $e = (u, v) \in E$.*

In other words, a schedule must preserve the *dependence constraints* induced by the precedence relation $\prec$ and embodied by the edges of the dependence graph; if $u \prec v$, then the execution of u begins at time $\sigma(u)$ and requires $w(u)$ units of time, and the execution of v at time $\sigma(v)$ must start after the end of the execution of u. Obviously, if there was a cycle in the task graph, no schedule could exist, hence the restriction to acyclic graphs (DAGs).

There are other constraints that must be met by schedules, namely, *resource constraints*. When there is an infinite number of processors (in fact, when there are as many processors as tasks), the problem is *with unlimited processors*, and denoted $\mathrm{Pb}(\infty)$. When there is only a fixed number p of available processors, the problem is *with limited processors*, and denoted $\mathrm{Pb}(p)$. In the latter case, an allocation function $\mathsf{alloc} : V \longrightarrow \mathcal{P}$ is required, where $\mathcal{P} = \{1, \ldots, p\}$ denotes the set of available processors. This function assigns a target processor to each task. The resource constraints simply specify that no processor can be allocated more than one task at the same time. This translates into the following conditions:

$$\mathsf{alloc}(T) = \mathsf{alloc}(T') \Rightarrow \begin{cases} \sigma(T) + w(T) \leqslant \sigma(T') \\ \text{or} \quad \sigma(T') + w(T') \leqslant \sigma(T). \end{cases}$$

This condition expresses the fact that if two tasks T and T' are allocated to the same processor, then their executions cannot overlap in time.

Definition 3 *Let $G = (V, E, w)$ be a task graph.*

1. *Let σ be a schedule for G. Assume σ uses at most p processors (let $p = \infty$ if the processors are unlimited). The makespan $MS(\sigma, p)$ of σ is its total execution time:*

$$MS(\sigma, p) = \max_{v \in V} \{\sigma(v) + w(v)\} - \min_{v \in V} \{\sigma(v)\}.$$

2. *$Pb(p)$ is the problem of determining a schedule σ of minimal makespan $MS(\sigma, p)$ assuming p processors (let $p = \infty$ if the processors are unlimited). Let $MS_{opt}(p)$ be the value of the makespan of an optimal schedule with p processors:*

$$MS_{opt}(p) = \min_{\sigma} MS(\sigma, p).$$

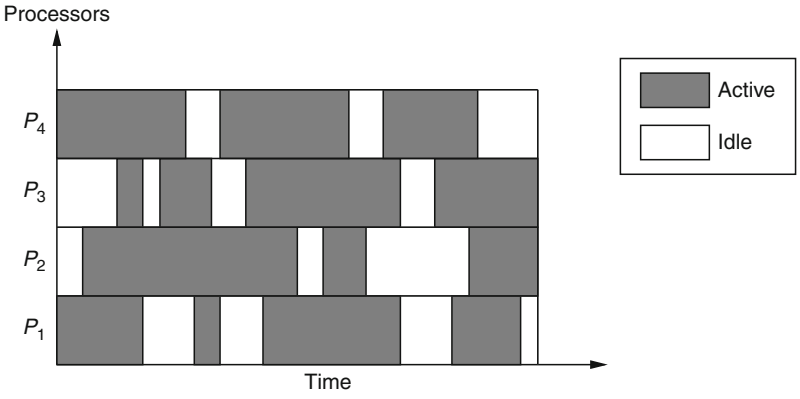
If the first task is scheduled at time 0, which is a common assumption, the expression of the makespan can be reduced to $\mathrm{MS}(\sigma, p) = \max_{v \in V} \{\sigma(v) + w(v)\}$. Weights extend to paths in G as usual; if $\Phi = (T_1, T_2, \ldots, T_n)$ denotes a path in G, then $w(\Phi) = \sum_{i=1}^{n} w(T_i)$. Because schedules respect dependences, the following easy bound on the makespan is readily obtained:

Proposition 1 *Let $G = (V, E, w)$ be a task graph and σ a schedule for G with p processors. Then, $MS(\sigma, p) \geqslant w(\Phi)$ for all paths Φ in G.*

The last definition introduces the notions of speedup and efficiency for schedules.

Definition 4 *Let $G = (V, E, w)$ be a task graph and σ a schedule for G with p processors:*

1. *The speedup is the ratio $s(\sigma, p) = \frac{Seq}{MS(\sigma, p)}$, where $Seq = \sum_{v \in V} w(v)$ is the sum of all task weights (Seq is the optimal execution time $MS_{opt}(1)$ of a schedule with a single processor).*
2. *The efficiency is the ratio $e(\sigma, p) = \frac{s(\sigma, p)}{p} = \frac{Seq}{p \times MS(\sigma, p)}$.*



Task Graph Scheduling. Fig. 2 Active and idle processors during execution

Theorem 1 *Let $G = (V, E, w)$ be a task graph. For any schedule σ with p processors,*

$$0 \leqslant e(\sigma, p) \leqslant 1.$$

Proof Consider the execution of σ as illustrated in Fig. 2 (this is a fictitious example, not related to the triangular system example). At any time during execution, some processors are active, and some are idle. At the end, all tasks have been processed. Let Idle denote the cumulated idle time of the p processors during the whole execution. Because Seq is the sum of all task weights, the quantity Seq + Idle is equal to the area of the rectangle in Fig. 2, that is, the product of the number of processors by the makespan of the schedule: $\text{Seq} + \text{Idle} = p \times \text{MS}(\sigma, p)$. Hence, $e(\sigma, p) = \frac{\text{Seq}}{p \times \text{MS}(\sigma, p)} \leqslant 1$. □

Solving Pb(∞)

Let $G = (V, E, w)$ be a given task graph and assume unlimited processors. Remember that a schedule σ for G is said to be *optimal* if its makespan $\text{MS}(\sigma, \infty)$ is minimal, that is, if $\text{MS}(\sigma, \infty) = \text{MS}_{opt}(\infty)$.

Definition 5 *Let $G = (V, E, w)$ be a task graph.*

1. *For $v \in V$, $PRED(v)$ denotes the set of all immediate predecessors of v, and $SUCC(v)$ the set of all its immediate successors.*
2. *$v \in V$ is an entry (top) vertex if and only if $PRED(v) = \emptyset$.*
3. *$v \in V$ is an exit (bottom) vertex if and only if $SUCC(v) = \emptyset$.*
4. *For $v \in V$, the top level $tl(v)$ is the largest weight of a path from an entry vertex to v, excluding the weight of v.*
5. *For $v \in V$, the bottom level $bl(v)$ is the largest weight of a path from v to an output vertex, including the weight of v.*

In the example of the triangular system, there is a single entry vertex, $T_{1,1}$, and a single exit vertex, $T_{n,n}$. The top level of $T_{1,1}$ is 0, and $tl(T_{1,2}) = tl(T_{1,1}) + w(T_{1,1}) = 1$. The value of $T_{2,3}$ is

$$tl(T_{2,3}) = \max\{w(T_{1,1}) + w(T_{1,2}) + w(T_{2,2}), w(T_{1,1}) + w(T_{1,3})\} = 3$$

because there are two paths from the entry vertex to $T_{2,3}$.

The top level of a vertex can be computed by a traversal of the DAG; the top level of an entry vertex is 0, while the top level of a non-entry vertex v is

$$tl(v) = \max\{tl(u) + w(u); u \in \text{PRED}(v)\}.$$

Similarly, $bl(v) = \max\{bl(u); u \in \text{SUCC}(v)\} + w(v)$ (and $bl(v) = w(v)$ for an exit vertex v). The top level of a vertex is the earliest possible time at which it can be executed, while its bottom level represents a lower bound of the remaining execution time once starting its execution. This can be stated more formally as follows.

Theorem 2 *Let $G = (V, E, w)$ be a task graph and define σ_{free} as follows:*

$$\forall v \in V,\ \sigma_{free}(v) = tl(v).$$

Then, σ_{free} is an optimal schedule for G.

From Theorem 2:

$$\mathrm{MS}_{opt}(\infty) = \mathrm{MS}(\sigma_{free}, \infty) = \max_{v \in V}\{tl(v) + w(v)\}.$$

Hence, $\mathrm{MS}_{opt}(\infty)$ is simply the maximal weight of a path in the graph. Note that σ_{free} is not the only optimal schedule.

Corollary 1 *Let $G = (V, E, w)$ be a directed acyclic graph. $Pb(\infty)$ can be solved in time $O(|V| + |E|)$.*

Going back to the triangular system (Fig. 1), because all tasks have weight 1, the weight of a path is equal to its length plus 1. The longest path is

$$T_{1,1} \to T_{1,2} \to T_{2,2} \to \cdots \to T_{n-1,n-1} \to T_{n,-1,n} \to T_{n,n},$$

whose weight is $2n-1$. Not as many processors as tasks are needed to achieve execution within $2n-1$ time units. For example, only $n-1$ processors can be used. Let $1 \leqslant i \leqslant n$; at time $2i-2$, processor P_1 starts the execution of task $T_{i,i}$, while at time $2i-1$, the first $n-i$ processors P_1, P_2, …, P_{n-i} execute tasks $T_{i,j}$, $i+1 \leqslant j \leqslant n$.

NP-completeness of Pb(p)

Definition 6 *The decision problem $Dec(p)$ associated with $Pb(p)$ is as follows. Given a task graph $G = (V, E, w)$, a number of processors $p \geqslant 1$, and an execution bound $K \in \mathbb{N}^*$, does there exist a schedule σ for G using at most p processors, such that $MS(\sigma, p) \leqslant K$? The restriction of $Dec(p)$ to independent tasks (no dependence, that is, when $E = \emptyset$) is denoted Indep-tasks(p). In both problems, p is arbitrary (it is part of the problem instance). When p is fixed a priori, say $p = 2$, problems are denoted as $Dec(2)$ and Indep-tasks(2).*

Well-known complexity results are summarized in the following theorem.

Theorem 3

- *Indep-tasks(2) is NP-complete but can be solved by a pseudo-polynomial algorithm. Moreover, $\forall\, \varepsilon > 0$, Indep-tasks$(2)$ admits a $(1+\varepsilon)$-approximation whose complexity is polynomial in $\frac{1}{\varepsilon}$.*
- *Indep-tasks(p) is NP-complete in the strong sense.*
- *$Dec(2)$ (and hence $Dec(p)$) is NP-complete in the strong sense.*

List Scheduling Heuristics

Because $\mathrm{Pb}(p)$ is NP-complete, heuristics are used to schedule task graphs with limited processors. The most natural idea is to use greedy strategies: At each instant, try to schedule as many tasks as possible onto available processors. Such strategies deciding *not to deliberately keep a processor idle* are called *list scheduling* algorithms. Of course, there are different possible strategies to decide which tasks are given priority in the (frequent) case where there are more free tasks than available processors. But a key result due to Graham [10] is that any list algorithm can be shown to achieve at most twice the optimal makespan.

Definition 7 *Let $G = (V, E, w)$ be a task graph and let σ be a schedule for G. A task $v \in V$ is free at time t (note $v \in FREE(\sigma, t)$) if and only if its execution has not yet started ($\sigma(v) \geqslant t$) but all its predecessors have been executed ($\forall\, u \in PRED(v),\ \sigma(u) + w(u) \leqslant t$).*

A list schedule is a schedule such that no processor is deliberately left idle; at each time t, if $|FREE(\sigma, t)| = r \geqslant 1$, and if q processors are available, then $\min(r, q)$ free tasks start executing.

Theorem 4 *Let $G = (V, E, w)$ be a task graph and assume there are p available processors. Let σ be any list schedule of G. Let $MS_{opt}(p)$ be the makespan of an optimal schedule. Then,*

$$MS(\sigma, p) \leqslant \left(2 - \frac{1}{p}\right) MS_{opt}(p).$$

It is important to point out that Theorem 4 holds for *any* list schedule, regardless of the strategy to choose among free tasks when there are more free tasks than available processors.

Lemma 1 *There exists a dependence path Φ in G whose weight $w(\Phi)$ satisfies*

$$Idle \leqslant (p-1) \times w(\Phi),$$

where Idle is the cumulated idle time of the p processors during the whole execution of the list schedule.

Proof Define the ancestors of a task are its predecessors, the predecessors of its predecessors, and so on. Let T_{i_1} be a task whose execution terminates at the end of the schedule:

$$\sigma(T_{i_1}) + w(T_{i_1}) = \mathrm{MS}(\sigma, p).$$

Let t_1 be the largest time smaller than $\sigma(T_{i_1})$ and such that there exists an idle processor during the time interval $[t_1, t_1 + 1]$ (let $t_1 = 0$ if such a time does not exist).

Why is this processor idle? Because σ is a list schedule, no task is free at t_1, otherwise the idle processor would start executing a free task. Therefore, there must be a task T_{i_2} that is an ancestor of T_{i_1} and that is being executed at time t_1; otherwise T_{i_1} would have been started at time t_1 by the idle processor. Because of the definition of t_1, it is known that all processors are active between the end of the execution of T_{i_2} and the beginning of the execution of T_{i_1}.

Then, start the construction again from T_{i_2} so as to obtain a task T_{i_3} such that all processors are active between the end of T_{i_3} and the beginning of T_{i_2}. Iterating the process, one ends up with r tasks $T_{i_r}, T_{i_{r-1}}, \ldots, T_{i_1}$ that belong to a dependence path Φ of G and such that all processors are active except perhaps during their execution. In other words, the idleness of some processors can only occur during the execution of these r tasks, during which at least one processor is active (the one that executes the task). Hence, Idle $\leqslant (p-1) \times \sum_{j=1}^{r} w(T_{i_j}) \leqslant (p-1) \times w(\Phi)$. □

Proof Going back to the proof of Theorem 4, recall that $p \times \mathrm{MS}(\sigma, p) = \mathrm{Idle} + \mathrm{Seq}$, where $\mathrm{Seq} = \sum_{v \in V} w(v)$ is the sequential time, that is, the sum of all task weights (see Fig. 2). Now take the dependence path Φ constructed in Lemma 1: $w(\Phi) \leqslant \mathrm{MS}_{opt}(p)$, because the makespan of any schedule is greater than the weight of all dependence paths in G (simply because dependence constraints are met). Furthermore, $\mathrm{Seq} \leqslant p \times \mathrm{MS}_{opt}(p)$ (with equality only if all p processors are active all the time). Putting this together:

$$\begin{aligned} p \times \mathrm{MS}(\sigma, p) &= \mathrm{Idle} + \mathrm{Seq} \leqslant (p-1)w(\Phi) + \mathrm{Seq} \\ &\leqslant (p-1)\mathrm{MS}_{opt}(p) + p\mathrm{MS}_{opt}(p) \\ &= (2p-1)\mathrm{MS}_{opt}(p), \end{aligned}$$

which proves the theorem. □

Fundamentally, Theorem 4 says that any list schedule is within 50% of the optimum. Therefore, list scheduling is guaranteed to achieve half the best possible performance, regardless of the strategy to choose among free tasks.

Proposition 2 *Let $MS_{list}(p)$ be the shortest possible makespan produced by a list scheduling algorithm. The bound*

$$MS_{list}(p) \leqslant \frac{2p-1}{p} MS_{opt}(p)$$

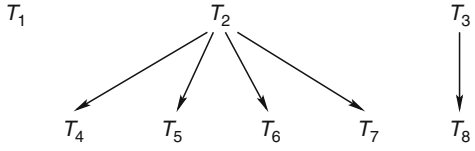
is tight.

Note that implementing a list scheduling algorithm is not difficult, but it is somewhat lengthy to describe in full detail; see Casanova et al. [3].

Critical Path Scheduling

A widely used list scheduling technique is *critical path scheduling*. The selection criterion for free tasks is based on the value of their bottom level. Intuitively, the larger the bottom level, the more "urgent" the task. The *critical path* of a task is defined as its bottom level and is used to assign priority levels to tasks. Critical path scheduling is list scheduling where the priority level of a task is given by the value of its critical path. Ties are broken arbitrarily.

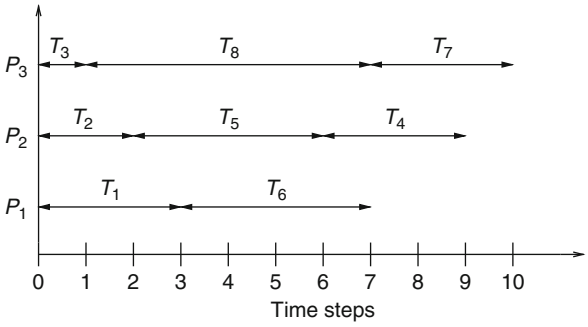
Consider the task graph shown in Fig. 3. There are eight tasks, whose weights and critical paths are listed in Table 1. Assume there are $p = 3$ available processors and let $\mathcal{Q}$ be the priority queue of free tasks. At $t = 0$, $\mathcal{Q}$ is initialized as $\mathcal{Q} = (T_3, T_2, T_1)$. These three tasks are executed. At $t = 1$, T_8 is added to the queue: $\mathcal{Q} = (T_8)$. There is one processor available, which starts the execution of T_8. At $t = 2$, the four successors of T_2 are added to the queue: $\mathcal{Q} = (T_5, T_6, T_4, T_7)$. Note that ties have been broken arbitrarily (using task indices in this case). The available processor picks the first task T_5 in $\mathcal{Q}$. Following this scheme, the execution goes on up to $t = 10$, as summarized in Fig. 4.



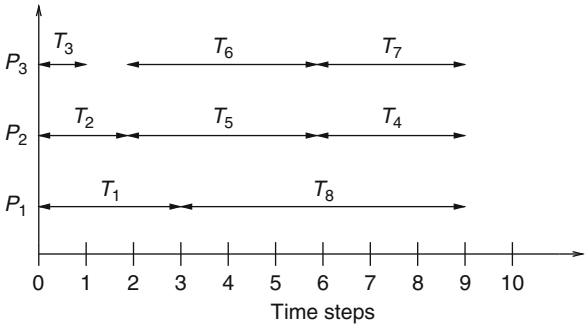
Task Graph Scheduling. Fig. 3 A small example

Task Graph Scheduling. Table 1 Weights and critical paths for the task graph in Fig. 3

Tasks	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Weights	3	2	1	3	4	4	3	6
Critical paths	3	6	7	3	4	4	3	6



Task Graph Scheduling. Fig. 4 Critical path schedule for the example in Fig. 3



Task Graph Scheduling. Fig. 5 Optimal schedule for the example in Fig. 3

Note that it is possible to schedule the graph in only 9 time units, as shown in Fig. 5. The trick is to leave a processor idle at time $t = 1$ deliberately; although it has the highest critical path, T_8 can be delayed by two time units. T_5 and T_6 are given preference to achieve a better load balance between processors. The schedule shown in Fig. 5 is optimal, because Seq = 26, so that three processors require at least $\lceil \frac{26}{3} \rceil = 9$ time units. This small example illustrates the difficulty of scheduling with a limited number of processors.

Taking Communication Costs into Account

The Macro-Dataflow Model

Thirty years ago, communication costs have been introduced in the scheduling literature. Because the performance of network communication is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms, the vast majority of results hold for a very simple model, which is as follows.

The target platform consists of p identical processors that are part of of a fully connected clique. All interconnection links have same bandwidth. If a task T communicates data to a successor task T', the cost is modeled as

$$\text{cost}(T, T') = \begin{cases} 0 & \text{if } \mathsf{alloc}(T) = \mathsf{alloc}(T') \\ c(T, T') & \text{otherwise,} \end{cases}$$

where $\mathsf{alloc}(T)$ denotes the processor that executes task T, and $c(T, T')$ is defined by the application specification. The time for communication between two tasks running on the same processor is negligible. This so-called *macro-dataflow* model makes two main assumptions: (i) communication can occur as soon as data are available and (ii) there is no contention for network links. Assumption (i) is reasonable as communication can overlap with (independent) computations in most modern computers. Assumption (ii) is much more questionable. Indeed, there is no physical device capable of sending, say, 1,000 messages to 1,000 distinct processors, at the same speed as if there were a single message. In the worst case, it would take 1,000 times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources. Nevertheless, this assumption is omnipresent in the traditional scheduling literature.

Definition 8 *A communication task graph (or commTG) is a direct acyclic graph $G = (V, E, w, c)$, where vertices represent tasks and edges represent precedence constraints. The computation weight function is $w : V \longrightarrow \mathbb{N}^*$ and the communication cost function is $c : E \longrightarrow \mathbb{N}^*$. A schedule σ must preserve dependences, which is written as*

$$\forall e = (T, T') \in E, \begin{cases} \sigma(T) + w(T) \leqslant \sigma(T') \\ \qquad \textit{if } \mathsf{alloc}(T) = \mathsf{alloc}(T') \\ \sigma(T) + w(T) + c(T, T') \leqslant \sigma(T') \\ \qquad \textit{otherwise.} \end{cases}$$

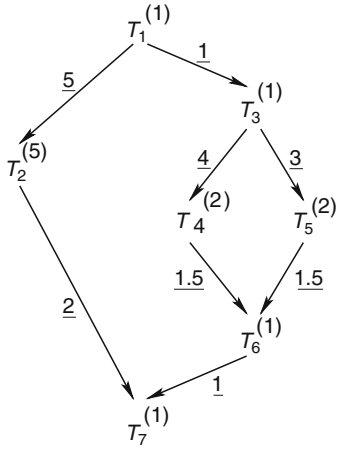
The expression of resource constraints is the same as in the no-communication case.

Complexity and List Heuristics with Communications

Including communication costs in the model makes everything difficult, including solving $\mathrm{Pb}(\infty)$. The intuitive reason is that a trade-off must be found between allocating tasks to either many processors (hence balancing the load but communicating intensively) or few processors (leading to less communication but less parallelism as well). Here is a small example, borrowed from [9].

Consider the commTG in Fig. 6. Task weights are indicated close to the tasks within parentheses, and communication costs are shown along the edges, underlined. For the sake of this example, two non-integer communication costs are used: $c(T_4, T_6) = c(T_5, T_6) = 1.5$. Of course, every weight w and cost c could be scaled to have only integer values. Observe the following:

- On the one hand, if all tasks are assigned to the same processor, the makespan will be equal to the sum of all task weights, that is, 13.
- On the other hand, with unlimited processors (no more than seven processors are needed because there are seven tasks), each task can be assigned to a different processor. Then, the makespan of the ASAP schedule is equal to 14. To see this, it is important to point out that once the allocation of tasks to processors is given, the makespan is computed easily: For each edge $e : T \to T'$, add a virtual node of weight $c(T, T')$ if the edge links two different processors ($\mathsf{alloc}(T) \neq \mathsf{alloc}(T')$), and do nothing otherwise. Then, consider the new graph as a DAG (without communications) and traverse it to compute the length of the longest path. Here, because all tasks are allocated to different processors, a virtual node is added on each edge. The longest path is $T_1 \to T_2 \to T_7$, whose length is $w(T_1) + c(T_1, T_2) + w(T_2) + c(T_2, T_7) + w(T_7) = 14$.

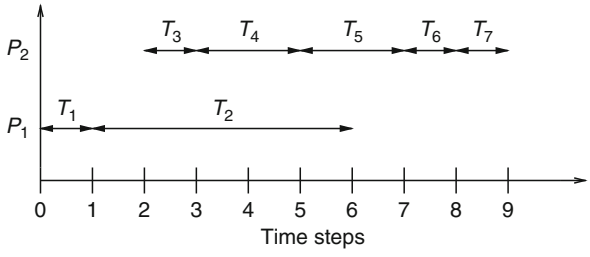


Task Graph Scheduling. Fig. 6 An example commTG

There is a difficult trade-off between executing tasks in parallel (hence with several distinct processors) and minimizing communication costs. In the example, it turns out that the best solution is to use two processors, according to the schedule in Fig. 7, whose makespan is equal to 9. Using more processors does not always lead to a shorter execution time. Note that dependence constraints are satisfied in Fig. 7. For example, T_2 can start at time 1 on processor P_1 because this processor executes T_1, hence there is no need to pay the communication cost $c(T_1, T_2)$. By contrast, T_3 is executed on processor P_2, hence it cannot be started before time 2 even though P_2 is idle: $\sigma(T_1) + w(T_1) + c(T_1, T_3) = 0 + 1 + 1 = 2$.

With unlimited processors, the optimization problem becomes difficult: $\mathrm{Pb}(\infty)$ is NP-complete in the strong sense. Even the problem in which all task weights and communication costs have the same (unit) value, the so-called UET-UCT problem (unit execution time-unit communication time), is NP-hard [13].

With limited processors, list heuristics can be extended to take communication costs into account, but Graham's bound does not hold any longer. For instance, the *Modified Critical Path (MCP)* algorithm proceeds as follows. First, bottom levels are computed using a pessimistic evaluation of the longest path, accounting for each potential communication (this corresponds to the allocation where there is a different processor per task).



Task Graph Scheduling. Fig. 7 An optimal schedule for the example

These bottom levels are used to determine the priority of free tasks. Then each free task is assigned to the processor that allows its earliest execution, given previous task allocation decisions. It is important to explain further what "previous task allocation decisions" means. Free tasks from the queue are processed one after the other. At any moment, it is known which processors are available and which ones are busy. Moreover, for the busy processors, it is known when they will finish computing their currently allocated tasks. Hence, it is always possible to select the processor that can begin executing the task soonest. It may well be the case that a currently busy processor is selected.

Extension to Heterogeneous Platforms

This section explains how to extend list scheduling techniques to heterogeneous platforms, that is, to platforms that consist of processors with different speeds and interconnection links with different bandwidths. Key differences with the homogeneous case are outlined.

Given a commTG with n tasks $T_1, \ldots, T_n$, the goal is to schedule it on a platform with p heterogeneous processors $P_1, \ldots, P_p$. There are now many parameters to instantiate:

Computation costs : The execution cost of T_i on P_q is modeled as w_{iq}. Therefore, an $n \times p$ matrix of values is needed to specify all computation costs. This matrix comes directly for the specific scheduling problem at hand. However, when attempting to evaluate competing scheduling heuristics over a large number of synthetic scenarios, one must generate this matrix. One can distinguish two approaches. In the first approach one generates a *consistent* (or *uniform*) matrix with $w_{iq} = w_i \times \gamma_q$, where w_i represents the number of operations required by T_i and γ_q is the inverse of the speed of P_q (in operations per second). With this definition the relative speed of the processors does not depend on the particular task they execute. If instead some processors are faster for some tasks than some other processors, but slower for other tasks, one speaks of an *inconsistent* (or *nonuniform*) matrix. This corresponds to the case in which some processors are specialized for some tasks (e.g., specialized hardware or software).

Communication costs : Just as processors have different speeds, communication links may have different bandwidths. However, while the speed of a processor may depend upon the nature of the computation it performs, the bandwidth of a link does not depend on the nature of the bytes it transmits. It is therefore natural to assume *consistent* (or *uniform*) links. If there is a dependence $e_{ij} : T_i \rightarrow T_j$, if T_i is executed on P_q and T_j executed on P_r, then the communication time is modeled as

$$\text{comm}(i,j,q,r) = \text{data}(i,j) \times v_{qr},$$

where $\text{data}(i,j)$ is the data volume associated to e_{ij} and v_{qr} is the communication time for a unit-size message from P_q to P_r (i.e., the inverse of the bandwidth). Like in the homogeneous case, let $v_{qr} = 0$ if $q = r$, that is, if both tasks are assigned the same processor. If one wishes to generate synthetic scenarios to evaluate competing scheduling heuristics, one then must generate two matrices: one of size $n \times n$ for data and one of size $p \times p$ for v_{qr}.

The main list scheduling principle is unchanged. As before, the priority of each task needs to be computed, so as to decide which one to execute first when there are more free tasks than available processors. The most natural idea is to compute averages of computation and communication times, and use these to compute priority levels exactly as in the homogeneous case:

- $\overline{w_i} = \frac{\sum_{q=1}^{p} w_{iq}}{p}$, the *average* execution time of T_i.
- $\overline{\text{comm}_{ij}} = \text{data}(i,j) \times \frac{\sum_{1 \leq q,r \leq p, q \neq r} v_{qr}}{p(p-1)}$, the *average* communication cost for edge $e_{ij} : T_i \rightarrow T_j$.

The last (but important) modification concerns the way in which tasks are assigned to processors: Instead of assigning the current task to the processor that will *start* its execution first (given all already taken decisions), one should assign it to the processor that will *complete* its execution first (given all already taken decisions). Both choices are equivalent with homogeneous processors, but intuitively the latter is likely to be more efficient in the heterogeneous case. Altogether, this leads to the list heuristic called HEFT, for *Heterogeneous Earliest Finish Time* [19].

Workflow Scheduling

This section discusses *workflow scheduling*, that is, the problem of scheduling a (large) collection of identical task graphs rather than a single one. The main idea is to pipeline the execution of successive instances. Think

of a sequence of video images that must be processed in a pipelined fashion: Each image enters the platform and follows the same processing chain, and a new image can enter the system while previous ones are still being executed. This section is intended to give a flavor of the optimization problems to be solved in such a context. It restricts to simpler problem instances.

Consider "chains," that is, applications structured as a sequence of stages. Each stage corresponds to a different computational task. The application must process a large number of data sets, each of which must go through all stages. Each stage has its own communication and computation requirements: It reads an input from the previous stage, processes the data, and outputs a result to the next stage. Initial data are input to the first stage and final results are obtained as the output from the last stage. The pipeline operates in synchronous mode: After some initialization delay, a new task is completed every period. The period is defined as the longest "cycle-time" to operate a stage, and it is the inverse of the throughput that can be achieved.

For simplicity, it is assumed that each stage is assigned to a single processor, that is in charge of processing all instances (all data sets) for that stage. Each pipeline stage can be viewed a sequential task that may write some global data structure, to disk or to memory, for each processed data set. In this case, tasks must always be processed in a sequential order within a stage. Moreover, due to possible local updates, each stage must be mapped onto a single processor. For a given stage, one cannot process half of the tasks on one processor and the remaining half on another without maintaining global information, which might be costly and difficult to implement. In other words, a processor that is assigned a stage will execute the operations required by this stage (input, computation, and output) for all the tasks fed into the pipeline.

Of course, other assumptions are possible: some stages could be replicated, or even data-parallelized. The reader is referred the bibliographical notes at the end of the chapter for such extensions.

Objective Functions

An important metric for parallel applications that consists of many individual computations is the *throughput*. The throughput measures the aggregate rate of data processing; it is the rate at which data sets can enter the system. Equivalently, the inverse of the throughput, defined as the *period*, is the time interval required between the beginning of the execution of two consecutive data sets. The period minimization problem can be stated informally as follows: Which stage to assign to which processor so that the largest period of a processor is kept minimal?

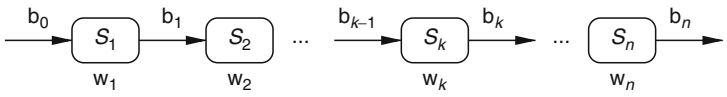
Another important metric is derived from makespan minimization, but it must be adapted. With a large number of data sets, the total execution time is less relevant, but the execution time for each data set remains important, in particular for real-time applications. One talks of *latency* rather than of makespan, in order to avoid confusion. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely.

Minimizing the latency is antagonistic to maximizing the throughput. In fact, assigning all application stages to the fastest processor (thus working in a fully sequential way) would suppress all communications and accelerate computations, thereby minimizing the latency, but achieving a very bad throughput. Conversely, mapping each stage to a different processor is likely to decrease the period, hence increase the throughput (work in a fully pipelined manner), but the resulting latency will be high, because all interstage communications must be accounted for in this latter mapping. Trade-offs will have to be found between these criteria.

How to deal with several objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But it is not natural for the user to maximize a quantity like $0.7T + 0.3L$, where T is the throughput and L the latency. Instead, one is more likely to fix a throughput T, and to search for the best latency that can be achieved while enforcing T? One single criterion is optimized, under the condition that a threshold is enforced for the other one.

Period and Latency

Consider a pipeline with n stages $\mathcal{S}_k$, $1 \leqslant k \leqslant n$, as illustrated in Fig. 8. Tasks are fed into the pipeline and processed from stage to stage, until they exit the pipeline after the last stage. The k-th stage $\mathcal{S}_k$ receives an input from the previous stage, of size b_{k-1}, performs a number of w_k operations, and outputs data of size b_k to the



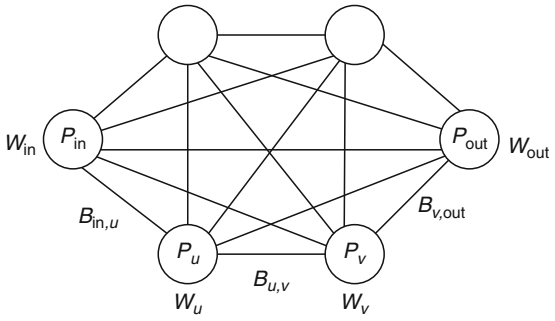
Task Graph Scheduling. Fig. 8 The application pipeline

next stage. The first stage $\mathcal{S}_1$ receives an initial input of size b_0, while the last stage $\mathcal{S}_n$ returns a final result of size b_n.

The target platform is a clique with p processors P_u, $1 \leqslant u \leqslant p$, that are fully interconnected (see Fig. 9). There is a bidirectional link $\mathsf{link}_{u,v} : P_u \leftrightarrow P_v$ with bandwidth $B_{u,v}$ between each processor P_u and P_v, The literature often enforces more realistic communication models for workflow scheduling than for Task Graph Scheduling. For the sake of simplicity, a very strict model is enforced here: A given processor can be involved in a single communication at any time unit, either a send or a receive. Note that independent communications between distinct processor pairs can take place simultaneously. Finally, there is no overlap between communications and computations, so that all the operations of a given processor are fully sequentialized. The speed of processor P_u is denoted as W_u, and it takes X/W_u time units for P_u to execute X operations. It takes $X/B_{u,v}$ time units to send (respectively, receive) a message of size X to (respectively, from) P_v.

The mapping problem consists in assigning application stages to processors. For *one-to-one mappings*, it is required that each stage $\mathcal{S}_k$ of the application pipeline be mapped onto a distinct processor $P_{\mathsf{alloc}(k)}$ (which is possible only if $n \leqslant p$). The function alloc associates a processor index to each stage index. For convenience, two fictitious stages $\mathcal{S}_0$ and $\mathcal{S}_{n+1}$ are created, assigning $\mathcal{S}_0$ to P_{in} and $\mathcal{S}_{n+1}$ to P_{out}.

What is the period of $P_{\mathsf{alloc}(k)}$, that is, the minimum delay between the processing of two consecutive tasks? To answer this question, one needs to know to which processors the previous and next stages are assigned. Let $t = \mathsf{alloc}(k-1)$, $u = \mathsf{alloc}(k)$, and $v = \mathsf{alloc}(k+1)$. P_u needs $\mathsf{b}_{k-1}/B_{t,u}$ time units to receive the input data from P_t, w_k/W_u time units to process it, and $\mathsf{b}_k/B_{u,v}$ time units to send the result to P_v, hence a cycle-time of $\mathsf{b}_{k-1}/B_{t,u} + \mathsf{w}_k/W_u + \mathsf{b}_k/B_{u,v}$ time units for P_u. These three steps are serialized (see Fig. 10 for an illustration). The *period* achieved with the mapping is the maximum of the cycle-times of the processors, which corresponds to the rate at which the pipeline can be activated.



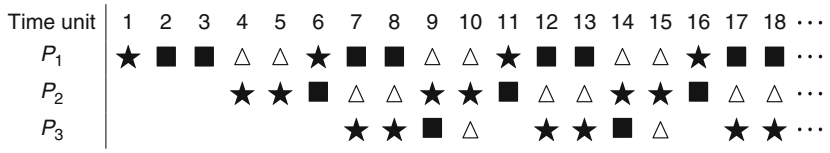
Task Graph Scheduling. Fig. 9 The target platform

In this simple instance, the optimization problem can be stated as follows: Determine a one-to-one allocation function $\mathsf{alloc} : [1,n] \to [1,p]$ (augmented with $\mathsf{alloc}(0) = \mathsf{in}$ and $\mathsf{alloc}(n+1) = \mathsf{out}$) such that

$$T_{\mathsf{period}} = \max_{1 \leqslant k \leqslant n} \left\{ \frac{\mathsf{b}_{k-1}}{B_{\mathsf{alloc}(k-1),\mathsf{alloc}(k)}} + \frac{\mathsf{w}_k}{W_{\mathsf{alloc}(k)}} + \frac{\mathsf{b}_k}{B_{\mathsf{alloc}(k),\mathsf{alloc}(k+1)}} \right\}$$

is minimized.

Natural extensions are *interval mappings*, in which each participating processor is assigned an interval of consecutive stages. Note that when $p < n$ interval mappings are mandatory. Intuitively, assigning several consecutive tasks to the same processor will increase its computational load, but will also decrease communication. The best interval mapping may turn out to be a one-to-one mapping, or instead may utilize only a very small number of fast computing processors interconnected by high-speed links. The optimization problem associated to interval mappings is formally expressed as follows. The intervals achieve a partition of the original set of stages $\mathcal{S}_1$ to $\mathcal{S}_n$. One searches for a partition of $[1,\ldots,n]$ into m intervals $I_j = [d_j, e_j]$ such that $d_j \leqslant e_j$ for $1 \leqslant j \leqslant m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leqslant j \leqslant m-1$, and $e_m = n$. Recall that the function $\mathsf{alloc} : [1,n] \to [1,p]$ associates a processor index to each stage index. In a one-to-one mapping, this function was a one-to-one



Task Graph Scheduling. Fig. 10 An example of one-to-one mapping with three stages and processors. Each processor periodically receives input data from its predecessor (★), performs some computation (■), and outputs data to its successor (△). Note that these operations are shifted in time from one processor to another. The cycle-time of P_1 and P_2 is 5 while that of P_3 is 4, hence $T_{\text{period}} = 5$

assignment. In an interval mapping, for $1 \leqslant j \leqslant m$, the whole interval I_j is mapped onto the same processor $P_{\text{alloc}(d_j)}$, that is, for $d_j \leqslant i \leqslant e_j$, $\text{alloc}(i) = \text{alloc}(d_j)$. Also, two intervals cannot be mapped to the same processor, that is, for $1 \leqslant j, j' \leqslant m$, $j \neq j'$, $\text{alloc}(d_j) \neq \text{alloc}(d_{j'})$. The period is expressed as

$$T_{\text{period}} = \max_{1 \leqslant j \leqslant m} \left\{ \frac{\mathsf{b}_{d_j - 1}}{B_{\text{alloc}(d_j - 1), \text{alloc}(d_j)}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{W_{\text{alloc}(d_j)}} + \frac{\mathsf{b}_{e_j}}{B_{\text{alloc}(d_j), \text{alloc}(e_j + 1)}} \right\}$$

Note that $\text{alloc}(d_j - 1) = \text{alloc}(e_{j-1}) = \text{alloc}(d_{j-1})$ for $j > 1$ and $d_1 - 1 = 0$. Also, $e_j + 1 = d_{j+1}$ for $j < m$, and $e_m + 1 = n + 1$. It is still assumed that $\text{alloc}(0) = \mathsf{in}$ and $\text{alloc}(n+1) = \mathsf{out}$. The optimization problem is then to determine the mapping that minimizes T_{period}, over all possible partitions into intervals, and over all mappings of these intervals to the processors.

The latency of an interval mapping is computed as follows. Each data set traverses all stages, but only communications between two stages mapped on the same processors take zero time units. Overall, the latency is expressed as

$$T_{\text{latency}} = \sum_{1 \leqslant j \leqslant m} \left\{ \frac{\mathsf{b}_{d_j - 1}}{B_{\text{alloc}(d_j - 1), \text{alloc}(d_j)}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{W_{\text{alloc}(d_j)}} \right\} + \frac{\mathsf{b}_n}{B_{\text{alloc}(n), \text{alloc}(n+1)}}$$

The latency for a one-to-one mapping obeys the same formula (with the restriction that each interval has length 1). Just as for the period, there are two minimization problems for the latency, with one-to-one and interval mappings.

It goes beyond the scope of this entry to assess the complexity of these period/latency optimization problems, and of their bi-criteria counterparts. The aim was to provide the reader with a quick primer on workflow scheduling, an activity that borrows several concepts from Task Graph Scheduling, while using more realistic platform models, and different objective functions.

Related Entries

►Loop Pipelining
►Modulo Scheduling and Loop Pipelining
►Scheduling Algorithms

Recommended Reading

Without communication costs, pioneering work includes the book by Coffman [5]. The book by El-Rewini et al. [7] and the IEEE compilation of papers [15] provide additional material. On the theoretical side, Appendix A5 of Garey and Johnson [8] provides a list of NP-complete scheduling problems. Also, the book by Brucker [2] offers a comprehensive overview of many complexity results.

The literature with communication costs is more recent. See the survey paper by Chrétienne and Picouleau [4]. See also the book by Darte et al. [6], where many heuristics are surveyed. The book by Sinnen [16] provides a thorough discussion on communication models. In particular, it describes several extensions for modeling and accounting for communication contention.

Workflow scheduling is quite a hot topic with the advent of large-scale computing platforms. A few representative papers are [1, 11, 17, 18].

Modern scheduling encompasses a wide spectrum of techniques: divisible load scheduling, cyclic scheduling, steady-state scheduling, online scheduling, job scheduling, and so on. A comprehensive survey is available in the book [14]. See also the handbook [12].

Most of the material presented in this entry is excerpted from the book by Casanova et al. [3].

Bibliography

1. Benoit A, Robert Y (2008) Mapping pipeline skeletons onto heterogeneous platforms. J Parallel Distr Comput 68(6): 790–808
2. Brucker P (2004) Scheduling algorithms. Springer, New York
3. Casanova H, Legrand A, Robert Y (2008) Parallel algorithms. Chapman & Hall/CRC Press, Beaumont, TX
4. Chrétienne P, Picouleau C (1995) Scheduling with communication delays: a survey. In: Chrétienne P, Coffman EG Jr, Lenstra JK, Liu Z (eds) Scheduling theory and its applications. Wiley, Hoboken, NJ, pp 65–89
5. Coffman EG (1976) Computer and job-shop scheduling theory. Wiley, Hoboken, NJ
6. Darte A, Robert Y, Vivien F (2000) Scheduling and automatic parallelization. Birkhaüser, Boston
7. El-Rewini H, Lewis TG, Ali HH (1994) Task scheduling in parallel and distributed systems. Prentice Hall, Englewood Cliffs
8. Garey MR, Johnson DS (1991) Computers and intractability, a guide to the theory of NP-completeness. WH Freeman and Company, New York
9. Gerasoulis A, Yang T (1992) A comparison of clustering heuristics for scheduling DAGs on multiprocessors. J Parallel Distr Comput 16(4):276–291
10. Graham RL (1996) Bounds for certain multiprocessor anomalies. Bell Syst Tech J 45:1563–1581
11. Hary SL, Ozguner F (1999) Precedence-constrained task allocation onto point-to-point networks for pipelined execution. IEEE Trans Parallel Distr Syst 10(8):838–851
12. Leung JY-T (ed) (2004) Handbook of scheduling: algorithms, models, and performance analysis. Chapman and Hall/CRC Press, Boca Raton
13. Picouleau C (1995) Task scheduling with interprocessor communication delays. Discrete App Math 60(1–3):331–342
14. Robert Y, Vivien F (eds) (2009) Introduction to scheduling. Chapman and Hall/CRC Press, Boca Raton
15. Shirazi BA, Hurson AR, Kavi KM (1995) Scheduling and load balancing in parallel and distributed systems. IEEE Computer Science Press, San Diego
16. Sinnen O (2007) Task scheduling for parallel systems. Wiley, Hoboken
17. Spencer M, Ferreira R, Beynon M, Kurc T, Catalyurek U, Sussman A, Saltz J (2002) Executing multiple pipelined data analysis operations in the grid. Proceedings of the ACM/IEEE supercomputing conference. ACM Press, Los Alamitos
18. Subhlok J, Vondran G (1995) Optimal mapping of sequences of data parallel tasks. Proceedings of the 5th ACM SIGPLAN symposium on principles and practice of parallel programming. ACM Press, San Diego, pp 134–143
19. Topcuoglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distr Syst 13(3):260–274

Task Mapping, Topology Aware

►Topology Aware Task Mapping

Tasks

►Processes, Tasks, and Threads

TAU

Sameer Shende, Allen D. Malony, Alan Morris, Wyatt Spear, Scott Biersdorff
University of Oregon, Eugene, OR, USA

Synonyms

TAU performance system®; Tuning and analysis utilities

Definition

The TAU Performance System® is an integrated suite of tools for instrumentation, measurement, and analysis of parallel programs with particular focus on large-scale, high-performance computing (HPC) platforms. TAU's objectives are to provide a flexible and interoperable framework for performance tool research and development, and robust, portable, and scalable set of technologies for performance evaluation on high-end computer systems.

Discussion

Introduction

Scalable parallel systems have always evolved together with the tools used to observe, understand, and optimize their performance. Next-generation parallel computing environments are guided to a significant degree

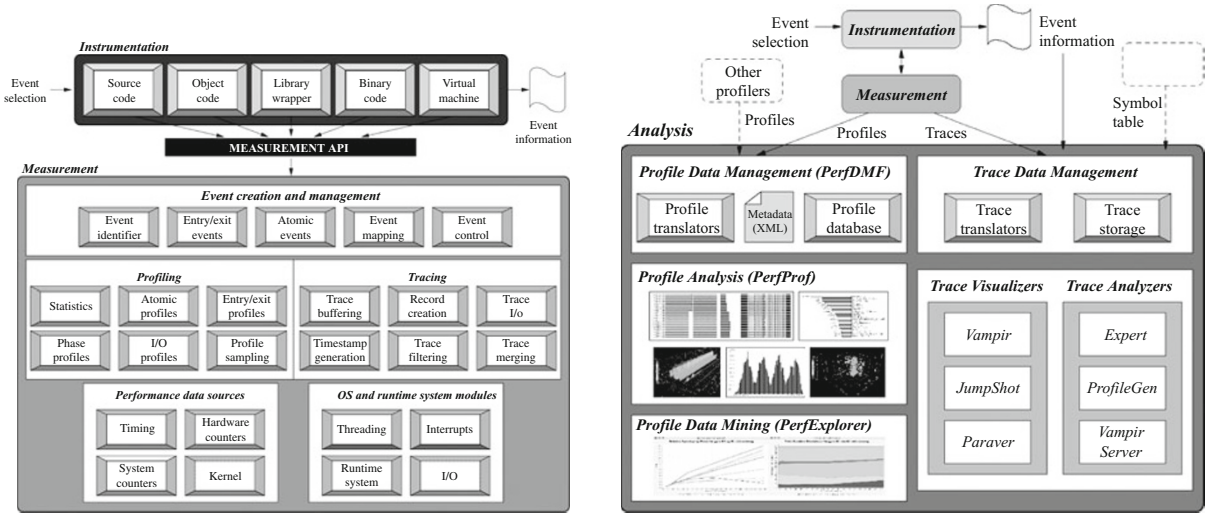
by what is known about application performance on current machines and how performance factors might be influenced by technological innovations. State-of-the-art performance tools play an important role in helping to understand application performance, diagnose performance problems, and guide tuning decisions on modern parallel platforms. However, performance tool technology must also respond to the growing complexity of next-generation parallel systems in order to help deliver the promises of high-end computing (HEC).

The TAU project began in the early 1990s with the goal of creating a performance instrumentation, measurement, and analysis framework that could produce robust, portable, and scalable performance tools for use in all parallel programs and systems over several technology generations. Today, the TAU Performance System® is a ubiquitous performance tools suite for shared-memory and message-passing parallel applications written in multiple programming languages (e.g., C, C++, Fortran, OpenMP, Java, Python, UPC, Chapel) that can scale to the largest parallel machines available.

TAU Design

TAU is of a class of performance systems based on the approach of *direct performance observation*, wherein execution *actions* of performance interest are exposed as *events* to the performance system through direct insertion of instrumentation in the application, library, or system code, at locations where the actions arise. In general, the actions reflect an occurrence of some execution state, most commonly as a result of a code location being reached (e.g., entry in a subroutine). However, it could also include a change in data. The key point is that the observation mechanism is direct. Generated events are made visible to the performance system in this way and contain implicit meta information as to their associated action. Thus, for any *performance experiment* using direct observation, the performance events of interest must be decided and necessary instrumentation done for their generation. Performance measurements are made of the events during execution and saved for analysis. Knowledge of the events is used to process the performance data and interpret the analysis results.

The TAU framework architecture, shown in Fig. 1, separates the functional concerns of a direct performance observation approach into three primary layers – *instrumentation*, *measurement*, and *analysis*. Each layer uses multiple modules which can be configured in a flexible manner under user control. This design makes it possible for TAU to target alternative models of parallel computation, from shared-memory multi-threading to distributed memory message passing to mixed-mode parallelism [17]. TAU defines an abstract computation model for parallel systems that captures general architecture and software execution features and can be mapped to existing complex system types [16].



TAU. Fig. 1 TAU architecture: instrumentation and measurement (*left*), analysis (*right*)

TAU's design has proven to be robust, sound, and highly adaptable to generations of parallel systems. The framework architecture has allowed new components to be added that have extended the capabilities of the TAU toolkit. This is especially true in areas concerning kernel-level performance integration [11, 14], performance monitoring [10, 12, 13], performance data mining [6], and GPU performance measurement [8].

TAU Instrumentation

The role of the instrumentation layer in direct performance observation is to insert code (a.k.a. *probes*) to make performance events visible to the measurement layer. Performance events can be defined and instrumentation inserted in a program at several levels of the program transformation process. In fact, it is important to realize that a complete performance view may require contribution of event information across code levels [15]. For these reasons, TAU supports several instrumentation mechanisms based on the code type and transformation level: source (manual, preprocessor, library interposition), binary/dynamic, interpreter, and virtual machine. There are multiple factors that affect the choice of what level to instrument, including accessibility, flexibility, portability, concern for intrusion, and functionality. It is not a question of what level is "correct" because there are trade-offs for each and different events are visible at different levels. TAU is distinguished by its broad support for different instrumentation methods and their use together.

TAU supports two general classes of events for instrumentation using any method: *atomic* events and *interval* events. An atomic event denotes a single action. Instrumentation is inserted at a point in the program code to expose an atomic action, and the measurement system obtains performance data associated with the action where and when it occurs. An interval event is a pair of events: *begin* and *end*. Instrumentation is inserted at two points in the code, and the measurement system uses data obtained from each event to determine performance for the interval between them (e.g., the time spent in a subroutine from entry (beginning of the interval) to exit (end of the interval)). In addition to the two general events classes, TAU allows events to be selectively enabled / disabled for any instrumentation method.

TAU Measurement

The TAU performance measurement system is a highly robust, scalable infrastructure portable to all HPC platforms. As shown in Fig. 1, TAU supports the two dominant methods of measurement for direct performance observation – parallel *profiling* and *tracing* – with rich access to performance data through portable timing facilities, integration with hardware performance counters, and user-level information. The choice of measurement method and performance data is made independently of the instrumentation decisions. This allows multiple performance experiments to be conducted to gain different performance views for the same set of events. TAU also provides unique support for novel performance mapping [15], runtime monitoring [10, 12, 13], and kernel-level measurements [11, 14].

TAU's measurement system has two core capabilities. First, the *event management* handles the registration and encoding of events as they are created. New events are represented in an *event table* by instantiating a new *event record*, recording the *event name*, and linking in storage allocated for the event performance data. The event table is used for all atomic and interval events regardless of their complexity. Event type and context information are encoded in the event names. The TAU event-management system hashes and maps these names to determine if an event has already occurred or needs to be created. Events are managed for every thread of execution in the application. Second, a runtime representation, called the *event callstack*, captures the nesting relationship of interval performance events. It is a powerful runtime measurement abstraction for managing the TAU performance state for use in both profiling and tracing. In particular, the event callstack is key for managing execution context, allowing TAU to associate this context to the events being measured.

Parallel profiling in TAU characterizes the behavior of every application thread in terms of its aggregate performance metrics. For interval events, TAU computes *exclusive* and *inclusive* metrics for each event. The TAU profiling system supports several profiling variants. The standard type of profiling is called *flat profiling*, which shows the exclusive and inclusive performance of each event but provides no other performance information about events occurring when an interval is active (i.e., nested events). In contrast, TAU's *event path profiling* can capture performance data with respect to event nesting

relationships. It is also interesting to observe performance data relative to an execution *state*. The structural, logical, and numerical aspects of a computation can be thought of as representing different execution *phases*. TAU supports an interface to create (*phase events*) and to mark their entry and exit. Internally in the TAU measurement system, when a phase, *P*, is entered, all subsequent performance will be measured with respect to *P* until it exits. When phase profiles are recorded, a separate parallel profile is generated for each phase.

TAU implements robust, portable, and scalable parallel tracing support to log events in time-ordered tuples containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event, event-specific information, and other performance-related data (e.g., hardware counters). All performance events are available for tracing. TAU will produce a trace for every thread of execution in its modern trace format as well as in OTF [7] and EPILOG [9] formats. TAU also provides mechanisms for online and hierarchical trace merging [3].

TAU Analysis

As the complexity of measuring parallel performance increases, the burden falls on analysis and visualization tools to interpret the performance information. As shown in Fig. 1, TAU includes sophisticated tools for parallel profile analysis and performance data mining. In addition, TAU leverages advanced trace analysis technologies from the performance tool community, primarily the Vampir [2] and Scalasca [18] tools. The following focuses on the features of the TAU profiling tools.

TAU's parallel profile analysis environment consists of a framework for managing parallel profile data, *PerfDMF* [5], and TAU's parallel profile analysis tool, *ParaProf* [1]. The complete environment is implemented entirely in Java. The performance data management framework (PerfDMF) in TAU provides a common foundation for parsing, storing, and querying parallel profiles from multiple performance experiments. It builds on robust SQL relational database engines and must be able to handle both large-scale performance profiles, consisting of many events and threads of execution, as well as many profiles from multiple performance experiments. To facilitate performance analysis development, the PerfDMF architecture includes a well-documented data-management API to abstract query and analysis operation into a more programmatic, non-SQL form.

TAU's parallel profile analysis tool, *ParaProf* [1], is capable of processing the richness of parallel profile information produced by the measurement system, both in terms of the profile types (flat, callpath, phase, snapshots) as well as scale. ParaProf provides the users with a highly graphical tool for viewing parallel profile data with respect to different viewing scopes and presentation methods. Profile data can be input directly from a PerfDMF database and multiple profiles can be analyzed simultaneously. ParaProf can show parallel profile information in the form of bargraphs, callgraphs, scalable histograms, and cumulative plots. ParaProf is also capable of integrating multiple performance profiles for the same performance experiment but using different performance metrics for each. ParaProf uses scalable histogram and three-dimensional displays for larger datasets.

To provide more sophisticated performance analysis capabilities, we developed support for parallel performance data mining in TAU. *PerfExplorer* [4, 6] is a framework for performance data mining motivated by our interest in automatic parallel performance analysis and by our concern for extensible and reusable performance tool technology. PerfExplorer is built on PerfDMF and targets large-scale performance analysis for single experiments on thousands of processors and for multiple experiments from parametric studies. PerfExplorer uses techniques such as clustering and dimensionality reduction to manage large-scale data complexity.

Summary

The TAU Performance System® has undergone several incarnations in pursuit of its primary objectives of flexibility, portability, integration, interoperability, and scalability. The outcome is a robust technology suite that has significant coverage of the performance problem solving landscape for high-end computing. TAU follows a direct performance observation methodology since it is based on the observation of effects directly associated with the program's exection, allowing performance

data to be interpreted in the context of the computation. Hard issues of instrumentation scope and measurement intrusion have to be addressed, but these have been aggressively pursued and the technology enhanced in several ways during TAU's lifetime. TAU is still evolving, and new capabilities are being added to the tools suite. Support for whole-system performance analysis, model-based optimization using performance expectations and knowledge-based data mining, and heterogeneous performance measurement are being pursued.

Related Entries

►Metrics
►Performance Analysis Tools

Bibliography

1. Bell R, Malony A, Shende S (2003) A portable, extensible, and scalable tool for parallel performance profile analysis. In: European conference on parallel computing (EuroPar 2003), Klagenfurt
2. Brunst H, Kranzlmüller D, Nagel WE (2004) Tools for scalable parallel program analysis – Vampir NG and DeWiz. In: Distributed and parallel systems, cluster and grid computing, vol 777. Springer, New York
3. Brunst H, Nagel W, Malony A (2003) A distributed performance analysis architecture for clusters. In: IEEE international conference on cluster computing (Cluster 2003), pp 73–83. IEEE Computer Society, Los Alamitos
4. Huck KA, Malony AD (2005) Perfexplorer: a performance data mining framework for large-scale parallel computing. In: High performance networking and computing conference (SC'05). IEEE Computer Society, Los Alamitos
5. Huck K, Malony A, Bell R, Morris A (2005) Design and implementation of a parallel performance data management framework. In: International conference on parallel processing (ICPP 2005). IEEE Computer Society, Los Alamitos
6. Huck K, Malony A, Shende S, Morris A (2008) Knowledge support and automation for performance analysis with PerfExplorer 2.0. J Sci Program 16(2–3):123–134 (Special issue on large-scale programming tools and environments)
7. Knüpfer A, Brendel R, Brunst H, Mix H, Nagel WE (2006) Introducing the Open Trace Format (OTF). In: International conference on computational science (ICCS 2006). Lecture notes in computer science, vol 3992. Springer, Berlin, pp 526–533
8. Mayanglambam S, Malony A, Sottile M (2009) Performance measurement of applications with GPU acceleration using CUDA. In: Parallel computing (ParCo), Lyon
9. Mohr B, Wolf F (2003) KOJAK – a tool set for automatic performance analysis of parallel applications. In: European conference on parallel computing (EuroPar 2003). Lecture notes in computer science, vol 2790. Springer, Berlin, pp 1301–1304
10. Nataraj A, Sottile M, Morris A, Malony AD, Shende S (2007) TAUoverSupermon: low-overhead online parallel performance monitoring. In: European conference on parallel computing (EuroPar 2007), Rennes
11. Nataraj A, Morris A, Malony AD, Sottile M, Beckman P (2007) The ghost in the machine: observing the effects of kernel operation on parallel application performance. In: High performance networking and computing conference (SC'07), Reno
12. Nataraj A, Malony A, Morris A, Arnold D, Miller B (2008) In search of sweet-spots in parallel performance monitoring. In: IEEE international conference on cluster computing (Cluster 2008), Tsukuba
13. Nataraj A, Malony A, Morris A, Arnold D, Miller B (2008) TAUoverMRNet (ToM): a framework for scalable parallel performance monitoring. In: International workshop on scalable tools for high-end computing (STHEC '08), Kos
14. Nataraj A, Malony AD, Shende S, Morris A (2008) Integrated parallel performance views. Clust Comput 11(1):57–73
15. Shende S (2001) The role of instrumentation and mapping in performance measurement. Ph.D. thesis, University of Oregon
16. Shende S, Malony A (2006) The TAU parallel performance system. Int J Supercomput Appl High Speed Comput 20(2, Summer):287–311 (ACTS collection special issue)
17. Shende S, Malony AD, Cuny J, Lindlan K, Beckman P, Karmesin S (1998) Portable profiling and tracing for parallel scientific applications using C++. In: SIGMETRICS symposium on parallel and distributed tools, SPDT'98, Welches, pp 134–145
18. Wolf F et al (2008) Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In: Proceedings of the second HLRS parallel tools workshop, Stuttgart. Lecture notes in computer science. Springer, Berlin

TAU Performance System®

►TAU

TBB (Intel Threading Building Blocks)

►Intel® Threading Building Blocks (TBB)

Tensilica

►Green Flash: Climate Machine (LBNL)

Tera MTA

Burton Smith
Microsoft Corporation, Redmond, WA, USA

Synonyms

Cray MTA; Cray XMT; Horizon

Definition

The MTA (for Multi-Threaded Architecture) is a highly multithreaded scalar shared-memory multiprocessor architecture developed by Tera Computer Company (renamed Cray Inc. in 2000) in Seattle, Washington. Work began in 1985 at The Institute for Defense Analyses Center for Computing Sciences on a closely related predecessor (Horizon), and development of both hardware and software was continuing at Cray Inc. as of 2010.

Discussion

Introduction

The Tera MTA [1] is in many respects a direct descendant of the Denelcor HEP computer [2]. Like the HEP, the MTA is a scalar shared-memory system equipped with full/empty bits at every 64-bit memory location and multiple protection domains to permit multiprogramming within a processor. However, the MTA introduced a few innovations including VLIW instructions without any register set partitioning, additional ILP via dependence data encoded in each instruction, two-phase blocking synchronization, unlimited data breakpoints, speculative loads, division and square root to full accuracy using iterative methods, operating system entry via procedure calls, traps that never change privilege, and no interrupts at all.

Software developed for the MTA introduced its share of novel ideas as well, including a user-mode runtime responsible for synchronization and work scheduling, negotiated resource management between the user-mode runtime and the operating system, an operating system that returns control to the user-mode runtime when the call blocks, a compiler for Fortran and C++ that parallelizes and restructures a wide variety of loops including those whose inter-iteration dependences require a parallel prefix computation, and dynamic scheduling of loop nests having mixed rectangular, triangular, and skyline loop bounds.

Beginnings

While spending the summer of 1984 as an intern at Denelcor, UC Berkeley graduate student Stephen W. Melvin invented a scheme for organizing a register file in a fine-grain multi-threaded processor to let VLIW instructions enjoy multiple register accesses per instruction while preserving a flat register address space within each hardware thread. The idea was simple: organize the register file into multiple banks with each bank containing all of the registers for an associated subset of the hardware threads; let each issued instruction use multiple cycles to read and write its associated bank as many times as necessary to implement the instruction; and have the instruction issue logic refrain from issuing from threads associated with currently busy banks. From this beginning, an architectural proposal emerged that was whimsically referred to as "Vulture" and was later known as the "HEP array processor". Denelcor envisioned heterogeneous systems with second-generation HEP processors sharing memory with processors based on this new VLIW idea.

Denelcor filed for Chapter 7 bankruptcy in mid-1985 whereupon its CTO, Burton J. Smith, joined the Supercomputing Research Center (now called the Center for Computing Sciences) of the Institute for Defense Analyses (IDA) in Maryland. His plan was to further evaluate the merits of the multithreaded VLIW ideas. It had already become clear that code generation and optimization for such a processor was only slightly more difficult than for the HEP but the resulting performance was potentially much higher. The design that resulted from collaborations on this topic at IDA over the years 1985–1987 was known as *Horizon* and was described in a series of papers presented at Supercomputing'88 [3, 4].

Horizon instructions were 64 bits wide and typically contained three operations: a memory reference (M) operation, an arithmetic or logic (A) operation, and a control (C) operation which did branches but could also do some arithmetic and logic. As many as ten five-bit register references might appear in any single instruction. The memory reference semantics were derived from those of the HEP but added data trap bits to implement data break points (watchpoints) and possibly other things. The A operations included fused multiply-adds for both integer and floating point arithmetic and a rich variety of operations on vectors and matrices of bits. Branches were encoded compactly as an opcode, an eight-bit condition mask, a two-bit condition code

number specifying one of the four most recently generated three-bit condition codes, and two bits naming a branch target register that had been preloaded with the branch address. Most A- and C-operations emitted a condition code as an optional side-effect.

Horizon increased ILP beyond three with an idea known as *explicit-dependence lookahead*, replacing the usual register reservation scheme. Intel would later employ a kindred concept for encoding dependence information within instructions in the Itanium architecture, calling it *explicitly parallel instruction computing* (EPIC). The Horizon version included in every instruction an explicit 3-bit unsigned integer, the *lookahead* that bounded from below the number of instructions separating this instruction from later ones that might depend on it. Subsequent instructions within the bound could overlap with the current instruction. To implement this scheme, every hardware thread was equipped with a three-bit *flag* counter, incremented when the thread is selected to issue its next instruction, and an array of eight three-bit *lock* counters. On instruction issue, the lock counter subscripted by the lookahead plus the flag (mod 8) is incremented; when the instruction fully retires, that same lock is decremented. A thread is permitted to issue its next instruction when the lock subscripted by its flag is zero. Instruction instances were thus treated very much like operations in a static data flow machine. As a further refinement, branches were available to terminate lookahead along the unlikely control flow direction, potentially increasing ILP along the likelier one.

Tera

Early in 1988, Smith and a colleague, James E. Rottsolk, decided to start a company to build general-purpose parallel computer systems based on the Horizon concepts. They named the new company *Tera*, acquired initial funding from private investors and from the Defense Advanced Research Projects Agency (DARPA), and began to search for a suitable home. In August 1988, Seattle, Washington was chosen and Tera began recruiting engineers. The University of Washington helped the company in its early days.

Languages

The Tera language strategy [5] was to add directives and pragmas to Fortran and C++ to guide compiler loop parallelization and provide performance and compatibility with existing vector processors. A consistency model strongly resembling release consistency was adopted based on acquire and release synchronization points to let the compiler cache values in registers and restructure code aggressively. Basically, memory references could not move backward over acquires or forward over releases. Only the built-in synchronization based on full/empty bits was permitted at first; later, volatile variable references were made legal (but deprecated) for synchronization. A *future* statement borrowed from Multilisp [6] was introduced in both Fortran and C++ to support task parallelism as well as divide-and-conquer data parallelism. It uses the full/empty bits to synchronize completion of the future with its invoker. The body of the future appears in-line and can reference variables in its enclosing environment.

Compiler Optimization

The MTA compilers can automatically parallelize a variety of loops [7]. Consider the example below:

```
void sort(int *src, int *dst, int nitems, int nvals) {
   int i, j, t1[nvals], t2[nvals];
   for (j = 0; j< nvals; j++) {
      t1[j] = 0;}
   for (i = 0; i< nitems; i++) {
      t1[src[i]]++;}                    //atomic update
   t2[0] = 0;
   for (j = 1; j<nvals; j++) {
      t2[j] = t2[j-1] + t1[j-1];}       //parallel prefix
   $t2 = (sync int *) t2;
#pragma tera assert parallel
   for (i = 0; i<nitems; i++) {
      dst[$t2[src[i]]++] = src[i];}
}
```

All four loops are parallelized by the MTA compiler. Updates like the one in the second loop are automatically made atomic using full/empty bits if and as necessary. The third loop is an example of a *parallel prefix computation* [8] (also called a *parallel scan*) which the compiler can automatically parallelize as long as the internal state of the accumulating prefix is bounded [9]. The fourth loop will not be parallelized automatically by the compiler and requires a pragma and explicit use of full/empty bits via the *sync* type qualifier.

To achieve as high a level of ILP as the instruction set can afford, software pipelining [10] is exploited by distributing loops based on estimates of register pressure and then unrolling and packing to obtain a good schedule. Experiments at both IDA and Tera led to a modification of the lookahead scheme to overlap only memory references. Software pipelining was further enabled by implementing speculative loads. A *poison* bit is associated with every register in the thread, and memory protection violations can optionally be made to poison the destination register instead of raising an exception. Any instruction that attempts to use a poisoned value will trap instead. The speculative load feature allows prefetching in a software pipeline without having to "unpeel" final iterations to prevent accesses beyond mapped memory. In any case the instruction can be reattempted if the cause of the protection violation is remediable.

Nests of parallelizable "for" loops may vary in iterations per loop, sometimes dynamically, making them hard to execute efficiently. The MTA compiler schedules these nests as a whole, even when inner loops have bounds that depend on outer iteration variables. First, code is generated to compute the total number of (inner loop) iterations of the whole nest. Functions are then generated to compute the iteration number of each loop from the total iteration count. Finally, code is generated to dynamically schedule the loop nest by having each worker thread acquire a "chunk" of total iterations using a variant of guided self-scheduling [11], reconstruct the iteration variable bindings, and then jump into the loop nest to iterate until the chunk is consumed.

User-Level Runtime

A user-level runtime environment was developed to help implement the language features and schedule fine-grain parallel tasks. The Horizon architecture was modified to make full/empty synchronization operations lock the location and generate a user-level trap after a programmable limit on the number of synchronization retry attempts is exceeded. The runtime trap handler saves the state of the blocked task, initializes a queue of blocked tasks containing this one as its first element, and places a pointer to this new queue in the still-locked full/empty location. It then sets a trap bit and unlocks the location so that subsequent references of any kind will trap immediately. In this way the tasks that arrive later either block, joining the queue of waiting tasks right away, or dequeue a waiting task immediately. The retry limit is set to match the time needed to save and later restore the task state, making this scheme within a factor of two of optimal. When memory references are frequent, the retry rate is throttled to be much less than that for new memory references, and if the processor is not starved for hardware threads the polling cost becomes almost negligible and the retry limit can be increased substantially.

When formerly blocked tasks are unblocked, they are enqueued in a pool of runnable tasks. Since these tasks are equipped with a stack and may have additional memory associated with them, they are run in preference to tasks associated with future statements that have not yet run. Still, the number of blocked tasks can be substantial. To reduce memory waste, stacks are organized as linked lists of fixed-size blocks. Automatic arrays and other things that must be contiguously allocated are stored in the heap. Interprocedural analysis is used to avoid most stack bounds checks.

Another modification to the Horizon architecture comprised instructions to allocate multiple hardware threads. Each protection domain has an operating system-imposed limit on the number of hardware threads in the domain and a *reservation* which can be increased (or decreased) by one of the new instructions [12]. One of them reserves a variable number, from zero up to a maximum specified as an argument, depending on availability. The other instruction either reserves the requested amount or none at all. In either case, the number of additional hardware threads actually allocated is returned in a register so a loop can be used to initiate them. The primary motivation for this reservation capability was to accommodate rapidly varying quantities of parallelism found in short parallel loops. Hardware threads can be materialized and put to work quickly when such opportunities are encountered.

Operating System

Since the MTA's operating system (OS) plays no role in user-level thread synchronization and allocates but does not micromanage the dynamic quantity of hardware threads, the usual OS invocation machinery (trapping) was rejected in favor of procedure calls. A protection ring-crossing instruction guarantees only valid OS

entry points are called. The operating system can allocate its own stack space when and if necessary. If an OS call ultimately blocks, the hardware thread is returned to user level via a runtime entry point that associates the continuation of the original user computation with a "cookie" supplied by the OS. When the original OS call completes, the appropriate cookie is passed to the user runtime so it can unblock the user-level continuation. As a result, the operating system executes in parallel with the user-level program. This scheme was invented independently [13] at the University of Washington and is referred to as *scheduler activations*.

When an illegal operation is attempted, a trap to the user-level trap handler occurs. If OS intervention is required, the trap handler calls the OS to service the trap. To evict a process from a protection domain, all of its hardware threads are made to trap in response to an OS-generated signal. The OS also has the ability to kill all hardware threads in a protection domain.

Memory Mapping

The MTA has a large virtual data address space, making high-performance memory address translation challenging. To address this issue, segmentation is used instead of paging, with a 48-bit virtual address comprising a 20-bit segment number and a 28-bit offset. Contiguous allocations of memory larger than 256 MB can use multiple segments. Segment size granularity is 8 KB. Each protection domain has base and limit registers that define its own range of segment numbers. A segment map entry specifies minimum privilege levels for loads and stores, and whether physical addresses are to be *distributed* across the multiple memory banks of the system by "scrambling" the address bits [14]. When memory is distributed in this fashion there are virtually no bank conflicts due to strided accesses. Program memory, as addressed by the program counters, is handled differently using a paging scheme with 4 KB pages. The program space is mapped to data space via a non-distributed (i.e., local) segment.

Arithmetic

The instruction set supports the usual variety of two's complement and unsigned integers and both 32- and 64-bit IEEE 754 floating point. Division for signed and unsigned integers [15] along with floating point division and square root all use Newton's method but nevertheless implement full accuracy and correct semantics. Excepting these few operations, denormalized arithmetic is fully supported in hardware. High-precision integer arithmetic is abetted by a 128-bit unsigned integer multiply instruction and the ability to propagate carry bits easily. There is also support for a 128-bit floating point format using pairs of 64-bit floats; the smaller value is insignificant with respect to the larger, thereby yielding 106 bits of significand precision or more. The existence of a fused multiply-add makes this format relatively inexpensive to implement and use, but instruction modifications were needed to mitigate a variety of pathologies. A true 128-bit IEEE format would doubtless be preferable.

MTA-1 and MTA-2

The MTA-1 was a water-cooled system built from Gallium Arsenide logic. To provide adequate memory bandwidth, the processors sparsely populated a 3D toroidal mesh interconnection network. As an example, 512 routing nodes were required for 64 processors. To make network wiring implementable, one-third of the mesh links were elided. The first and only MTA-1 system was delivered to the San Diego Supercomputer Center in June 1999.

The MTA-2 was a major improvement in manufacturability over the MTA-1. It used CMOS logic and had an interconnection network based on notions from group theory [16]. It was first delivered in 2002 to the US Naval Research Laboratory in Washington, DC. A few other MTA-2 systems were built before deliveries of the Cray XMT (*q.v.*) began in 2007.

Related Entries

►Cray MTA
►Cray XMT
►Data Flow Computer Architecture
►Denelcor HEP
►EPIC Processors
►Futures
►Interconnection Networks
►Latency Hiding
►Little's Law
►Memory Wall
►MIMD (Multiple Instruction, Multiple Data) Machines

►Modulo Scheduling and Loop Pipelining
►Multilisp
►Multi-Threaded Processors
►Networks, Direct
►Networks, Multistage
►Processes, Tasks, and Threads
►Processors-in-Memory
►Shared-Memory Multiprocessors
►SPMD Computational Model
►Synchronization
►Ultracomputer, NYU
►VLIW Processors

Bibliography

1. Alverson R, Callahan D, Cummings D, Koblenz B, Porterfield A, Smith B (1990) The Tera computer system. In: Proceedings of the 1990 international conference on supercomputing, Amsterdam
2. Smith BJ (1981) Architecture and applications of the HEP multiprocessor computer system. Proc SPIE Real-Time Signal Process IV 298:241–248
3. Kuehn JT, Smith BJ (1988) The Horizon supercomputing system: architecture and software. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, Orlando
4. Thistle MR, Smith BJ (1988) A processor architecture for Horizon. In: Proceedings of the 1988 ACM/IEEE conference on supercomputing, Orlando
5. Callahan D, Smith B (1990) A future-based parallel language for a general-purpose highly-parallel computer. In: Selected papers of the second workshop on languages and compilers for parallel computing, Irvine
6. Halstead RH (1985) MultiLisp: a language for concurrent symbolic computation. ACM T Program Lang Syst 7(4):501–538
7. Alverson G, Briggs P, Coatney S, Kahan S, Korry R (1997) Tera hardware-software cooperation. In: Proceedings of supercomputing, San Jose
8. Ladner RE, Fischer MJ (1980) Parallel prefix computation. J ACM 27(4):831–838
9. Callahan D (1991) Recognizing and parallelizing bounded recurrences. In: Proceedings of the fourth workshop on languages and compilers for parallel computing, Santa Clara
10. Lam M (1988) Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the ACM SIGPLAN 88 conference on programming language design and implementation, Atlanta
11. Polychronopoulos C, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE T Comput C-36(12):1425–1439
12. Alverson G, Alverson R, Callahan D, Koblenz B, Porterfield A, Smith B (1992) Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In: Proceedings of the 1992 international conference on supercomputing, Washington, DC
13. Anderson T, Bershad B, Lazowska E, Levy H (1992) Scheduler activations: effective kernel support for the user-level management of parallelism. ACM T Comput Syst 10(1):53–79
14. Norton A, Melton E (1987) A class of Boolean linear transformations for conflict-free power-of-two stride access. In: Proceedings of the international conference on parallel processing, St. Charles, IL
15. Alverson R (1991) Integer division using reciprocals. In: Proceedings of the 10th IEEE symposium on computer arithmetic, Grenoble
16. Akers S, Krishnamurthy B (1989) A group-theoretic model for symmetric interconnection networks. IEEE T Comput C-38(4):555–566
17. Alverson G, Kahan S, Korry R, McCann C, Smith B (1995) Scheduling on the Tera MTA. In: Proceedings of the first workshop on job scheduling strategies for parallel processing, Santa Barbara. Lecture Notes in Computer Science 949:19–44

Terrestrial Ecosystem Carbon Modeling

Dali Wang[1], Daniel Ricciuto[1], Wilfred Post[1], Michael W. Berry[2]
[1]Oak Ridge National Laboratory, Oak Ridge, TN, USA
[2]The University of Tennessee, Knoxville, TN, USA

Synonyms

Carbon cycle research; System integration; Terrestrial ecosystem modeling; Uncertainty quantification

Definition

A Terrestrial Ecosystem Carbon Model (TECM) is a category of process-based ecosystem models that describe carbon dynamics of plants and soils within global terrestrial ecosystems. A TECM generally uses spatially explicit information on climate/weather, elevation, soils, vegetation, and water availability as well as soil- and vegetation-specific parameters to make estimates of important carbon fluxes and carbon pool sizes in terrestrial ecosystems.

Discussion

Introduction

Terrestrial ecosystems are a primary component of research on global environmental change. Observational and modeling research on terrestrial ecosystems at the global scale, however, has lagged behind

their counterparts for oceanic and atmospheric systems, largely because of the unique challenges associated with the tremendous diversity and complexity of terrestrial ecosystems. There are eight major types of terrestrial ecosystem: tropical rain forest, savannas, deserts, temperate grassland, deciduous forest, coniferous forest, tundra, and chaparral. The carbon cycle is an important mechanism in the coupling of terrestrial ecosystems with climate through biological fluxes of CO_2. The influence of terrestrial ecosystems on atmospheric CO_2 can be modeled via several means at different timescales to incorporate several important processes, such as plant dynamics, change in land use, as well as ecosystem biogeography. Over the past several decades, many terrestrial ecosystem models (see the "►Model Developments" section) have been developed to understand the interactions between terrestrial carbon storage and CO_2 concentration in the atmosphere, as well as the consequences of these interactions. Early TECMs generally adapted simple box-flow exchange models, in which photosynthetic CO_2 uptake and respiratory CO_2 release are simulated in an empirical manner with a small number of vegetation and soil carbon pools. Demands on kinds and amount of information required from global TECMs have grown. Recently, along with the rapid development of parallel computing, spatially explicit TECMs with detailed process-based representations of carbon dynamics become attractive, because those models can readily incorporate a variety of additional ecosystem processes (such as dispersal, establishment, growth, mortality, etc.) and environmental factors (such as landscape position, pest populations, disturbances, resource manipulations, etc.), and provide information to frame policy options for climate change impact analysis.

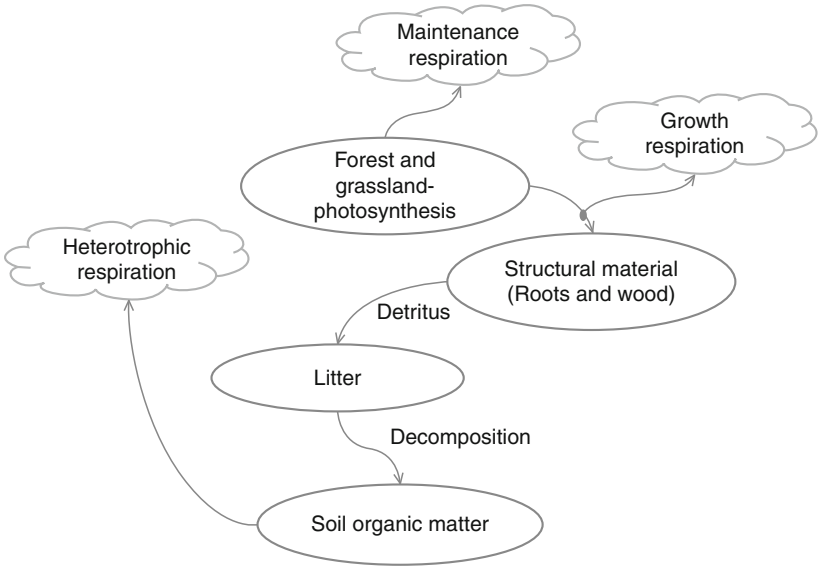
Key Components of TECM

1. Fundamental terrestrial ecosystem carbon dynamics

Terrestrial carbon processes can be described by an exchange between four major compartments: (1) foliage where photosynthesis occurs; (2) structural material, including roots and wood; (3) surface detritus or litter; and (4) soil organic matter (including peat). Nearly all life on Earth depends (directly or indirectly) on photosynthesis, in which carbon dioxide and water are used, and oxygen is released. The majority of the carbon in the living vegetation of terrestrial ecosystems is found in woody material, which constitutes a major carbon reservoir in the carbon cycle. Detritus refers to leaf litter and other organic matter on or below the soil surface. Dead woody material, often called coarse woody debris, is a large component of the surface detritus in forest ecosystems. Detritus is typically colonized by communities of microorganisms which act to decompose (or remineralize) the material. Transformation and translocation of detritus is the source of soil organic matter, another major component in the global carbon cycle. Globally three times as much carbon is stored in soils as in the atmosphere with peatlands contributing a third of this. Thus even relatively small changes in soil C stocks might contribute significantly to atmospheric CO_2 concentrations and thus global climate change. The soil carbon pool is vulnerable to impacts of human activity especially agriculture. A simplified scheme of carbon cycle dynamics is shown in Fig. 1.

2. Terrestrial carbon observations and experiments

Early research generally focused on determining characteristics of individual plants and small soil samples, often in a laboratory setting. This type of research continues today and provides a wealth of information that is used to develop and to parameterize TECMs. However, successful modeling of the carbon cycle also requires understanding the structure and response of entire ecosystems. Observation networks involving ecosystems have expanded greatly in the past two decades. One important development for in situ monitoring of ecosystem-level carbon exchange has been the establishment of flux towers that use the eddy covariance method. Atmospheric CO_2 concentration measurements using satellites, tall towers, and aircraft provide information about carbon dioxide fluxes over a larger scale. Finally, remote sensing products provide important information about changes in land use and vegetation characteristics (e.g., total leaf area) that can be used to either drive or validate TECMs. While these observations are important for characterizing the carbon cycle at present, they do not provide information about how the carbon cycle may change in the future as a result of climate change. In order to address those challenging questions, several large-scale ecosystem-level experiments have been conducted to mimic possible



Terrestrial Ecosystem Carbon Modeling. Fig. 1 Simplified schematic of the carbon dynamics (photosynthesis, autotrophic respiration, allocation, and heterotrophic respiration) within a typical TECM model

future conditions (such as rising CO_2 concentrations, potential future precipitation patterns and the future temperature scenarios) and associated impacts and mitigation options for terrestrial ecosystems. Figure 2 illustrates some of these carbon observation systems and experiments.

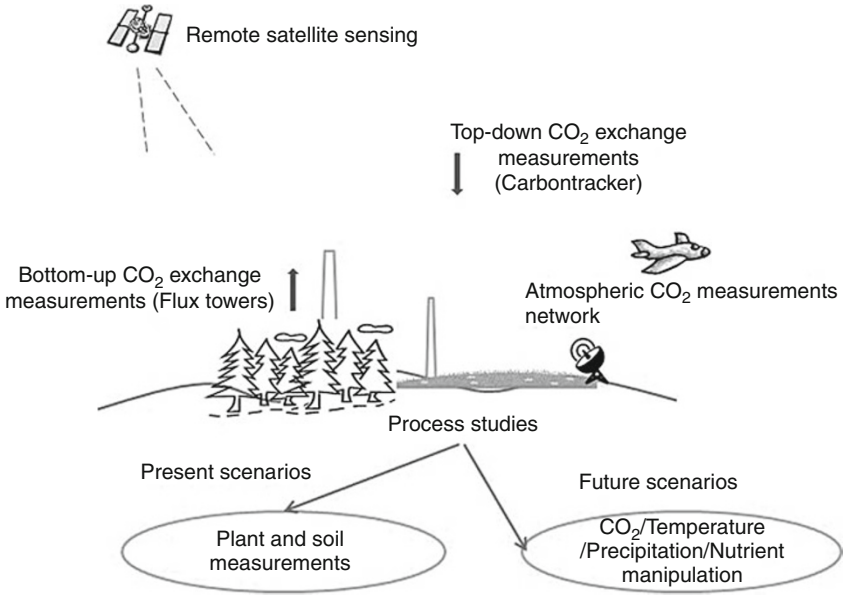
3. Terrestrial ecosystem carbon model developments

In the early 1970s, several process-based conceptual models were developed to study the primary productivity of the biosphere and the uptake of anthropogenic CO_2 emissions. Early box models were improved in the 1980s to include more spatially explicit ecological representations of terrestrial ecosystems along with a significant push to understand the relationships between climatic measurements and properties of ecosystem processes. The concept of biomes was used to categorize terrestrial ecosystems using several climatically and geographically related factors (i.e., plant structure, leaf types, and climate), instead of the traditional classification by taxonomic similarity. In the 1990s, rapid developments of general circulation models and scientific computing, along with the increasing availability of remote sensing data (from satellites), led to the development of land-surface models. These models used satellite images to obtain information about the spatial distribution of surface properties (such as vegetation type, phenology, and density) along with spatially explicit forcing from numerical weather prediction reanalysis or coupled general circulation models (e.g., temperature and precipitation) to improve prediction and enhance the model representation of land-atmosphere water and energy interactions within global climate models. Recently, emphasis has been focused on improving the predictive capacity of climate models at the decadal to century scale through a better characterization of carbon cycle feedbacks with climate. For example, several TECMs are incorporating nutrient cycles and shifts in vegetation distribution, in response and a potential feedback to climate change.

The Contributions of Parallel Computing to TECM Developments

The contributions of parallel computing to the TECM can be classified into three separate categories: (1) model construction, (2) model integration, and (3) model behavior controls.

As more processes are incorporated into TECMs to replace simple empirical relationships, computational demands have increased. Since these processes are very



Terrestrial Ecosystem Carbon Modeling. Fig. 2 Illustrated carbon observations and experiments

sensitive to environmental heterogeneity inherent in spatial patterns of temperature, radiation, precipitation, soil characteristics, etc., increased spatial resolution improved simulation accuracy. A new class of models is becoming more widespread for global scale TECMs. This class, instead of using a traditional system of differential equations, is agent-based and requires a fine grid spatial representation to represent the competition for resources among the coexisting agents. This approach dramatically increases the computational demand, but is more compatible with experimental and observational data and population scale vegetation change processes. Parallel computing has enabled models to be constructed with these additional complexities.

Over several decades of research, TECMs have dramatically changed in structure and in the amount and kind of information required and produced making model integration a challenge. In addition, these models are now being incorporated into climate models to form Earth System Models (ESMs). From a parallel computing perspective, there is huge demand from the modeling community to develop a parallel model coupling framework (Earth System Modeling Frame (ESMF) is one of these kinds of efforts) to enable further parallel model developments and validations. Instead of rewriting a package *wrapper* for each component, memory-based IO staging systems may provide an alternative method for fast and seamless coupling. There are two basic methods to provide climate forcing information for a TECM, from observation data or coupling to a global climate model simulation. Currently, model simulation can provide global data, but only at low spatial resolutions. Observation datasets are available at those fine resolutions, and can be used for validation over observed time frames at those observation stations. However, further research and parallel computing will be needed for gap-filling and downscaling those observation datasets for global terrestrial ecosystem carbon modeling.

One consequence of TECM complexity is the increasing demand for better methodologies that can exploit ever-increasing rich datastreams and thereby improve model behavior (e.g., the ecosystem model's sensitivity to the model parameters, and software structure). Quantitative methods need to be established to determine model uncertainty and reduce uncertainty through model-data analysis. As computers become larger and larger in the number of CPU cores, not necessarily faster and faster at the single CPU core level, we envision that ultrascale software designs for systematic

uncertainty quantification for TECM will became one research area which will require the full advantage of parallel computing and statistics.

As our understanding of the global carbon cycle improves, high fidelity, process-based models will continue to be developed, and the increasing complexity of these ecosystem model systems will require that parallel computing play an increasingly important role. We have explained several key components of terrestrial ecosystem carbon modeling, and have classified three categories that parallel computing can play significant contributions to the TECM developments. It is our view that parallel computing will increasingly be an integral part of modern terrestrial ecosystem modeling efforts, which require solid, strong partnerships between the high-performance computing community and the carbon cycle science community. Through such partnerships these two communities can share a common mission to advance our understanding of global change using computational sciences.

Related Entries

►Analytics, Massive-Scale
►Computational Sciences
►Exascale Computing

Bibliographic Notes and Further Reading

As mentioned in the model development section, terrestrial carbon modeling started in the early 1970s [1, 2], when beta-factor concept was developed to account for CO_2 fertilization using a nonspatial representation of terrestrial carbon dynamics. In 1975, Lieth described a model (MIAMI, the first gridded model) [3] to estimate the primary productivity of the biosphere. A carbon accounting model was developed at Marine Biological Laboratory (MBL) at Woods Hole to track carbon fluxes associated with land-use change. Along with the success of International Biological Program, spatially distributed compartment models representing different ecosystem types responding to local environmental conditions were developed. Widely used examples include the Terrestrial Ecosystem Model (www.mbl.edu/eco42/) at MBL, and CENTURY (www.nrel.colostate.edu/projects/century/) at Colorado State University. As satellite measurements of basic terrestrial properties became available, several models were developed that utilized this information directly, including the Ames-Stanford Approach (CASA) (geo.arc.nasa.gov/sge/casa/bearth.html) and Biome-BGC (www.ntsg.umt.edu/models/bgc/). In the 1990s, land surface components of climate models incorporated an aspect of terrestrial carbon cycling, namely photosynthesis, for the purpose of providing a mechanistic model of latent heat exchange with the atmosphere. The Simple Biosphere (SiB) biophysical model [4] and Biosphere-Atmosphere Transfer Scheme (BATS) [5] at National Center for Atmospheric Research (NCAR), and STOMATE [6] at Laboratoire des Sciences du Climat et de l'Environnement (LSCE) are examples. Later at NCAR, additional components of terrestrial carbon cycle were included in the Land Surface Model (LSM) (www.cgd.ucar.edu/tss/lsm/). The Community Land Model (CLM-CN) (www.cgd.ucar.edu/tss/clm/) is the successor of LSM and is being further developed as a community-based model. Agent-based models at the global scale, a class of what are called Dynamic Global Vegetation Models (DGVM), have been developed independently because of their data and computation demands. Developments in parallel computer systems are making incorporation of such dynamics plausible for earth system models. HYBRID [7] was an early experimental model, and now prototypes exist for the NCAR CCSM land surface CLM-CN which is based on the Lund-Postdam-Jena (LPJ) model [8] and called CLM-CN-DV. Evolved from the Ecosystem Demography (ED) model [9, 10], ENT [11] is another Dynamic Global Terrestrial Ecosystem Model (DGTEM), being coupled with NASA's GEOS-5 General Circulation Models.

Observation networks involving ecosystems have expanded greatly in the past two decades. AmeriFlux (public.ornl.gov/ameriflux/) is an effort to use flux towers to monitor ecosystem-level carbon exchange with atmosphere. Since 1990, more than 400 such flux towers have been established on six continents representing every major biome. First established in Mauna Loa in 1958, the CO_2 measurements have become a global operation. Inversion techniques are used to infer the pattern of CO_2 fluxes required to produce the observed CO_2 concentrations; one such product using this technique is CarbonTracker (www.esrl.noaa.gov/gmd/ccgg/carbontracker/), which provides weekly flux

estimates that can be compared against output from process-based TECMs. Currently, a variety of remote sensing products (such as Moderate Resolution Imaging Spectroradiometer (MODIS)) are available to either drive or validate TECMs.

Several experiments have been conducted or initiated to understand potential climate change impacts. The Free-Air CO_2 Enrichment (FACE) (public.ornl.gov/face/global_face.shtml) experiment has been running for over a decade at several sites in different biomes to study the potential effects of higher CO_2 concentration. The Throughfall Displacement Experiment (TDE) (tde.ornl.gov/) used elaborate systems to alter the amount of precipitation that is available to an ecosystem. A new experiment has been initiated at Oak Ridge National Laboratory to assess the responses of northern peatland ecosystems to increased temperature and exposures to elevated atmospheric CO_2 concentrations (mnspruce.ornl.gov).

More information on terrestrial ecosystem carbon modeling can be found in books devoted to this subject [12, 13].

Bibliography

1. Bacastow RB, Keeling CD (1973) Atmospheric carbon dioxide and radiocarbon in the natural carbon cycle: II. Changes from AD 1700 to 2070 as deduced from a geochemical model. In: Woodwell GM, Pecan EV (eds) Carbon and the biosphere. CONF-720510. National Technical Information Service, Springfield, Virginia, pp 86–135
2. Emanuel WR, Killough GG, Post WM, Shugart HH (1984) Modeling terrestrial ecosystems in the global carbon cycle with shifts in carbon storage capacity by land-use change. Ecology 65(3): 970–983
3. Lieth H (1975) Modeling the primary productivity of the world. In: Lieth H, Wittaker RH (eds) Primary productivity of the biosphere, ecological studies, vol 14. Springer-Verlag, New York, pp 237–283
4. Sellers JP, Randell DA, Collatz GJ, Berry JA, Field CB, Dazlich DA, Zhang C, Collelo GD, Bounua L (1996) A revised land surface parametrization (SiB 2) for atmospheric GCMs. Part I: model formulation. J Climate 9:676–705
5. Dickinson R, Henderson-sellers A, Kennedy P (1993) Biosphere-atmosphere transfer scheme (BATS) version as coupled to the NCAR community climate model. Technical report, National Center for Atmospheric Research
6. Ducoudré N, Laval K, Perrier A (1993) SECHIBA, a new set of parametrizations of the hydrologic exchanges at the land/atmosphere interface within the LMD atmospheric general circulation model. J Climate 6(2):248–273
7. Friend AD, Stevens AK, Knox RG, Cannell MGR (1997) A process-based, terrestrial biosphere model of ecosystem dynamics (Hybrid v3.0). Ecol Model 95:249–287
8. Prentice IC, Heimann M, Sitch S (2000) The carbon balance of the terrestrial biosphere: ecosystem models and atmospheric observations. Ecol Appl 10:1553–1573
9. Moorcroft P, Hurtt GC, Pacala SW (2001) A method for scaling vegetation dynamics: the ecosystem demography model (ED). Ecol Monogr 71(4):557–586
10. Govindarajan S, Dietze MC, Agarwal PK, Clark JS (2004) A scalable simulator for forest dynamics. In: Proceedings of the twentieth annual symposium on computational geometry SCG 04, Brooklyn, NY, pp 106–115, doi:10.1145/997817.997836
11. Yang W, Ni-Meister W, Kiang NY, Moorcroft P, Strahler AH, Oliphant A (2010) A clumped-foliage canopy radiative transfer model for a Global Dynamic Terrestrial Ecosystem Model II: Comparison to measurements. Agricultural and Forest Meteorology, 150(7):895–907, doi:10.1016/j.agrformet.2010.02.008
12. Trabalka JR, Reichle DE (ed) (1986) The changing carbon cycle: a global analysis. Springer-Verlag, Berlin
13. Field CB, Raupach MR (ed) (2004) The global carbon cycle: integrating human, climate, and the natural world. Island, Washington, DC

Terrestrial Ecosystem Modeling

►Terrestrial Ecosystem Carbon Modeling

The High Performance Substrate

►HPS Microarchitecture

Theory of Mazurkiewicz-Traces

►Trace Theory

Thick Ethernet

►Ethernet

Thin Ethernet

►Ethernet

Thread Level Speculation (TLS) Parallelization

►Speculation, Thread-Level

Thread-Level Data Speculation (TLDS)

►Speculative Parallelization of Loops
►Speculation, Thread-Level

Thread-Level Speculation

►Speculative Parallelization of Loops
►Speculation, Thread-Level

Threads

►Processes, Tasks, and Threads

Tiling

François Irigoin
MINES ParisTech/CRI, Fontainebleau, France

Synonyms

Blocking; Hyperplane partitioning; Loop blocking; Loop tiling; Supernode partitioning

Definition

Tiling is a program transformation used to improve the spatial and/or temporal memory locality of a loop nest by changing its iteration order, and/or to reduce its synchronization or communication overhead by controlling the granularity of its parallel execution. Tiling adds some control overhead because the number of loops is doubled, and reduces the amount of parallelism available in the outermost loops. The n initial loops are replaced by n outer loops used to enumerate the tiles and n inner loops used to execute all the iterations within a tile.

Discussion

Introduction

Tiling is useful for most recent parallel computer architectures, with shared or distributed memory, since they all rely on locality to exploit their memory hierarchies and on parallelism to exploit several cores. It is also useful for heterogeneous architectures with hardware accelerators, and for monoprocessors with caches. Unlike many loop transformations, tiling is not a unimodular transformation. Iterations that are geometrically close in the loop nest iteration set are grouped in so-called *tiles* to be executed together atomically. Tiles are also called blocks when their edges are parallel to the axes or more generally when their facets are orthogonal to the base vectors. For instance, the parallel stencil written in C:

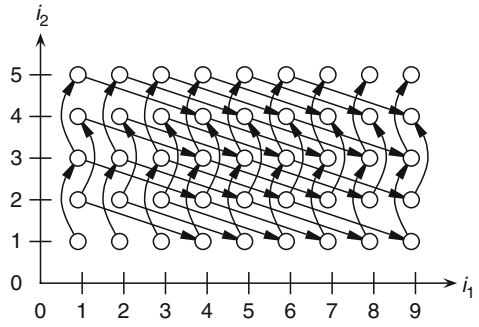
```
for(i1=1;i1<n;i1++)
  for(i2=1;i2<m;i2++)
    a[i1][i2] = 0.2*(b[i1-1][i2]
                    +b[i1][i2]
                    +b[i1+1][i2]
                    +b[i1][i2-1]
                    +b[i1][i2+1])
```

can be transformed into:

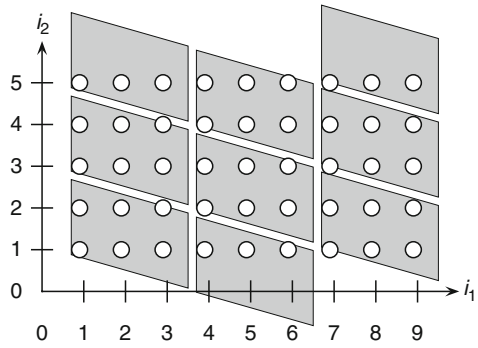
```
#pragma omp parallel for
for(t1=1;t1<n;t1+=b1)
#pragma omp parallel for
  for(t2=1;t2<m;t2+=b2)
    // tile code
    for(i1=t1;i1<min(t1+b1, n);i1++)
      for(i2=t2;i2<min(t2+b2, m);i2++)
        a[i1][i2] = 0.2*(b[i1-1][i2]
                        +b[i1][i2]
                        +b[i1+1][i2]
                        +b[i1][i2-1]
                        +b[i1][i2+1])
```

where `t1` and `t2` are tile coordinates and `b1` and `b2` are the tile or block sizes.

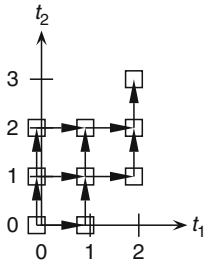
Initially, this tiling transformation was called loop blocking by Allen & Kennedy and tiling by Wolfe [32, 33] before it was extended to slanted tiles under the name of supernode partitioning by Irigoin and Triolet [21]. Wolfe suggested to use systematically the



Tiling. Fig. 1 Iteration space with dependence vectors



Tiling. Fig. 2 Tiled iteration set



Tiling. Fig. 3 Iteration set for tiles, with tile dependence vectors

name tiling as it is short and easy to understand. He uses it in his textbook about program transformations [34]. But loop blocking is still used because it is a proper subset of tiling: see, for instance, Allen and Kennedy [4].

Slanted tiles can be used with these sequential Fortran loops taken from Xue [35]

```
do i1 = 1, 9
  do i2 = 1, 5
    a(i1,i2) = a(i1,i2-2)+a(i1-3,i2+1)
  enddo
enddo
```

whose 2-D iteration set and dependence vectors are shown in Fig. 1 (All figures are taken or derived from Figure 4.1, page 103, of Xue [35] by courtesy of the publisher. Some were adapted or derived to fit the notations used in this entry.). These two loops can be transformed into

```
do t1 = 0, 2
   do t2 = max(it-1,0), (it+4)/2
      do i1 = 3*t1+1, 3*t1+3
         do i2 = max(-t1+2*t2+1,1),
                 min((-t1+6*t2+9)/3,5)
            a(i1,i2) = a(i1,i2-2)
                       +a(i1-3,i2+1)
         enddo
      enddo
   enddo
enddo
```

using slanted tiles with vectors $(3,-1)$ and $(0,2)$ shown in Fig. 2. The sets of integer points in each tile are not slanted in this case, but they are not horizontally aligned as shown by the *grey* areas. The tile iteration set is shown in Fig. 3. Note that Fortran allows negative array indices, which makes references to `a(1,-1)` and `a(-2,2)` possibly legal.

Mathematically speaking, this grouping/blocking is a partition of the loop iteration space that induces a renumbering and a reordering of the iterations. This reordering should not modify the program semantics. Hence, several issues are linked to tiling as to any other program transformations: Why should tiling be considered? What are the legal tilings for a given loop nest? How is an optimal tiling chosen? How is the transformed code generated?

Motivations for Tiling

Tiling has several positive impacts. Depending on the target architecture, it reduces the synchronization overhead, the communication overhead, the cache coherency traffic, the number of external memory accesses, and the amount of memory required to execute a loop nest in an accelerator or a scratch pad memory, or the number of cache misses. Thus, the execution time and/or the energy used to execute the loop

nest are reduced, or the execution with a small memory is made possible.

Tiling also has two possibly negative impacts. The control overhead is increased, if only because the number of loops is doubled, and the amount of parallelism degree is smaller at the tile level because the initial parallelism is partly transferred within each tile and traded for locality and communication and synchronization overheads. The control overhead depends on the code generation phase, especially when partial tiles are needed to cover the boundaries of the iteration set.

Tile selection depends on the target architecture. For shared memory multiprocessors, including multicores, the primary bottleneck is the memory bandwidth and tiling is used to improve the cache hit ratio by reducing the memory footprint, that is, the set of live variables that should be kept in cache, and to reduce the cache coherency traffic. Some array references in the initial code must exhibit some spatial and/or temporal locality for tiling to be beneficial.

Tiling can be applied again, recursively or hierarchically, to increase locality at the different cache levels (L0, L1, L2, L3,…) and even at the register level by using very small tiles compatible with the number of registers. These register tiles are fully unrolled to exploit the registers. The tiling of tiles is also known as multilevel tiling. Tiling can also be used to increase locality at the virtual memory page level as shown in 1969 by McKeller and Coffman in [24].

For vector multiprocessors, the size of the tiles must be large enough to use the vector units efficiently, but small enough for their memory footprint to fit in the local cache, which is one of many trade-offs encountered in tile selection.

For distributed memory multiprocessors, tiling is used to generate automatically distributed code. The tiles are mapped on the processors and the processors communicate data on or close to the tile boundaries. Let p be the edge length of a n-dimensional tile. The idea is to compute $O(p^n)$ values and exchange only $O(p^{n-1})$ values so as to overlap the computation with communication although computations are faster than communications. The amount of memory on each processor is supposed large enough not to be a constraint, but the value of p is adjusted to trade parallelism against communication and synchronization overheads. As mentioned above, these large tiles can be tiled again if locality or parallelism is an issue at the elementary processor level.

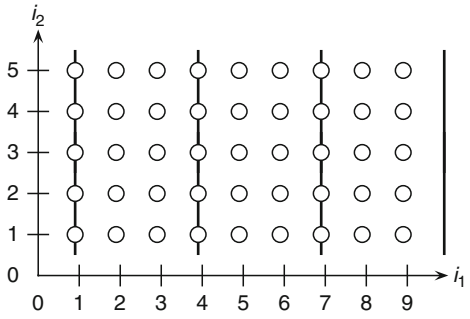
Heterogeneous processors using hardware accelerators, FPGA- or GPGPU- based, require the same kind of trade-offs. Either large tiles must be executed on the accelerator to benefit from its parallel architecture, or small tiles only are possible because of the limited local memory, but in both cases communications between the host and the accelerator must take place asynchronously during the computation. Tiling is used to meet the local memory or vector register size constraints, and to generate opportunities for asynchronous transfers overlapped with the computation.

Multiprocessors System-on-Chip (MPSoC) designed for embedded processing may combine some local internal memories, a.k.a. scratchpad memories, and a global external memory, which make them distributed systems with a global memory. Tiling is used to meet the local memory constraints, but communications between the processors or between the processors and the external global memory must be generated. Other transformations, such as *loop fusion* and *array contraction* (see ►Parallelization, Automatic), are used in combination with tiling to reduce the communication and the execution time. A combination of loop fusion and loop distribution may give a better result than the tiling of fused loops, at least when the scratchpad memory is very small.

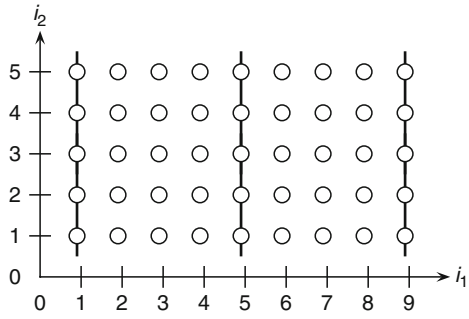
Tiling can also be applied to multidimensional arrays instead of loop nests. This approach is used by *High-Performance Fortran* (see ►High Performance Fortran (HPF)). The code generation is derived from the initial code and from mapping constraints, based, for instance, on the *owner-computes* rule: the computation must be located where the result is stored. This idea may also be applied to speed up array IOs and out-of-core computations.

Finally, tiling can be applied to more general spaces and sets. For instance, Griebl [15] use the ►*Polyhedron Model* to map larger pieces of code to a unique space of large dimension. Sequences of loops can be mapped onto such a space and be tiled globally.

Because of the many machine architectures that can benefit from tiling, numerous papers have been published on the subject. It is important to check what kind of architecture is targeted before reading or comparing them.



Tiling. Fig. 4 First hyperplane partitioning with $h_1 = \left(\frac{1}{3}, 0\right)$ and $o = (1,1)$



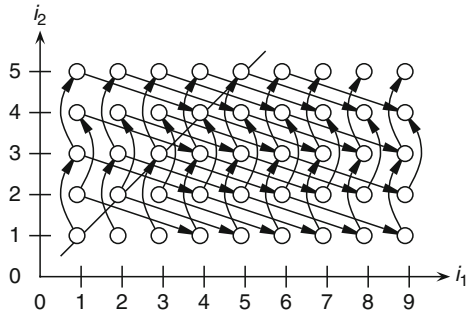
Tiling. Fig. 5 Same hyperplane partitioning with a smaller $h_1 = \left(\frac{1}{4}, 0\right)$

Legality of Tiling

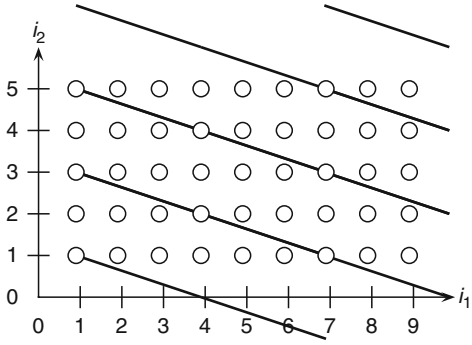
The general definition and legality of tiling were introduced by Irigoin and Triolet who gave sufficient legality conditions in [21]. Necessary conditions were added later by Xue [35].

The basic idea is to use parallel hyperplanes defined by a normal vector h to slice the iteration space Z^n, where n is the number of nested loops, to obtain a partition. The partition of a set E is a set P of nonempty subsets of E, whose two-by-two intersection is always empty and whose union is equal to e. Each slice is mathematically speaking a part (there is no agreement about the naming of elements of P: part, block or cell are used. We chose to use *part*) and two iteration vectors j_1 and j_2 belong to the same part if $\lfloor h.(j_1 - o) \rfloor = \lfloor h.(j_2 - o) \rfloor$, where . denotes the scalar product, $\lfloor\ \rfloor$ the floor function, and o is an offset vector. For instance, using the normal vector $h = \left(\frac{1}{3}, 0\right)$ and the offset $o = (1,1)$, the iteration set of Fig. 1 is partitioned in three parts shown in Fig. 4. Because of this definition, the slices, or parts become larger when the norm of h decreases (see Fig. 5). To make sure that the parts can be executed one after the other, dependence cycles between two parts must be avoided. For instance, the diagonal partition in Fig. 6 creates a cycle between the two subsets. Iteration $(2,1)$ must be executed before iteration $(2,3)$, so the left subset must be executed before the right subset, but iteration $(1,2)$ must be executed before iteration $(4,1)$, which is incompatible.

Cycles between subsets are avoided if each dependence vector d in the loop nest meets the condition $h.d \geq 0$. This condition does depend neither on $\|h\|$, the norm of h, nor on the iteration set, which makes



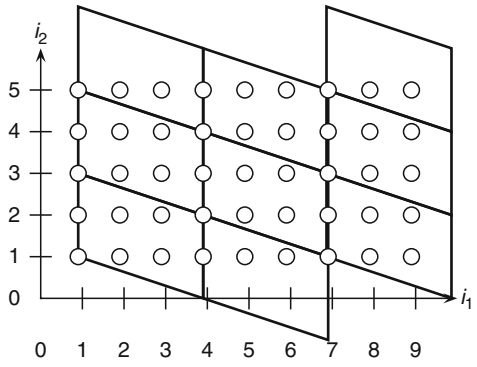
Tiling. Fig. 6 Dependence cycle between two parts



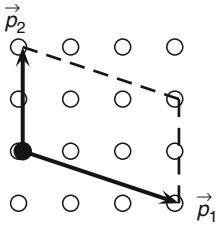
Tiling. Fig. 7 Second hyperplane partitioning with $h_2 = \left(\frac{1}{6}, \frac{1}{2}\right)$ and $o = (1,1)$

all such legal tilings scalable to meet the different needs enumerated above.

Several hyperplane partitionings h_1, h_2,…can be combined to increase the number of parts and reduce their sizes (see Figs. 7 and 8). The vectors h_1, h_2,…are usually grouped (Xue [35] uses H to denote the transposed matrix of H, that is, the h vectors are transformed into affine forms) together in a matrix, H. When the



Tiling. Fig. 8 Combined 2-D hyperplane partitioning



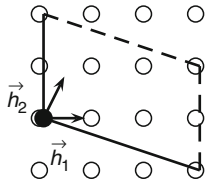
$$P = [\vec{p}_1\ \vec{p}_2] = \begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}$$

Tiling. Fig. 9 Partitioning or clustering matrix

number of different hyperplane families h_i used is equal to the number of nested loops n and when the h_i are linearly independent vectors, the part sizes are bounded, regardless of the iteration set, and all parts of the iteration space are equal up to a translation if H^{-1} is an integer matrix. However, the parts of the iteration set L may differ because of the loop boundaries. See, for instance, tile $(2,3)$, the upper right tile in Fig. 2, which contains only three iterations.

The transpose of H^{-1}, the partitioning matrix P, contains the edges of the tile and its determinant is the number of iterations within each tile. So P (Fig. 9) is easier to visualize than H (Fig. 10). Note that $det(P) = 6$, which is the maximum number of iterations within one tile, as can be seen on Fig. 2.

A tiling H is defined by the number of hyperplane sets used, by the directions of the normal vectors h_i and by their relative norms, that is, the tile shape, and by the number of iterations within a tile, that is, the tile size. The tile shape is defined when $det(H) = 1$. The tile size is controlled by a scaling coefficient. The tiling origin is another parameter impacting mostly the code generation, but also the execution time. Often, a



$$H = [\vec{h}_1\ \vec{h}_2] = \begin{bmatrix} \frac{1}{3} & \frac{1}{6} \\ 0 & \frac{1}{2} \end{bmatrix}$$

Tiling. Fig. 10 Hyperplane matrix H

tiling selection is decomposed into the selection of a shape and the selection of a size and finally the choice of an offset.

The legality conditions are summed up by $H^T R \geq 0$, where R is a matrix made of the *rays of a convex cone* containing all possible dependence vectors d, because the condition $h.d \geq 0$ is convex. The computation of R by a dependence test is explained by Irigoin and Triolet [21] and the dependence cone is one of many approximations of the dependence vector set. When dependences are *uniform* (see ▸Dependences), R can be built directly with the dependence vectors. The valid hyperplanes h belong to another cone, dual of R.

A necessary and sufficient condition, $\lfloor H^T d \rfloor \geq 0$, was introduced by Xue [35] in 1997, where $\geq$ is the lexicographic order, but it does not bring any practical improvement over the previous sufficient condition. Xue also provided an exact legality test based on integer programming and using information about the iteration set $L(j)$, that is, the loop bounds of the initial loop nest.

Note also that the subset of tilings such that $det(H) = 1$ is the set of unimodular transformations and that $H^T d \geq 0$ is their legality condition (see ▸Loop Nest Parallelization).

Tile Selection and Optimal Tiling

Tile selection requires some choice criterion, and optimal tile selection some cost function. The cost function is obviously dependent on the target machine, which makes many optimal tilings possible since many target architectures can benefit from tiling. Also, if the cost function is the execution time, the tiling per se is only part of the compilation scheme. The execution time of one tile depends on the scheduling of the local iterations, for instance because each processor has some

vector capability. It also depends on the tiles previously executed on the same processor, whether a cache or a local memory is used, that is, it depends on the *mapping* of tiles on processor. And finally, the total execution time of the tiled nest depends on the schedule and on the mapping of the tiles on the logical or physical resources, threads, or cores.

In other words, any optimal tiling is optimal with respect to a cost function modeling the execution time or the energy for a given target. Models used to derive analytical optimal solutions often assume that the execution and the communication times are respectively proportional to the computation and communication volumes, which is not realistic, especially with multicores, superword parallelism, and several levels of cache memories. Models may also assume that the number of processors available (because of multicore and multithreaded architectures, the definition of *processor*, virtual or physical, is not well defined. Here, the processor is not a chip, but rather the total number of physical threads in a multicore or the total number of user processes running simultaneously in the machine) is greater than the number of tiles that can be executed simultaneously, which simplifies the mapping of tiles on processors.

When the target architecture has some kind of implicit vector capability, for instance when the cache lines are loaded, a partial tiling with $rank(H) < n$, that is, a set of hyperplane partitionings, may be more effective than a full tiling. Tiles are not longer bounded by the hyperplanes, but they remain bounded by the initial loop nest iteration set.

Because the execution time of real machine becomes more and more complex with the number of transistors used, iterative compilation is used when performance is key. The code is compiled and executed with different tile sizes and the best tile size is retained. Symbolic tiling is useful to speed up the process.

Some decisions can be made at run time. For instance, Rastello et al. [25] use run-time scheduling to speed-up the execution. Note that they overlap the computations and the communications related to *one* tile, which somehow breaks the tile atomicity constraint.

Tiled Code Generation

As for tiling optimality, tile code generation depends on the target machine. A parallel machine requires the mapping and the parallel execution of the tiles. A distributed memory machine also requires communication generation. A processor with a memory hierarchy and/or a vector capability requires loop optimization at the tile level. The minimal requirement is that all iterations of the initial loop nest are performed by the tiled nest.

Let vector j be an iteration of loop nest L, t a tile coordinate, and l the local coordinate of an iteration within a tile t. Since no redundant computations are added by hyperplane partitioning, the relationship between j and (t, l) is a one-to-one mapping from the initial set of iterations $L(j)$ to the new iteration set $T(t, l)$. Ancourt and Irigoin [5] show that an affine relationship can be built between j and (t, l) and that the new loop bounds for t and l can be derived from this relationship and from L when the matrix H is numerically known and when L is a parametric polyhedron, that is, when the loop bounds are affine functions of loop indices and parameters. To simplify array subscript expressions, the code may be generated using t and j instead of t and l. This optimization is used in the code examples given above.

This loop nest generation is sufficient for shared memory machines, although it is better to generate several versions of the tile code, one for the full tiles, and several ones for the partial tiles on the iteration set boundaries, in order to reduce the average control overhead. Multilevel tiling is also used to reduce the overhead due to partial tiles on the boundaries (see the top left and top right tiles in Fig. 2).

This does not specify the mapping of tiles onto threads or cores when the tile parallelism is greater than the number of processors. But locality-aware scheduling lets tiles inherit data from other tiles previously executed on the same thread as suggested by Xue and Huang in [37] who minimize the number of partitioning hyperplanes, that is, the rank of H.

Parametric tiling does not require H to be numerically known at compile time. The tile size, if not the tile shape, can be adjusted at run time or optimized dynamically. A technique is proposed by Renganarayanna et al. [28] for multilevel tiling, and another one by Hartono et al. [18, 23].

Note that parallelism within tiles or across tiles is obtained by *wavefronting*, a unimodular loop transformation (see ▶Loop Nest Parallelization), unless the initial loop nest is fully parallel, in which case cone R is

empty or reduced to $\{0\}$. For instance, the tiles $(1,0)$ and $(1,0)$ on Fig. 3 can be computed in parallel.

Applicability

Tiling and hyperplane partitioning are defined for perfectly nested loops only, but many algorithms, including matrix multiply, are made of non-perfectly nested loops. It is possible to move all non-perfectly nested statements into the loop nest body by adding guards (a.k.a. statement sinking), but these guards must then be carefully moved or removed when the tile code is generated. The issue is tackled directly by Ahmed et al. [3] and Griebl [15, 16], who avoid statement sinking by mapping all statement in another space (see ►Polyhedron Model) and by applying transformations, including tiling, on this space before code generation, and by Hartono et al. [18] who use a polyhedral representation of the code to generate multilevel tilings of imperfectly nested loops. See the Polyhedron Model entry for more information about the mapping of a piece of code onto a polyhedral space.

Tiling can also be applied to loop nests containing commutative and associative reductions, but this does not fit the general legality condition $HR \geq 0$.

Tiling can be applied by the programmer. For instance, 3-D tiling improved with array padding has been used to optimize 3-D PDE solvers and 3-D stencil codes. Tiling has been used to optimize some instances of dynamic programming, the resolution of the heat equation, and even some sparse computations. Because tiling is difficult to apply by hand, source-to-source tilers have been developed.

Finally, Guo et al. suggest in [17] to support tiling at the programming language level, using hierarchically tiled arrays (HTA) to keep the code readable, while letting the programmer be in control.

Related Loop Transformations

The partitioning matrix $P = \left(H^T\right)^{-1}$ can be built step-by-step by a combination of loop skewing, or more generally any unimodular loop transformations, strip-minings (1-D tiling), and loop interchanges. Loop skewing is a unimodular transformation used to change the iteration coordinates and to make loop blocking legal because the new loop nest obtained is *fully permutable*.

In other words, the P matrix is replaced by the product of a diagonal matrix Λ, which defines a rectangular tiling, a.k.a. loop blocking, and of $\frac{1}{det(P)}P$, and the tiling by a sequence of easier transformations. This is advocated by Allen & Kennedy in their textbook [4]. Reducing tiling to blocking via basis changes, for example, using a Smith normal form of H, is also often used to optimize the tile shape and size, but some of the tile shape problem remains and the constraints of the iteration set L usually become more complex.

Strip-mining is a 1-D tiling, a degenerated case of hyperplane partitioning. It is used to adapt the parallelism available to the hardware resources, for instance vector registers.

Loop interchange is a unimodular transformation. Like all unimodular transformations, it is an extreme case of tiling with no tiling effect because $det(H) = 1$, that is, each tile contains only one element. It is often used to increase locality.

Loop unroll-and-jam first unrolls an outer loop by some factor k, that is, it is a strip-mining followed by a full unroll of the new loop. Then, the replicated innermost loops are fused (jammed). This is equivalent to a rectangular hyperplane partitioning with blocking factors $(k, 1, 1, \ldots)$, followed by an unrolling of the tile loop. Unroll-and-jam can be applied to several outer loops with several factors, which again is equivalent to a rectangular tiling with the same factors followed by an unrolling of the tile loops. Unroll-and-jam is used to increase locality and is effective like tiling if some references exhibit temporal locality along outer loops.

Tiling is designed to forbid redundant computations. However, overlap between tiles can reduce communications at the expense of additional computation. Data overlaps are also used to compile ►HPF (High-Performance Fortran) using the owner compute rule.

Finally, tiling is also related to the partitioning of ►systolic arrays, used to fit a large parametric size iteration set on a fixed-size chip.

Future Directions

Although tiling is a powerful transformation by itself, and quite complex to use, it does not include some other key transformations such as loop fusion or loop peeling. Furthermore, the loop body is handled as a unique statement although it may contain sequences, tests, and loops.

So more complex code transformations were advocated in 1991 by Wolf and Lam [31] to optimize parallelism and locality. More recently, Griebl [15] and Bondhugula et al. [8] use the polyhedral framework (see ►Polyhedron Model) to handle each elementary statement individually, at least within static control pieces of code. This is also attempted within gcc with the Graphite plug-in.

Otherwise, it is possible to move away from the complexity of tiling by replacing it with sequences of simpler transformations, including hyperplane partitioning. The difficulty is then to decide which sequence leads to an optimal or at least to a well-performing code.

In case the execution time of each iteration is different or even very different, the tile equality constraint could be lifted up to obtain a nonuniform partitioning. This has already been done in the 1-D case. In such cases, strip-mining is replaced by more complex partitions to map the parallel iterations onto the processors. The larger partitions are executed first to reduce the imbalance between processors at the end without increasing the control overhead at the beginning (see ►Nested loops scheduling).

Related Entries

►Code Generation
►Dependences
►Dependence Abstractions
►Dependence Analysis
►Distributed-Memory Multiprocessor
►HPF (High Performance Fortran)
►Locality of Reference and Parallel Processing
►Loop Nest Parallelization
►Parallelization, Automatic
►Polyhedron Model
►Shared-Memory Multiprocessors
►Systolic Arrays
►Unimodular Transformations

Bibliographic Notes and Further Reading

The best reference about tiling is the book written by Xue [35]. It provides the necessary background on linear algebra and program transformations. It starts with rectangular tilings before moving to slanted, that is, parallelepiped, tilings. Code generation for distributed memory machines and tiling optimizations are finally addressed.

Tile size optimization is addressed explicitly by Coleman and McKinley [12] to eliminate cache capacity, self-interference, and cross-interference misses, that is, for locality improvement.

For shared memory machines, Högstedt et al. [20] introduce the concepts of *idle time* and *rise* for a tiling, and optimal tile shape for a multiprocessor with a memory hierarchy. They propose an algorithm to select an non-rectangular optimal tile shape for a shared memory multiprocessor, with enough processors to use all parallelism available. They assume that the tile execution time is proportional to its volume. Rastello and Robert [26] provide a closed form for the tile shape that minimizes the number of cache misses during the execution of a rectangular tile for a given cache size and for parallel loops, that is, without tiling legality constraints. They also provide a heuristic to optimize the same function for any shape of tiles.

For distributed memory machines, the contributions to the quest for analytic solutions are numerous. Boulet et al. [10] carefully include a question mark in their paper title, *(Pen)-ultimate tiling?*. Hodzic and Shang give several closed forms for the size and the relative side lengths [19]. Xue [36] uses communication/computation overlap and provides a closed form for the optimal tile size.

Tile shapes are restrained to orthogonal shapes by Andonov et al. [7] in order to find an optimal solution. For 2-D iteration spaces, using the BSP model and an unbounded number of processors, Andonov et al. [6] provide closed forms for the optimal tiling parameters and the optimal number of processors.

Agarwal et al. [1, 2] introduce a method for deriving an optimal hyperparallelepiped tiling of iteration spaces for minimal communication in multiprocessors with caches and for distributed shared-memory multiprocessors. More recently, Bondhugula et al. [9] tile sequences of imperfectly nested loops for locality and parallelism. They use an analytical model and integer linear optimization.

Carter et al. introduce hierarchical tiling for superscalar machines [11], but the tuning is handmade. Renganarayanna and Rajopadhye [27] determine optimal tile sizes with a BSP-like model. Strzodka et al. [29] use multilevel tiling to speed up stencil computations

by optimizing simultaneously locality, parallelism and vectorization.

For distributed memory machines, communication code must be generated too. See Ancourt [5], Tang [30], Xue [35], Chapter 6 and 7, and finally Goumas et al. [14], who generate MPI code automatically.

Goumas et al. propose [13] a tile code generation algorithm for parallelepiped tiles. This can be used for general tiles thanks to changes of basis.

Bibliography

1. Agarwal A, Kranz D, Natarajan V (1993) Automatic partitioning of parallel loops for cache-coherent multiprocessors. In: International conference on parallel processing (ICPP), Syracuse University, Syracuse, NY, 16–20 August 1993, vol 1, pp 2–11
2. Agarwal A, Kranz DA, Natarajan V (September 1995) Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 6(9):943–962
3. Ahmed N, Mateev N, Pingali K (2000) Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In: Proceedings of the 14th international conference on supercomputing, Santa Fe, 8–11 May 2000, pp 141–152
4. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures: a dependence-based approach. Morgan-Kaufmann. San Francisco, pp 477–491
5. Ancourt C, Irigoin F (1991) Scanning polyhedra with DO loops. In: Third ACM symposium on principles and practice of parallel programming, Williamsburg, VA, pp 39–50
6. Andonov R, Balev S, Rajopadhye S, Yanev N (July 2001) Optimal semi-oblique tiling. In: Proceedings of the 13th annual ACM symposium on parallel algorithms and architectures, Crete Island, pp 153–162
7. Andonov R, Rajopadhye SV, Yanev N (1998) Optimal orthogonal tiling. In: Proceedings of the fourth international Euro-Par conference on parallel processing, Southampton, 1–4 Sept 1998, pp 480–490
8. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Proceedings of the joint European conferences on theory and practice of software 17th international conference on compiler construction, Budapest, Hungary, 29 March–6 April 2008
9. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (June 2008) A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI 2008. ACM SIGPLAN Not 43(6)
10. Boulet P, Darte A, Risset T, Robert Y (1996) (Pen)-ultimate tiling? Integr: VLSI J 17:33–51
11. Carter L, Ferrante J, Hummel SF (1995) Hierarchical tiling for improved superscalar performance. In: Proceedings of the ninth international symposium on parallel processing, Santa Barbara, 25–28 April 1995, pp 239–245
12. Coleman S, McKinley KS (June 1995) Tile size selection using cache organization and data layout. In: PLDI'95; ACM SIGPLAN Not 30(6):279–290
13. Goumas G, Athanasaki M, Koziris N (2002) Automatic code generation for executing tiled nested loops onto parallel architectures. In: Proceedings of the 2002 ACM symposium on applied computing, Madrid, Spain, 11–14 March 2002
14. Goumas G, Drosinos N, Athanasaki M, Koziris N (November 2006) Message-passing code generation for non-rectangular tiling transformations. Parallel Computing 32(10): 711–732
15. Griebl M (July 2001) On tiling space-time mapped loop nests. In: Proceedings of the 13th annual ACM symposium on parallel algorithms and architectures, Crete Island, pp 322–323
16. Griebl M (June 2004) Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis, Department of Informatics and Mathematics, University of Passau. http://www.fim.uni-passau.de/cl/publications/docs/Gri04.pdf
17. Guo J, Bikshandi G, Fraguela BB, Garzaran MJ, Padua D (2008) Programming with tiles. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City, UT, USA, 20–23 Feb 2008
18. Hartono A, Manikandan Baskaran M, Bastoul C, Cohen A, Krishnamoorthy S, Norris B, Ramanujam J, Sadayappan P (2009) Parametric multi-level tiling of imperfectly nested loops. In: Proceedings of the 23rd international conference on supercomputing, Yorktown Heights, NY, USA, 8–12 June 2009
19. Hodzic E, Shang W (December 2002) On time optimal supernode shape. IEEE Trans Parallel Distrib Syst 13(12):1220–1233
20. Högstedt K, Carter L, Ferrante J (March 2003) On the parallel execution time of tiled loops. IEEE Trans Parallel Distrib Syst 14(3):307–321
21. Irigoin F, Triolet R (1988) Supernode partitioning. In: Fifteenth annual ACM symposium on principles of programming languages, San Diego, CA, pp 319–329
22. Jiménez M, Llabería JM, Fernández A (July 2002) Register tiling in nonrectangular iteration spaces. ACM Trans Program Lang Syst 24(4):409–453
23. Manikandan Baskaran M, Hartono A, Tavarageri S, Henretty T, Ramanujam J, Sadayappan P (2010) Parameterized tiling revisited. In: CGO'10: proceedings of the eighth annual IEEE/ACM international symposium on code generation and optimization, pp 200–209
24. McKeller AC, Coffman EG (1969) The organization of matrices and matrix operations in a paged multiprogramming environment. Commun ACM 12(3):153–165
25. Rastello F, Rao A, Pande S (February 2003) Optimal task scheduling at run time to exploit intra-tile parallelism. Parallel Comput 29(2):209–239
26. Rastello F, Robert Y (May 2002) Automatic partitioning of parallel loops with parallelepiped-shaped tiles. IEEE Trans Parallel Distrib Syst 13(5):460–470
27. Renganarayana L, Rajopadhye S (2004) A geometric programming framework for optimal multi-level tiling. In: Proceedings of the 2004 ACM/IEEE conference on supercomputing, Pittsburgh, PA, 6–12 Nov 2004, p 18

28. Renganarayanan L, Kim D, Rajopadhye S, Strout MM (June 2007) Parameterized tiled loops for free. In: PLDI'07, ACM SIGPLAN Not 42(6)
29. Strzodka R, Shaheen M, Pajak D, Seidel H-P (2010) Cache oblivious parallelograms in iterative stencil computations. In: ICS'10: proceedings of the 24th ACM international conference on supercomputing, Tsukuba, Japan, pp 49–59
30. Tang P, Xue J (2000) Generating efficient tiled code for distributed memory machines. Parallel Comput 26(11):1369–1410
31. Wolf ME, Lam MS (October 1991) A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans Parallel Distrib Syst 2(4):452–471
32. Wolfe MJ (1987) Iteration space tiling for memory hierarchies. In: Rodrigue G (ed) Parallel processing for scientific computing. SIAM, Philadelphia, pp 357–361
33. Wolfe MJ (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE conference on supercomputing, Reno, NV, 12–17 Nov 1989, pp 655–664
34. Wolfe MJ (1995) High performance compilers for parallel computing. Addison-Wesley Longman, Boston
35. Xue J (2000) Loop tiling for parallelism. Kluwer, Boston
36. Xue J, Cai W (June 2002) Time-minimal tiling when rise is larger than zero. Parallel Comput 28(6):915–939
37. Xue J, Huang C-H (December 1998) Reuse-driven tiling for improving data locality. Int J Parallel Program 26(6):671–696

Titanium

Katherine Yelick[1], Susan L. Graham[2], Paul Hilfinger[2], Dan Bonachea[3], Jimmy Su[2], Amir Kamil[2], Kaushik Datta[2], Phillip Colella[2], Tong Wen[2]
[1]University of California at Berkeley and Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]University of California, Berkeley, CA, USA
[3]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Definition

Titanium is a parallel programming language designed for high-performance scientific computing. It is based on Java™ and uses a Single Program Multiple Data (SPMD) parallelism model with a Partitioned Global Address Space (PGAS).

Discussion

Introduction

Titanium is an explicitly parallel dialect of Java™ designed for high-performance scientific programming [14, 15]. The Titanium project started in 1995, at a time when custom supercomputers were losing market share to PC clusters. The motivation was to create a language design and implementation that would enable portable programming for a wide range of parallel platforms by striking an appropriate balance between expressiveness, user-provided information about concurrency and memory locality, and compiler and runtime support for parallelism. The goal was to design a language that could be used for high performance on some of the most challenging applications, such as those with adaptivity in time and space, unpredictable dependencies, and sparse, hierarchical, or pointer-based data structures.

The strategy was to build on the experience of several Partitioned Global Address Space (PGAS) languages, but to design a higher-level language offering object orientation with strong typing and safe memory management in the context of applications requiring high performance and scalable parallelism. Titanium uses Java as the underlying base language, but is neither a strict superset nor subset of that language. Titanium adds general multidimensional arrays, support for extending the value types in the language, and an unordered loop construct. In place of Java threads, which are used for both program structuring and concurrency, Titanium uses a static thread model with a partitioned address space to allow for locality optimizations.

Titanium's Parallelism Model

Titanium uses a Single Program Multiple Data (SPMD) parallelism model, which is familiar to users of message-passing models. The following simple Titanium program illustrates the use of built-in methods **Ti.numProcs()** and **Ti.thisProc()**, which query the environment for the number of threads (or processes) and the index within that set of the executing thread. The example prints these indices in arbitrary order. The number of Titanium threads need not be equal to the number of physical processors, a feature that is often useful when debugging parallel code on single-processor machines. However, high-performance runs typically use a one-to-one mapping between Titanium threads and physical processors.

```
class HelloWorld {
  public static void main (String [] argv) {
```

```
    System.out.println("Hello from proc " +
      Ti.thisProc() + " out of " + Ti.numProcs());
  }
}
```

Titanium supports Java's synchronized blocks, which are useful for protecting asynchronous accesses to shared objects. Because many scientific applications use a bulk-synchronous style, Titanium also has a barrier-synchronization construct, **Ti.barrier()**, as well as a set of collective communication operations to perform broadcasts, reductions, and scans. A novel feature of Ti- tanium's parallel execution model is that barriers must be textually aligned in the program – not only must all threads reach a barrier before any one of them may proceed, but they must all reach the same textual barrier. For example, the following program is not legal in Titanium:

```
if (Ti.thisProc() == 0) Ti.barrier();
  //illegal barrier
else Ti.barrier();//illegal barrier
```

Aiken and Gay developed the static analysis the compiler uses to enforce this alignment restriction, based on two key concepts [1]:

- A *single method* is one that must be invoked by all threads collectively. Only single methods can execute barriers.
- A *single-valued expression* is an expression that is guaranteed to take on the same sequence of values on all processes. Only single-valued expressions may be used in conditional expressions that affect which barriers or single-method calls get executed.

The compiler automatically determines which methods are single by finding barriers or (transitively) calls to other single methods. Single-valued expressions are required in statements that determine the flow of control to barriers, ensuring that the barriers are executed by all threads or by none. Titanium extends the Java type system with the single qualifier. Variables of single-qualified type may only be assigned values from single-valued expressions. Literals and values that have been broadcast are simple examples of single-valued expressions. The following example illustrates these concepts. Because the loop contains barriers, the expressions in the for-loop header must be single-valued. The compiler can check that property statically, since the variables are declared single and are assigned from single-valued expressions.

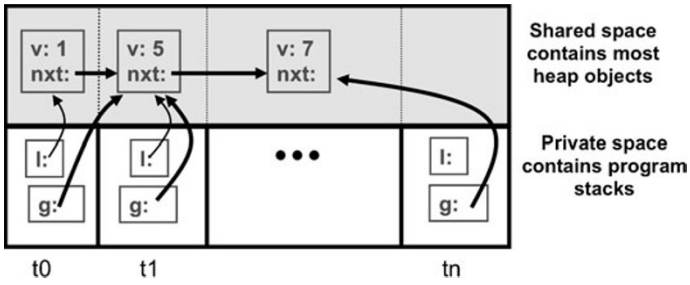
```
int single allTimestep = 0;
int single allEndTime = broadcast
   inputTimeSteps from 0;
for (; allTimestep < allEndTime;
   allTimestep)++{
  < read values belonging to other threads >
  Ti.barrier();
  < compute new local values >
  Ti.barrier();
}
```

Barrier analysis is entirely static and provides compile-time prevention of barrier-based deadlocks. It can also be used to improve the quality of concurrency analysis used in optimizations. Single qualification on variables and methods is a useful form of program design documentation, improving readability by making replicated quantities and collective methods explicitly visible in the program source and subjecting these properties to compiler enforcement.

Titanium's Memory Model

The two basic mechanisms for communicating between threads are accessing shared variables and sending messages. Shared memory is generally considered easier to program, because communication is one-sided: Threads can access shared data at any time without interrupting other threads, and shared data structures can be directly represented in memory. Titanium is based on a Partitioned Global Address Space (PGAS) model, which is similar to shared memory but with an explicit recognition that access time is not uniform. As shown in Fig. 1, memory is partitioned such that each partition has affinity to one thread. Memory is also partitioned orthogonally into private and shared memory, with stack variables living in private memory, and heap objects, by default, living in the shared space. A thread may access any variable that resides in shared space, but has fast access to variables in its own partition. Objects created by a given thread will reside in its own part of the memory space.

Titanium statically makes an explicit distinction between local and global references: A local reference must refer to an object within the same thread partition, while a global reference may refer to either a remote or



Titanium. Fig. 1 Titanium's partitioned global address space memory model

local partition. In Fig. 1, instances of **l** are local references, whereas **g** and **nxt** are global references and can therefore cross partition boundaries. The motivation for this distinction is performance. Global references are more general than local ones, but they often incur a space penalty to store affinity information and a time penalty upon dereference to check whether communication is required. References in Titanium are global by default, but may be designated local using the local type qualifier. The compiler performs type inference to automatically label variables as local [10].

The partitioned memory model is designed to scale well on distributed memory platforms without the need for caching of remote data and the associated coherence protocols. Titanium also runs well on shared memory multiprocessors and uniprocessors, where the partitioned-memory model may not correspond to any physical locality on the machine and the global references generally incur no overhead relative to local ones. Naively written Titanium programs may ignore the partitioned-memory model and, for example, allocate all data structures in one thread's shared memory partition or perform fine-grained accesses on remote data. Such programs would run correctly on any platform but would likely perform poorly on a distributed memory platform. In contrast, a program that carefully manages its data-structure partitioning and access behavior in order to scale well on distributed memory hardware is likely to scale well on shared memory platforms as well. The partitioned model provides the ability to start with a functional, shared memory style code and incrementally tune performance for distributed memory hardware by reorganizing the affinity of key data structures or adjusting access patterns in program bottlenecks to improve communication performance.

Titanium Arrays

Java arrays do not support sub-array objects that are shared with larger arrays, nonzero base indices, or true multidimensional arrays. Titanium retains Java arrays for compatibility, but adds its own multidimensional array support, which provides the same kinds of sub-array operations available in Fortran 90. Titanium arrays are indexed by integer tuples known as points and built on sets of points, called domains. The design is taken from that of a language for Finite Different Calculations, FIDIL, designed by Colella and Hilfinger [7]. Points and domains are first-class entities in Titanium – they can be stored in data structures, specified as literals, passed as values to methods, and manipulated using their own set of operations. For example, NAS multigrid (MG) benchmark requires a 256^3 grid. The problem has periodic boundaries, which are implemented using a one-deep layer of surrounding ghost cells, resulting in a 258^3 grid. Such a grid can be constructed with the following declaration:

```
double [3d] gridA
  = new double [[-1,-1,-1]:[256,256,256]];
```

The 3D Titanium array **gridA** has a rectangular index set that consists of all points [i, j, k] with integer coordinates such that $-1 \leqslant i, j, k \leqslant 256$. Titanium calls such an index set a rectangular domain of Titanium type **RectDomain**, since all the points lie within a rectangular box. Titanium also has a type **Domain** that represents an arbitrary set of points, but Titanium arrays can only be built over **RectDomains**. Titanium arrays may start at an arbitrary base point, as the example with a [−1, −1, −1] base shows. In this example, the grid was designed to have space for ghost regions, which are

all the points that have either −1 or 256 as a coordinate. On machines with hierarchical memory systems, **gridA** resides in memory with affinity to exactly one process, namely the process that executes the above statement. Similarly, objects reside in a single logical memory space for their entire lifetime (there is no transparent migration of data), though they are accessible from any process in the parallel program.

The power of Titanium arrays stems from array operators that can be used to create alternative views of an array's data, without an implied copy of the data. While this is useful in many scientific codes, it is especially valuable in hierarchical grid algorithms like Multigrid and Adaptive Mesh Refinement (AMR). In a Multigrid computation on a regular mesh, there is a set of grids at various levels of refinement, and the primary computations involve sweeping over a given level of the mesh performing nearest neighbor computations (called stencils) on each point. To simplify programming, it is common to separate the interior computation from computation at the boundary of the mesh, whether those boundaries come from partitioning the mesh for parallelism or from special cases used at the physical edges of the computational domain. Since these algorithms typically deal with many kinds of boundary operations, the ability to name and operate on sub-arrays is useful.

Domain Calculus

Titanium's domain calculus operators support subarrays both syntactically and from a performance standpoint. The tedious business of index calculations and array offsets has been migrated from the application code to the compiler and runtime system. For example, the following Titanium code creates two blocks that are logically adjacent, with a boundary of ghost cells around each to hold values from the adjacent block. The shrink operation creates a view of **gridA** by shrinking its domain on all sides, but does not copy any of its elements. Thus, **gridAInterior** will have indices from [0, 0, 0] to [255, 255, 255] and will share corresponding elements with **gridA**. The **copy** operation in the last line updates one plane of the ghost region in **gridB** by copying only those elements in the intersection of the two arrays. Operations on Titanium arrays such as **copy** are not opaque method calls to the Titanium compiler. The compiler recognizes and treats such operations specially, and thus can apply optimizations to them, such as turning blocking operations into non-blocking ones.

```
double [3d] gridA =
  new double [[-1,-1,-1]:[256,256,256]];
double [3d] gridB =
  new double [[-1,-1,256]:[256,256,512]];
//define interior without creating a copy
double [3d] gridAInterior = gridA.shrink(1);
//update overlapping ghost cells
   from neighboring block
//by copying values from gridA to gridB
gridB.copy(gridAInterior);
```

The above example appears in a NAS MG implementation in Titanium [4], except that **gridA** and **gridB** are themselves elements of a higher-level array structure. The copy operation as it appears here performs contiguous or noncontiguous memory copies, and may perform interprocessor communication when the two grids reside in different processor memory spaces. The use of a global index space across distinct array objects (made possible by the arbitrary index bounds of Titanium arrays) makes it easy to select and copy the cells in the ghost region, and is also used in the more general case of adaptive meshes.

Unordered Loops, Value Types, and Overloading

The *foreach* construct provides an unordered looping construct designed for iterating through a multidimensional space. In the foreach loop below, the point **p** plays the role of a loop index variable.

```
foreach (p in gridAInterior.domain()) {
  gridB[p] = applyStencil(gridAInterior, p);
}
```

The **applyStencil** method may safely refer to elements that are one point away from **p**, since the loop is over the interior of a larger array.

This one loop concisely expresses an iteration over a multidimensional domain that would correspond to a multi-level loop nest in other languages. A common class of loop bounds and indexing errors is avoided by having the compiler and runtime system automatically manage the iteration boundaries for the multidimensional traversal. The foreach loop is a purely serial iteration construct – it is not a data-parallel construct. In addition, if the order of loop execution is irrelevant to a computation, then using a foreach loop

to traverse the points in a domain explicitly allows the compiler to reorder loop iterations to maximize performance – for instance, by performing automatic cache blocking and tiling optimizations [12]. It also simplifies bounds-checking elimination and array access strength-reduction optimizations.

The Titanium immutable class feature provides language support for defining application-specific primitive types (often called "lightweight" or "value" classes), allowing the creation of user-defined unboxed objects, analogous to C structs. Immutables provide efficient support for extending the language with new types which are manipulated and passed by value, avoiding pointer-chasing overheads which would otherwise be associated with the use of tiny objects in Java.

One compelling example of the use of immutables is for defining a Complex number class, which was used in a Titanium implementation of the NAS FT benchmark.

Titanium also allows for operator overloading, a feature that was strongly desired by application developers on the team, and was used in the FT example to simplify the expressions on complex values.

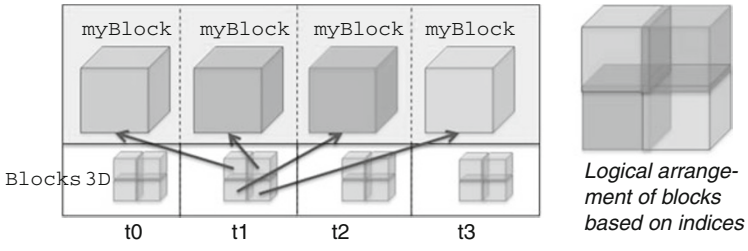
Distributed Arrays

Titanium also supports the construction of distributed array data structures, which are built from local pieces rather than declared as distributed types. This reflects the design emphasis on adaptive and sparse data structures in Titanium, rather than the simpler "regular array" computations that could be supported with simpler flat arrays. The general pointer-based distribution mechanism combined with the use of arbitrary base indices for arrays provides an elegant and powerful mechanism for shared data.

The following code is a portion of the parallel Titanium code for a multigrid computation. It is run on every processor and creates the **blocks3D distributed** array, which can access any processor's portion of the grid.

```
Point< 3 > startCell =
myBlockPos * numCellsPerBlockSide;
Point< 3 > endCell = startCell + (numCellsPerBlock
  Side - [1,1,1]);
double [3d] myBlock =
new double[startCell:endCell];
//"blocks" is used to create "blocks3D" array
double [1d] single [3d] blocks =
  new double [0:(Ti.numProcs()-1)] single [3d];
blocks.exchange(myBlock);
//create local "blocks3D" array
double [3d] single [3d] blocks3D =
  new double [[0,0,0]:numBlocksInGridSide -
       [1,1,1]]single [3d];
//map from "blocks" to "blocks3D" array
foreach (p in blocks3D.domain())
  blocks3D[p] = blocks[procForBlockPosition(p)];
```

Each processor computes its start and end indices by performing arithmetic operations on **Points**. These indices are used to create a local **myBlock** array. Every processor also allocates its own 1D array **blocks**. Then, by combining the **myBlock** arrays using the exchange operation, **blocks** becomes a distributed data structure. As shown in Fig. 2, the exchange operation performs an all-to-all broadcast and stores each processor's contribution in the corresponding element of its local blocks array. To create a more natural mapping, a 3D processor array is used, with each element containing a reference to a particular local block. By using global indices in the local block – meaning that each block has a different set of indices that overlap only in the area of ghost regions – the copy operations described above can be used to update the ghost cells. The generality of Titanium's distributed data structures is not fully utilized in the example of a uniform mesh, but in an adaptive block structured mesh, a union of rectangles can be used to



Titanium. Fig. 2 Distributed 3D array in titanium's PGAS address space. The pointers in the blocks3D array are shown only for thread t1 for simplicity

fill a spatial area, and the global indexing and global address space used to simplify much more complicated ghost region updates.

Implementation Techniques and Research

The Titanium compiler translates Titanium code into C code, and then hands that code off to a C compiler to be compiled and linked with the Titanium runtime system and, in the case of distributed memory back ends, with the GASNet communication system [5]. The choice of C as a target was made to achieve portability, and produces reasonable performance without the overhead of a virtual machine. GASNet is a one-sided communication library that is used within a number of other PGAS language implementations, including Co-Array Fortran, Chapel, and multiple UPC implementations. GASNet is itself designed for portability, and it runs on top of Ethernet (UDP) and MPI, but there are optimized implementations for most of the high-speed networks that are used in clusters and supercomputers designs. Titanium can also run on shared memory systems using a runtime layer based on POSIX Threads, and on combinations of shared and distributed memory by combining this with GASNet. Titanium, like Java, is designed for memory safety, and the Titanium runtime system includes the Boehm-Weiser garbage collector for shared memory code. To handle distributed memory environments, the runtime system tracks references that leak to remote nodes, but also adds a scalable region-based memory management concept to the language along with compiler analysis [5].

Aggressive program analysis is crucial for effective optimization of parallel code. In addition to serial loop optimizations [12] and some standard optimizations to reduce the size and complexity of generate C code, the compiler performs a number of novel analyses on parallelism constructs. For example, information about what sections of code may operate concurrently is useful for many optimizations and program analyses. In combination with alias analysis, it allows the detection of potentially erroneous race conditions, the removal of unnecessary synchronization operations, and the ability to provide stronger memory consistency guarantees. Titanium's textually aligned barriers divide the code into independent phases, which can be exploited to improve the quality of concurrency analysis. The single-valued expressions are also used to improve concurrency analysis on branches. These two features allow a simple graph encoding of the concurrency in a program based on its control-flow graph. We have developed quadratic-time algorithms that can be applied to the graph in order to determine all pairs of expressions that can run concurrently.

Alias analysis identifies pointer variables that may, must, or cannot reference the same object. The Titanium compiler uses alias analysis to enable other analyses (such as locality and sharing analysis), and to find places where it is valid to introduce restrict qualifiers in the generated C code, enabling the C compiler to apply more aggressive optimizations. The Titanium compiler's alias analysis is a Java derivative of Andersen's points-to analysis with extensions to handle multiple threads. The modified analysis is only a constant factor slower than the sequential analysis, and its running time is independent of the number of runtime threads.

Application Experience

A number of benchmarks and larger applications have been written in Titanium, starting with some of the NAS Benchmarks [4]. In addition, Yau developed a distributed matrix library that supports blocked-cyclic layouts and implemented Cannon's Matrix Multiplication algorithm, Cholesky and LU factorization (without pivoting). Balls and Colella built a 2D version of their Method of Local Corrections algorithm for solving the Poisson equation for constant coefficients over an infinite domain [2]. Bonachea, Chapman, and Putnam built a Microarray Optimal Oligo Selection Engine for selecting optimal oligonucleotide sequences from an entire genome of simple organisms, to be used in microarray design. The most ambitious efforts have been applications frameworks for Adaptive Mesh Refinement (AMR) algorithms and Immersed Boundary Method simulations [6] by Tong Wen and Ed Givelberg, respectively. In both cases, these application efforts have taken a few years and were preceded by implementations of Titanium codes for specific problem instances, e.g., AMR Poisson by Luigi Semenzato, AMR gas dynamics [11] by Peter McCorquodale and Immersed Boundaries for simulation of the heart by Armando Solar-Lezama and cochlea by Ed Givelberg, with various optimization and analysis efforts by Sabrina Merchant, Jimmy Su, and Amir Kamil.

The performance results show good scalability on the applications problems on up to hundreds of separate distributed memory nodes, and performance that is in some cases comparable to applications written in C++ or FORTRAN with message passing. The compiler is a research prototype and does not have all of the static and dynamic optimizations one would expect from a commercial compiler, but even serial running-time comparisons show competitive performance. No formal productivity studies involving humans have been done, but a variety of case studies have shown that the global address space combined with a powerful multi-dimensional array abstraction and the data abstraction support derived from Java leads to code that is elegant and concise.

Related Entries

►Coarray Fortran
►PGAS (Partitioned Global Address Space) Languages
►UPC

Bibliography

1. Aiken A, Gay D (1998) Barrier inference. In: Principles of programming languages, San Diego, CA
2. Balls GT, Colella P (2002) A finite difference domain decomposition method using local corrections for the solution of Poisson's equation. J Comput Phys 180(1):25–53
3. Bonachea D (2002) GASNet specification. Technical report CSD-02-1207, University of California, Berkeley
4. Datta K, Bonachea D, Yelick K (2005) Titanium performance and potential: an NPB experimental study. In: 18th international workshop on languages and compilers for parallel computing (LCPC). Hawthorne, NY, October 2005
5. Gay D, Aiken A (2001) Language support for regions. In: SIGPLAN conference on programming language design and implementation. Washington, DC, pp 70–80
6. Givelberg E, Yelick K Distributed immersed boundary simulation in titanium. http://titanium.cs.berkeley.edu, 2003
7. Hilfinger PN, Colella P (1989) FIDIL: a language for scientific processing. In: Grossman R (ed) Symbolic computation: applications to scientific computing. SIAM, Philadelphia, pp 97–138
8. Kamil A, Yelick K (2007) Hierarchical pointer analysis for distributed programs. Static Analysis Symposium (SAS), Kongens Lyngby, Denmark, August 22–24, 2007
9. Kamil A, Yelick K (2010) Enforcing textual alignment of collectives using dynamic checks. In: 22nd international workshop on languages and compilers for parallel computing (LCPC), October 2009. Also appears in Lecture notes in computer science, vol 5898. Springer, Berlin, pp 368–382. DOI: 10.1007/978-3-642-13374-9
10. Liblit B, Aiken A (2000) Type systems for distributed data structures. In: The 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), Boston, January 2000
11. McCorquodale P, Colella P (1999) Implementation of a multilevel algorithm for gas dynamics in a high-performance Java dialect. In: International parallel computational fluid dynamics conference (CFD'99)
12. Pike G, Semenzato L, Colella P, Hilfinger PN (1999) Parallel 3D adaptive mesh refinement in Titanium. In: 9th SIAM conference on parallel processing for scientific computing, San Antonio, TX, March 1999
13. Su J, Yelick K (2005) Automatic support for irregular computations in a high-level language. In: 19th International Parallel and Distributed Processing Symposium (IPDPS)
14. Yelick K, Hilfinger P, Graham S, Bonachea D, Su J, Kamil A, Datta K, Colella P, Wen T (2007) Parallel languages and compilers: perspective from the titanium experience. Int J High Perform Comput App 21:266–290
15. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: a high-performance Java dialect. Concur: Pract Exp 10:825–836

Web Documentation Bibliography

GASNet Home Page. http://gasnet.cs.berkeley.edu/
Titanium Project Home Page at http://titanium.cs.berkeley.edu.

TLS

►Speculation, Thread-Level
►Speculative Parallelization of Loops

TOP500

Jack Dongarra, Piotr Luszczek
University of Tennessee, Knoxville, TN, USA

Definition

TOP500 is a list of 500 fastest supercomputers in the world ranked by their performance achieved from running the LINPACK Benchmark. The list is assembled twice a year and officially presented at two supercomputing conferences: one in Europe and one in the USA. This list has been put together since 1993.

Discussion

Statistics on high-performance computers are of major interest to manufacturers, users, and potential users. These people wish to know not only the number of systems installed, but also the location of the various supercomputers within the high-performance computing community and the applications for which a computer system is being used. Such statistics can facilitate the establishment of collaborations, the exchange of data and software, and provide a better understanding of the high-performance computer market.

Statistical lists of supercomputers are not new. Every year since 1986, Hans Meuer has published system counts of the major vector computer manufacturers, based principally on those at the Mannheim Supercomputer Seminar. In the early 1990s, a new definition of supercomputer was needed to produce meaningful statistics. After experimenting with metrics based on processor count in 1992, the idea was born at the University of Mannheim to use a detailed listing of installed systems as the basis. In early 1993, Jack Dongarra was convinced to join the project with the LINPACK benchmark. A first test version was produced in May 1993. Today the TOP500 list is compiled by Hans Meuer of the University of Mannheim, Germany, Jack Dongarra of the University of Tennessee, Knoxville, and Erich Strohmaier and Horst Simon of NERSC/Lawrence Berkeley National Laboratory.

New statistics are required that reflect the diversification of supercomputers, the enormous performance difference between low-end and high-end models, the increasing availability of massively parallel processing (MPP) systems, and the strong increase in computing power of the high-end models of workstation suppliers (SMP).

To provide a new statistical foundation, the authors of the TOP500 decided in 1993 to assemble and maintain a list of the 500 most powerful computer systems. The list is updated twice a year. The first of these updates always coincides with the International Supercomputer Conference in June (submissions are accepted until April 15), the second one is presented in November at the IEEE Super Computer Conference in the USA (submissions are accepted until October 1st). The list is assembled with the help of high-performance computer experts, computational scientists, and manufacturers.

In the present list (called the TOP500), computers are ranked by their performance on the LINPACK Benchmark. The list is freely available at http://www.top500.org/ where users can create additional sublists and statistics out of the TOP500 database on their own.

The main objective of the TOP500 list is to provide a ranked list of general-purpose systems that are in common use for high-end applications. A general-purpose system is expected to be able to solve a range of scientific problems.

The TOP500 list shows the 500 most powerful commercially available computer systems known. To keep the list as compact as possible, only a part of the information is shown:

- Nworld – Position within the TOP500 ranking
- Manufacturer – Manufacturer or vendor
- Computer – Type indicated by manufacturer or vendor
- Installation Site – Customer
- Location – Location and country
- Year – Year of installation/last major update
- Field of Application
- #Proc. – Number of processors (Cores)
- Rmax – Maximal LINPACK performance achieved
- Rpeak – Theoretical peak performance
- Nmax – Problem size for achieving Rmax
- N1/2 – Problem size for achieving half of Rmax

In the TOP500 List table, the computers are ordered first by their Rmax value. In the case of equal performances (Rmax value) for different computers, a choice was made to order by Rpeak. For sites that have the same computer, the order is by memory size and then alphabetically.

Method of Solution

In an attempt to obtain uniformity across all computers in performance reporting, the algorithm used in solving the system of equations ($Ax = b$, for a dense matrix A) in the benchmark procedure must conform to LU factorization with partial pivoting. In particular, the operation count for the algorithm must be $2/3\ n^3 + O(n^2)$ double point floating point operations (Rmax value is computed by dividing this count by the time taken to solve). Here a floating point operation is an addition or multiplication of 64-bit operands. This excludes the use of a fast matrix multiply algorithm like "Strassen's

Method" or algorithms which compute a solution in a precision lower than full precision (64 bit floating point arithmetic) and refine the solution using an iterative approach. This is done to provide a comparable set of performance numbers across all computers. Submitters of the results are free to implement their own solution as long as the above criteria are met. A reference implementation of the benchmark called HPL (High Performance LINPACK) is provided at: http://www.netlib.org/benchmark/hpl/. In addition to satisfying the rules, HPL also verifies the result with a numerical check of the obtained solution.

Restrictions

The main objective of the TOP500 list is to provide a ranked list of general-purpose systems that are in common use for high-end applications. The authors of the TOP500 reserve the right to independently verify submitted LINPACK Benchmark [1] results, and exclude systems from the list, which are not valid or not general purpose in nature. A system is considered to be of general purpose if it is able to be used to solve a range of scientific problems. Any system designed specifically to solve the LINPACK benchmark problem or have as its major purpose the goal of a high TOP500 ranking will be disqualified. The systems in the TOP500 list are expected to be persistent and available for use for an extended period of time. In that period, it is allowed to submit new results which will supersede any prior submissions. Thus, an improvement over time is allowed. The TOP500 authors will reserve the right to deny inclusion in the list if it is suspected that the system violates these conditions.

The TOP500 List keepers can be reached by sending email to info at top500.org.

The TOP500 list can be found at www.top500.org.

Related Entries

►Benchmarks
►HPC Challenge Benchmark
►LINPACK Benchmark
►Livermore Loops

Bibliography

1. Dongarra JJ, Luszczek P, Petitet A (2003) The LINPACK benchmark: past, present, and future. Concurr Comput Pract Exp 15:1–18

Topology Aware Task Mapping

Abhinav Bhatele
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Synonyms

Graph embedding; MPI process mapping

Definition

Topology aware task mapping refers to the mapping of communicating parallel objects, tasks, or processes in a parallel application on nearby physical processors to minimize network traffic, by considering the communication of the objects or tasks and the interconnect topology of the machine.

Discussion

Introduction

Processors in modern supercomputers are connected together using a variety of interconnect topologies: meshes, tori, fat-trees, and others. Increasing size of the interconnect leads to an increased sharing of resources (network links and switches) among messages and hence network contention. This can potentially lead to significant performance degradation for certain classes of parallel applications. Sharing of links can be avoided by minimizing the distance traveled by messages on the network. This is achieved by mapping communicating objects or tasks on nearby physical processors on the network topology and is referred to as topology aware task mapping. Topology aware mapping is a technique to minimize communication traffic over the network and hence optimize performance of parallel programs. It is becoming increasingly relevant for obtaining good performance on current supercomputers.

The general mapping problem is known to be NP-hard [1, 2]. Apart from parallel computing, topology aware mapping also has applications in graph embedding in mathematics and VLSI circuit design. The problem of embedding one graph on another while minimizing some metric has been well studied in mathematics. Layout of VLSI circuits to minimize length of the longest wire is another problem that requires mapping of one grid on to another. However, the problems

to be tackled are different in several aspects in parallel computing from mathematics or circuit layout. For example, in VLSI, the size of the host graph can be larger than that of the guest graph whereas, in parallel computing, typically, the host graph is equal to or smaller than the guest graph.

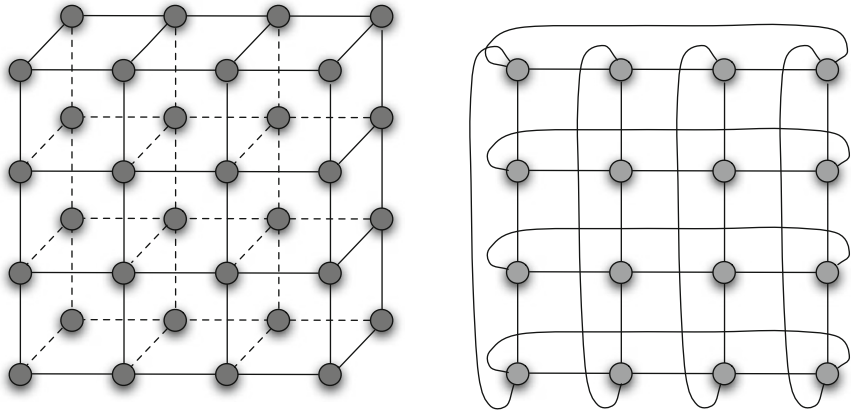
Research on topology aware mapping in parallel computing began in the 1980s with a paper by Bokhari [1]. Work in this area has primarily involved the development of heuristics that target different mapping scenarios. Heuristics typically provide close to optimal solutions in a reasonable time. Arunkumar et al. [3] categorize various heuristic techniques into – deterministic, randomized, and random start heuristics. Over the years, specific techniques better suited for certain architectures were developed – for hypercubes and array processors in the 1980s and meshes and tori in the 1990s. In the recent years, developers of certain parallel applications have also developed application specific mapping techniques to map their codes on to modern supercomputers [4–6].

Prior to a survey of the various heuristic techniques for task mapping, a brief description of the existing interconnect topologies and the kinds of communication graphs that are prevalent in parallel applications is essential.

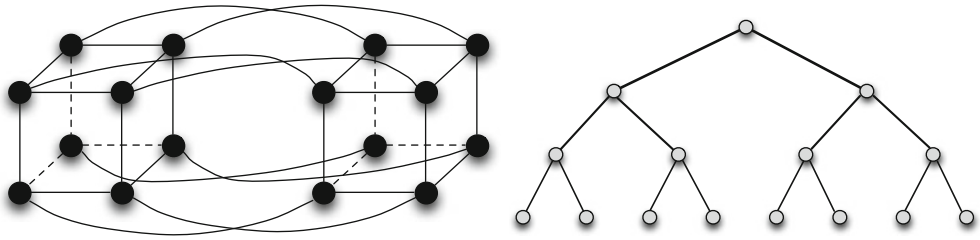
Interconnect Topologies

Various common and radical topologies have been deployed in supercomputers, ranging from hypercubes to fat-trees to three-dimensional tori and meshes. They can be divided into two categories:

1. *Direct networks*: In direct networks, each processor is connected to a few other processors directly. A message travels from source to destination by going through several links connecting the processors. Hypercubes, tori, meshes, etc., are all examples of direct networks (see Figs. 1 and 2). Several modern supercomputers currently have a three-dimensional (3D) mesh or torus interconnect topology. IBM Blue Gene/L and Blue Gene/P machines are 3D tori built from blocks of a torus of size $8 \times 8 \times 8$ nodes. Cray XT machines (XT5 and XE6) are also 3D tori. The primary difference between IBM and Cray machines is that on IBM machines, each allocated job partition



Topology Aware Task Mapping. Fig. 1 A three-dimensional mesh and a two-dimensional torus



Topology Aware Task Mapping. Fig. 2 A four-dimensional hypercube and a three-level fat-tree network

is a contiguous mesh or torus. However, on Cray machines, nodes are randomly selected for a job and do not constitute a complete torus.

2. *Indirect networks*: Indirect networks have switches which route the messages to the destination. No two processors are connected directly and messages always have to go through switches to reach their destination. Fat-tree networks are examples of indirect networks (see Fig. 2). Infiniband, IBM's Federation interconnect and SGI Altix machines are examples of fat-tree networks. LANL's RoadRunner also has a fat-tree network.

Some of these networks benefit from topology aware task mapping more than others. A significant percentage of the parallel machines in the 1980s had a hypercube interconnect and hence, much of the research then was directed toward such networks. More recent work involves optimizing applications on 3D meshes and tori.

Communication Graphs

Tasks in parallel applications can interact in a variety of ways in terms of the specific communication partners, number of communicators, global versus localized communication, etc. All applications can be classified into a few different categories based on different parameters governing the communication patterns:

Static versus dynamic communication: Depending on whether the communication graph of the application changes at runtime, graph can be classified as static or dynamic. If the communication graph is static, topology aware mapping can be done offline and used for the entirety of the run. If the communication is dynamic, periodic remapping depending on the changes in the communication graph are required. Several categories of parallel applications such as Lattice QCD, Ocean Simulations, and Weather Simulations have a stencil-like communication pattern that does not change during the run. On the other hand, molecular dynamics and cosmological simulations have dynamic communication patterns.

Regular versus irregular communication: Communication in a parallel application can be regular (structured) or irregular (unstructured). An example of regular communication is a five-point stencil-like application where every task communicates with four of its neighbors. When no specific pattern can be attributed to the communication graph, it is classified as irregular or unstructured. Unstructured grid computations are examples of applications with irregular communication graphs.

Point-to-point versus collective communication: Some applications primarily use point-to-point messages with minimal global communication. Others, however use collective operations such as broadcasts, reductions, and all-to-alls between all or a subset of processors. Different mapping algorithms are required to optimize different types of communication patterns.

Parallel applications can also be classified into computation bound or communication bound depending on the relative amount of communication involved. A large body of parallel applications spend a small portion of their overall execution time doing communication. Such applications will typically be unaffected by topology aware mapping. Communication-bound latency-sensitive applications benefit most in terms of performance from topology aware task mapping.

The Mapping Process

An algorithm for mapping of tasks in an application requires two inputs – the communication graph of an application and the machine topology of the allocated job partition. Given these two inputs, the aim is to map communicating objects or tasks close to one another on nearby physical processors. The success of a mapping algorithm is evaluated in terms of minimizing or maximizing some function which correlates well with the contention on the network or actual application performance.

Objective Functions

Mapping algorithms aim at minimizing some metric referred to as an objective function which should be chosen carefully. A good objective function is one that does an accurate evaluation of a mapping solution in terms of yielding better performance. Objective functions are also important to compare the optimality of different mapping solutions. Several objective functions which have been used for different mapping algorithms are listed below:

- Overlap between guest and host graph edges: One metric to determine the quality of the mapping is

the number of edges in the guest graph which fall on the host graph. This metric is referred to as the cardinality of the mapping by Bokhari [1]. The mapping which yields the highest cardinality is the best.

- Maximum dilation: This metric is used for architectures and applications where the longest edge in the communication graph determines the performance. In other words, the message that travels the maximum number of hops or links on the network determines the overall performance [7, 8].

$$\text{Maximum dilation} = \max_{i=1}^{n} |d_i| \tag{1}$$

The mapping which leads to the smallest dilation for any edge in the guest graph on the processor interconnect is the best.

- Hop-bytes: This is the weighted sum of all edges in the communication graph multiplied by their dilation on the processor graph as per the mapping algorithm [9, 10].

$$\text{Hop-bytes} = \sum_{i=1}^{n} d_i \times b_i \tag{2}$$

where d_i is the number of hops or links traveled by the message on the network and b_i is the size of the message in bytes.

Hop-bytes is a measure of the total communication traffic on the network and hence, an approximate indication of the contention. A smaller value for hop-bytes indicates less contention on the network. Average hops per byte is another way of expressing the same metric,

$$\text{Average hops per byte} = \frac{\sum_{i=1}^{n} d_i \times b_i}{\sum_{i=1}^{n} b_i} \tag{3}$$

The last two objective functions, maximum dilation and hop-bytes, are typically used today and are applicable in different scenarios. The choice of one over the other depends upon the parallel application and the architecture for which the mapping is being performed.

Heuristic Techniques for Mapping

Owing to the general applicability of mapping in various fields, a huge body of work exists targeting this problem. Many techniques used for solving combinatorial optimization problems can be used for obtaining solutions to the mapping problem. Simulated annealing, genetic algorithms, and neural network–based heuristics are examples of such physical optimization techniques. Other heuristic techniques are recursive partitioning, pairwise exchanges, and clustering and geometry-based mapping. Arunkumar et al. [3] categorize various heuristic techniques into – deterministic, randomized, and random start heuristics. The following sections discuss some of the mapping techniques classified into these categories.

Deterministic Heuristics

In this class, the choice of search path is deterministic and typically a fixed search strategy is used taking the domain-specific knowledge about the parallel application into account. Yu et al. [11] present folding and embedding techniques to obtain deterministic solutions for mapping of two- and three-dimensional grids on to 3D mesh topologies. Their topology mapping library provides support for MPI virtual topology functions on IBM Blue Gene machines. Bhatele [12] uses domain-specific knowledge and communication patterns of parallel application for heuristic techniques such as "affine transformation" inspired mapping and guided graph traversals to map on to 3D tori. The mapping library developed as a result can map application graphs that are regular (n-dimensional grids) as well as those that are irregular. Several application developers such as those of Blue Matter [4], Qbox [5], and OpenAtom [6] have developed application specific mapping algorithms to map tasks on to processor topologies. Recursive graph partitioning–based strategies which partition both the application and processor graph for mapping also fall under this category [13]. Algorithms using deterministic algorithms are typically the fastest among the three categories.

Randomized Heuristics

This category of solutions does not depend on domain-specific knowledge and uses search techniques that are randomized, yielding different solutions in successive executions. Neural networks, genetic algorithms, and simulated annealing–based heuristics are example of this class. Bokhari's algorithm of pairwise exchanges accompanied by probabilistic jumps also falls under this category.

In genetic algorithm–based heuristics [3], possible mapping solutions are first encoded in some manner and a random population of such patterns is generated. Then different genetic operators such as crossover and mutation are applied to derive new generations from old ones. Certain criteria are used to estimate the fitness of a selection and unfit solutions are rejected. Given a termination rule, the best solution among the population is taken to be the solution at termination.

Obtaining an exact solution to the mapping problem is difficult and iterative algorithms tend to produce solutions that are not globally optimal. The technique of simulated annealing provides a mechanism to escape local optima and hence is a good fit for mapping problems. The most important considerations for a simulated annealing algorithm are deciding a good objective function and an annealing schedule. This technique has been used for processor and link assignment by Midkiff et al. [14] and Bhanot et al. [15].

Random Start Heuristics

In some algorithms, a random initial mapping is chosen and then improved iteratively. Such solutions fall under the category of *random start* heuristics. Techniques such as pairwise exchanges and recursive partitioning fall under this category.

The technique of pairwise exchanges that starts from an initial assignment, is a simple brute force method which has been used with different variations to tackle the mapping problem [7]. The basic idea is simple: An objective function or metric to be optimized is selected and then an initial mapping of the guest graph on the host graph is determined. Then, a pair of nodes is chosen, either randomly or based on some selection criteria and their mappings are interchanged. If the metric or objective function becomes better, the exchange is preserved and the process is repeated, until some termination criterion is achieved.

Another technique in this class is task clustering followed by cluster allocation. In the clustering phase, tasks are clustered into groups equal to the number of processors using recursive min-cut algorithms. Then these clusters are allocated to the processors by starting with a random assignment and iteratively improving it by local exchanges. The first phase aims at minimizing intercluster communication without comprising load balancing while the second phase aims at minimizing inter-processor communication. This is especially useful for models such as Charm++ where the number of tasks is much larger than the number of processors.

Future Directions

The emergence of new architectures and network topologies requires modifying existing algorithms and developing new ones to suit them. As an example, the increase in number of cores per node adds another dimension to the network topology and should be taken into account. Algorithms also need to be developed for new parallel applications. There is a growing need for runtime support in the form of an automated mapping framework that can map applications intelligently on to the processor topology. This will reduce the burden on application developers to map individual applications and will also help reuse algorithms across similar communication graphs. Bhatele et al. [16] are making some efforts in this direction. There is an increasing demand for support in the MPI runtime for mapping of MPI virtual topology functions [11].

The increase in size of parallel machines and in the number of threads in a parallel program requires parallel and distributed techniques for mapping. Gathering the entire communication graph on one processor and applying sequential centralized techniques will not be feasible in the future. Hence, an effort should be made towards developing strategies which are distributed, scalable, and can be run in parallel. Hierarchical multilevel graph partitioning techniques are one such effort in this direction.

Related Entries

►Cray XT3 and Cray XT Series of Supercomputers
►Cray XT4 and Seastar 3-D Torus Interconnect
►Graph Partitioning
►Hypercubes and Meshes
►IBM Blue Gene Supercomputer
►Infiniband
►Interconnection Networks
►Load Balancing, Distributed Memory
►Locality of Reference and Parallel Processing
►Processes, Tasks, and Threads
►Routing (Including Deadlock Avoidance)
►Space-Filling Curves
►Task Graph Scheduling

Bibliographic Notes and Further Reading

Bokhari [1] wrote one of the first papers on task mapping for parallel programs. A good discussion of the various objective functions used for comparing mapping algorithms can be found in [7]. Fox et al. [17] divide the various mapping algorithms into physical optimization and heuristic techniques. Arunkumar et al. [3] provide another classification into deterministic, randomized, and random start heuristics.

Application developers attempting to map their parallel codes can gain insights from mapping algorithms developed by individual application groups [4–6]. Bhatele and Kale have been developing an automatic mapping framework for mapping of Charm++ and MPI applications to the processor topology [12]. They are also developing techniques for parallel and distributed topology aware mapping.

Bibliography

1. Bokhari SH (1981) On the mapping problem. IEEE Trans Comput 30(3):207–214
2. Kasahara H, Narita S (1984) Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans Comput 33:1023–1029
3. Arunkumar S, Chockalingam T (1992) Randomized heuristics for the mapping problem. Int J High Speed Comput (IJHSC) 4(4):289–300
4. Fitch BG, Rayshubskiy A, Eleftheriou M, Ward TJC, Giampapa M, Pitman MC (2006) Blue matter: approaching the limits of concurrency for classical molecular dynamics. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, ACM Press, New York, 11–17 Nov 2006
5. Gygi F, Draeger EW, Schulz M, Supinski BRD, Gunnels JA, Austel V, Sexton JC, Franchetti F, Kral S, Ueberhuber C, Lorenz J (2006) Large-scale electronic structure calculations of high-Z metals on the blue gene/L platform. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM Press, New York
6. Bhatelé A, Bohm E, Kalé LV (2011) Optimizing communication for Charm++ applications by reducing network contention. Concurr Comput 23(2):211–222
7. Lee S-Y, Aggarwal JK (1987) A mapping strategy for parallel processing. IEEE Trans Comput 36(4):433–442
8. Berman F, Snyder L (1987) On mapping parallel algorithms into parallel architectures. J Parallel Distrib Comput 4(5):439–458
9. Ercal F, Ramanujam J, Sadayappan P (1988) Task allocation onto a hypercube by recursive mincut bipartitioning. In: Proceedings of the 3rd conference on Hypercube concurrent computers and applications, ACM Press, New York, pp 210–221
10. Agarwal T, Sharma A, Kalé LV (2006) Topology-aware task mapping for reducing communication contention on large parallel machines, In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006, Rhodes Island, 25–29 Apr 2006. IEEE, Piscataway
11. Yu H, Chung I-H, Moreira J (2006) Topology mapping for blue gene/L supercomputer. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, 11–17 Nov 2006. ACM, New York, p 116
12. Bhatele A (2010) Automating topology aware mapping for supercomputers. Ph.D. thesis, Dept. of Computer Science, University of Illinois. http://hdl.handle.net/2142/16578 (August 2010)
13. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(1):291–307
14. Bollinger SW, Midkiff SF (1988) Processor and link assignment in multicomputers using simulated annealing. In: 1988 ICPP, vol 1, Aug 1988, pp 1–7
15. Bhanot G, Gara A, Heidelberger P, Lawless E, Sexton JC, Walkup R (2005) Optimizing task layout on the blue gene/L supercomputer. IBM J Res Dev 49(2/3):489–500
16. Bhatele A, Gupta G, Kale LV, Chung I-H (2010) Automated mapping of regular communication graphs on mesh interconnects. In: Proceedings of International Conference on High Performance Computing & Simulation (HiPCS) 2010, Caen, 28 June–2 July 2010. IEEE, Piscataway
17. Mansour N, Ponnusamy R, Choudhary A, Fox GC (1993) Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In: ICS'93: Proceedings of the 7th International Conference on Supercomputing, Tokyo, 19–23 July 1993. ACM, New York, pp 1–10

Torus

►Networks, Direct

Total Exchange

►Allgather

Trace Scheduling

Stefan M. Freudenberger
Zürich, Switzerland

Definition

Trace scheduling is a global acyclic instruction scheduling technique in which the scheduling region consists

of a linear acyclic sequence of basic blocks embedded in the control flow graph. Trace scheduling differs from other global acyclic scheduling techniques by allowing the scheduling region to be entered after the first instruction.

Trace scheduling was the first global instruction scheduling technique that was proposed and successfully implemented in both research and commercial compilers. By demonstrating that simple microcode operations could be statically compacted and scheduled on multi-issue hardware, trace scheduling provided the basis for making large amounts of instruction-level parallelism practical. Its first commercial implementation demonstrated that commercial codes could be statically compiled for multi-issue architectures, and thus greatly influenced and contributed to the performance of superscalar architectures. Today, the ideas of trace scheduling and its descendants are implemented in most compilers.

Discussion

Introduction

Global scheduling techniques are needed for processors that expose instruction-level parallelism (ILP), that is, processors that allow multiple operations to execute simultaneously. This situation may independently arise for two reasons: either because a processor issues more than a single operation during each clock cycle, or because a processor allows issuing independent operations while deeply pipelined operations are still executing. The number of independent operations that need to be found for an ILP processor is a function of both the number of operations issued per clock cycle, and the latency of operations, whether computational or memory. The latency of computational operations depends upon the design of the functional units. The latency of memory operations depends upon the design and latencies of caches and main memory, as well as on the availability of prefetch and cache-bypassing operations. Global scheduling techniques are needed for these processors because the number of independent operations available in a typical basic block is too small to fully utilize their available hardware resources. By expanding the scheduling region, more operations become available for scheduling. Global scheduling techniques differ from other global code motion techniques (such as loop-invariant code motion or partial redundancy elimination) because they take into account the available hardware resources (such as available functional units and operation issue slots).

Instruction scheduling techniques can be broadly classified based on the region that they schedule, and whether this region is cyclic or acyclic. Algorithms that schedule only single basic blocks are known as *local scheduling* algorithms; algorithms that schedule multiple basic blocks at once are known as *global scheduling* algorithms. Global scheduling algorithms that operate on entire loops of a program are known as *cyclic scheduling* algorithms, while methods that impose a scheduling barrier at the end of a loop body are known as *acyclic scheduling* algorithms. Global scheduling regions include regions consisting of a single basic block as a "degenerate" form of region, and acyclic schedulers may consider entire loops but, unlike cyclic schedulers, stop at the loops' back edges (a back edge points to an ancestor in a depth-first traversal of the control flow graph; it captures the flow from one iteration of the loop to the start of the next iteration).

All scheduling algorithms can benefit from hardware support. When control-dependent operations that can cause side effects move above their controlling branch, they need to be either executed conditionally so that their effects only arise if the operation is executed in the original program order, or any side effects must be delayed until the point at which the operation would have been executed originally.

Hardware techniques to support this include *predication* of operations, implicit or explicit *register renaming*, and mechanisms to suppress or delay exceptions in order to prevent an incorrect exception to be signaled. Predication of operations controls whether the side effects of the predicated operations become visible to the program state through an additional predicate operand. The predicate operand can be implicit (such as the conditional execution of operations in branch delay slots depending on the outcome of the branch condition) or explicit (through an additional machine register operand); in the latter case, the predicate operand could simply be the same predicate that controls the conditional branch on which the operation was control-dependent in the original flow graph (in which case the predicated operation could move just above a single conditional branch). Register renaming refers to the technique where additional machine registers are

used to hold the results of an operation until the point where the operation would have occurred in the original program order.

Global scheduling algorithms principally consist of two phases: *region formation* and *schedule construction*. Algorithms differ in the shape of the region and the global code motions permitted during scheduling. Depending on the region and the allowed code motions, *compensation code* needs to be inserted at appropriate places in the control flow graph to maintain the original program semantics; depending on the code motions allowed during scheduling, compensation code needs to be inserted during the scheduling phase of the compiler.

Trace scheduling allows traces to be entered after the first operation and before the last operation. This complicates the determination of compensation code because the location of *rejoin points* cannot be done before a trace has been scheduled. This leads to the following overall trace scheduling loop:

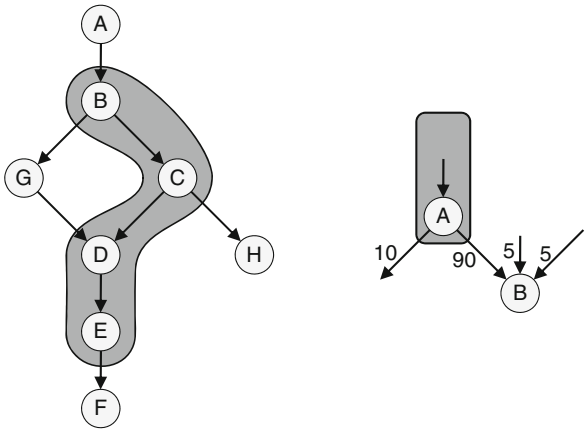
```
while (unscheduled operations remain)
{
       select trace T
       construct schedule for T
       bookkeeping -
         determine rejoin points to T
         generate compensation code
 }
```

The remainder of this entry first discusses region formation and schedule construction in general and as it applies to trace scheduling, and then compares trace scheduling to other acyclic global scheduling techniques. Cyclic scheduling algorithms are discussed elsewhere.

Region Formation – Trace Picking

Traces were the first global scheduling region proposed, and represent contiguous linear paths through the code (Fig. 1). More formally, a trace consists of the operations of a sequence of basic blocks B_0, B_1, …, B_n with the properties that:

- Each basic block is a predecessor of the next in the sequence (i.e., for each $k = 0, \ldots, n-1$, B_k is a predecessor of B_{k+1}, and B_{k+1} is a successor of B_k in the control flow graph).



Trace Scheduling. Fig. 1 Trace selection. The *left* diagram shows the selected trace. The *right* diagram illustrates the mutual-most-likely trace picking heuristic: assume that *A* is the last operation of the current trace, and that *B* is one of *A*'s successors. Here *B* is the most likely successor of *A*, and *A* is the most likely predecessor of *B*

- For any j, k there is no path $B_j \rightarrow B_k \rightarrow B_j$ except for those that include B_0 (i.e., the code is cycle free, except that the entire region can be part of some encompassing loop).

Note that this definition does not exclude forward branches within the region, nor control flow that leaves the region and reenters it at a later point. This generality has been controversial in the research community because many felt that the added complexity of its implementation was not justified by its added benefit and has led to several alternative approaches that are discussed below.

Of the many ways in which one can form traces, the most popular algorithm employs the following simple trace formation algorithm:

- Pick the as-yet unscheduled operation with the largest expected execution frequency as the seed operation of the trace.
- Grow the trace both forward in the direction of the flow graph as well as backward, picking the *mutually most-likely* successor (predecessor) operation to the currently last (first) operation on the trace.
- Stop growing a trace when either no mutually most-likely successor (predecessor) exists, or when some heuristic trace length limit has been reached.

The mutually most-likely successor S of an operation P is the operation with the properties that:

- S is the most likely successor of P;
- P is the most likely predecessor of S.

For this definition, it is immaterial whether the likelihood that S follows P (P precedes S) is based on available profile data collected during earlier runs of the program, has been determined by a synthetic profile, or is based on source annotations in the program. Of course, the more benefit is derived from having picked the correct trace, the greater is the penalty when picking the wrong trace.

Trace picking is the region formation technique used for trace scheduling. Other acyclic region formation techniques and their relationship to trace scheduling are discussed below.

Region Enlargement

Trace selection alone typically does not expose enough ILP for the instruction scheduler of a typical ILP processor. Once the limit on the length of a "natural" trace has been reached (e.g., the entire loop body), *region-enlargement* techniques can be employed to further increase the size of the region, albeit at the cost of a larger code size for the program. Many enlargement techniques exploit the fact that programs iterate and grow the size of a region by making extra copies of highly iterated code, leading to a larger region that contains more ILP.

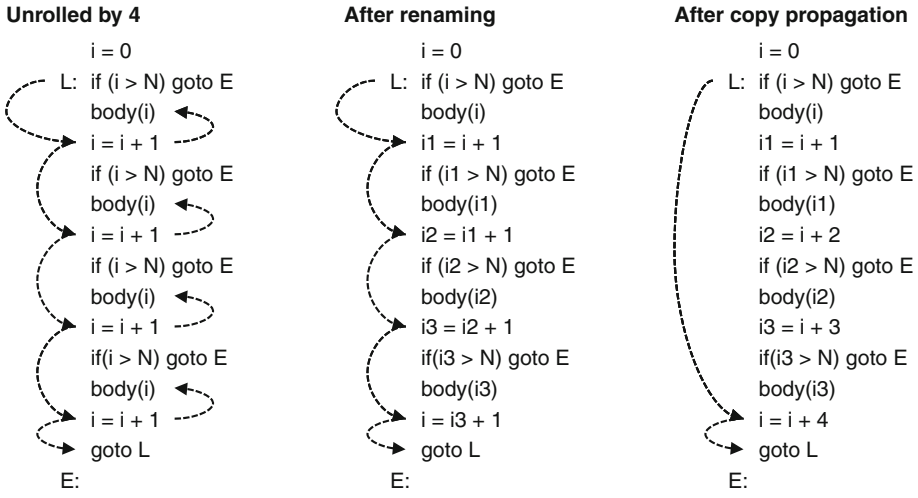
These code-replicating techniques have been criticized by advocates of other approaches, such as cyclic scheduling and loop-level parallel processing, because comparable benefits to larger schedule regions may be found using other techniques. However, no study appears to exist that quantifies such claims.

The simplest and oldest region-enlargement technique is *loop unrolling* (Fig. 2): to unroll a loop, duplicate its body several times, change the targets of the back edges of each copy but the last to point to the header of the next copy (so that the back edges of the last copy point back to the loop header of the first copy). Variants of loop unrolling include pre-/post-conditioning of a loop by k for counted *for* loops with unknown loop bounds (leading to two loops: a "fixup loop" that executes up to k iterations; and a "main loop" that is unrolled by k and has its internal exits removed; the fixup loop can precede or follow the main loop), and loop peeling by the expected small iteration count. When the iteration count of the fixup loop of a p-conditioned loop is small (which it typically is), the fixup loop is completely unrolled.

Typically, loop unrolling is done before region formation so that the enlarged region becomes available

Original loop	Unrolled by 4	Pre-conditioned by 4	Post-conditioned by 4
L: if … goto E	L: if … goto E	if … goto L	L: if … goto X
body	body	body	body
goto L	if … goto E	if … goto L	body
E:	body	body	body
	if … goto E	if … goto L	body
	body	body	goto L
	if … goto E	L: if … goto E	X: if … goto E
	body	body	body
	goto L	body	if … goto E
	E:	body	body
		body	if … goto E
		goto L	body
		E:	E:

Trace Scheduling. Fig. 2 Simplified illustration of variants of loop unrolling. "if" and "goto" represent the loop control operations; "body" represents the part of the loop without loop-related control flow. In the general case (e.g., a *while* loop) the loop exit tests remain inside the loop. This is shown in the second column ("unrolled by 4"). For counted loops (i.e., *for* loops), the compiler can condition the unrolled loop so that the loop conditions can be removed from the main body of the loop. Two variants of this are shown in the two rightmost columns. Modern compilers will typically precede the loop with a zero trip count test and place the loop condition at the bottom of the loop. This removes the unconditional branch from the loop



Trace Scheduling. Fig. 3 Typical induction variable manipulations for loops. Downward arrows represent flow dependences; upward arrows represent anti dependences. Only the critical dependences are shown

to the region selector. This is done to keep the region selector simpler but may lead to phase-ordering issues, as loop unrolling has to guess the "optimal" unroll amount. At the same time, when loops are unrolled before region formation then the resulting code can be scalar optimized in the normal fashion; in particular height-reducing transformations that remove dependences between the individual copies of the unrolled loop body can expose a larger amount of parallelism between the individual iterations (Fig. 3). Needless to say, if no parallelism between the iterations exists or can be found, loop unrolling is ineffective.

Loop unrolling in many industrial compilers is often rather effective because a heuristically determined small amount of unrolling is sufficient to fill the resources of the target machine.

Region Compaction – Instruction Scheduler

Once the scheduling region has been selected, the instruction scheduler assigns functional units of the target machine and time slots in the instruction schedule to each operation of the region. In doing so, the scheduler attempts to minimize an objective cost function while maintaining program semantics and obeying the resource limitations of the target architecture. Often, the objective cost function is the expected execution time, but other objective functions are possible (for example, code size and energy efficiency could be part of an objective function).

The semantics of a program defines certain sequential constraints or *dependences* that must be maintained by a valid execution. These dependences preclude some reordering of operations within a program. The data flow of a program imposes *data dependences*, and the control flow of a program imposes *control dependences*. (Note the difference between control flow and control dependence: block B is control dependent on block A if A precedes B along some path, but B does not post-dominate A. In other words, the result of the control decision made in A directly affects whether or not B is executed.)

There are three types of data dependences: *read-after-write* dependences (also called *RAW*, *flow*, or *true* dependences), *write-after-read* dependences (also called *WAR* or *anti* dependences), and *write-after-write* dependences (also called *WAW* or *output* dependences). The latter two types are also called *false* dependences because they can be removed by renaming.

There are two types of control dependences: *split* dependences may prevent operations from moving below the exit of a basic block, and *join* dependences may prevent operations from moving above the entrance to a basic block. Control dependence does not constrain the relative order of operations within a

basic block but rather expresses constraints on moving operations between basic blocks.

Both data and control dependences represent ordering constraints on the program execution, and hence induce a partial ordering on the operations. Any partial ordering can be represented as a *directed acyclic graph* (DAG), and DAGs are indeed often used by scheduling algorithms. Variants to the simple DAG are the *data dependence graph* (DDG), and the *program dependence graph* (PDG). All these graphs represent operations as nodes and dependences as edges (some graphs only express data dependences, while others include both data and control dependences).

Code Motion Between Adjacent Blocks

Two fundamental techniques, predication and speculation, are employed by schedulers (or earlier phases) to transform or remove control dependence. While it is sometimes possible to employ either technique, they represent independent techniques, and usually one is more natural to employ in a given situation. Speculation is used to move operations above a branch that is highly weighted in one direction; predication is used to collapse short sequences of alternative operations following a branch that is nearly equally likely in each direction. Predication can also play an important role in software pipelining.

Speculative code motion (or *code hoisting* and sometimes *code sinking*) moves operations above control-dominating branches (or below joins for sinking). In principle, this transformation does not always maintain the original program semantics, and in particular it may change the exception behavior of the program. If an operation may generate an exception and the exception recovery model does not allow speculative exceptions to be dismissed (ignored), then the compiler must generate recovery code that raises the exception at the original program point of the speculated operation. Unlike predication, speculation actually removes control dependences, and thus potentially reduces the length of the critical path of execution. Depending on the shape and size of recovery code, and if multiple operations are speculated, the addition of recovery code can lead to a substantial amount of code.

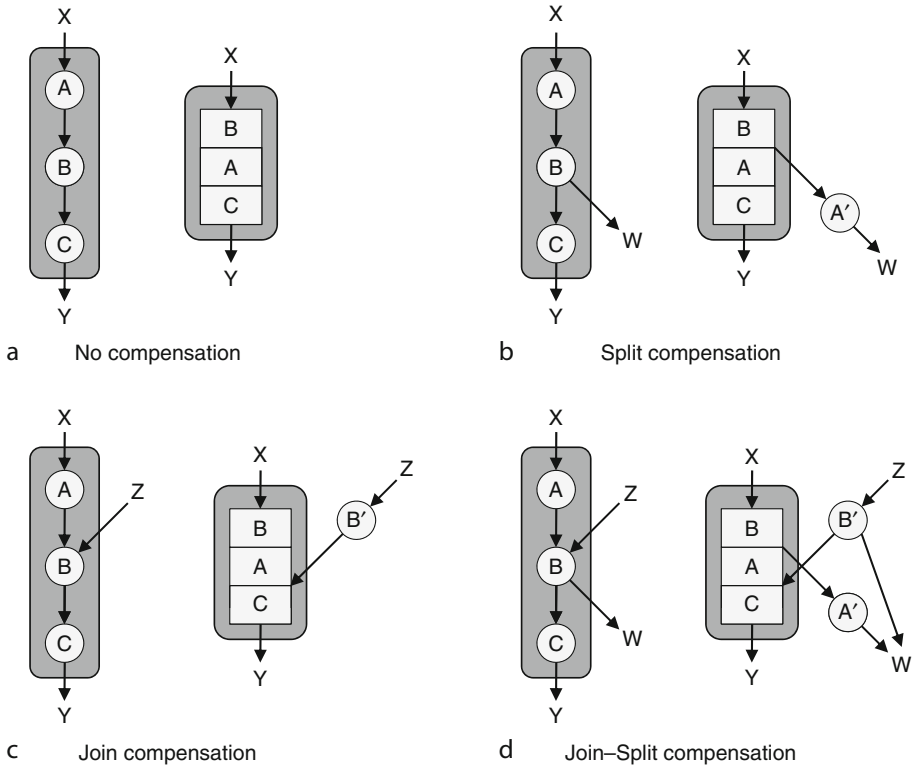
Predication is a technique where with hardware support operations have an additional input operand, the predicate operand, which determines whether any effects of executing the operations are seen by the program execution. Thus, from an execution point of view, the operation is conditionally executed under the control of the predicate input. Hence changing a control-dependent operation to its predicated equivalent that depends on a predicate that is equivalent to the condition of the control dependence turns control dependence into data dependence.

Trace Compaction

There are many different scheduling techniques, which can broadly be classified by features into cycle versus operation scheduling, linear versus graph-based, cyclic versus acyclic, and greedy versus backtracking. However, for trace scheduling itself the scheduling technique employed is not of major concern; rather, trace scheduling distinguishes itself from other global acyclic scheduling techniques by the way the scheduling region is formed, and by the kind of code motions permitted during scheduling. Hence these techniques will not be described here, and in the following, a greedy graph-based technique, namely list scheduling, will be used.

Compensation Code

During scheduling, typically only a very small number of operations can be moved freely between basic blocks without changing program semantics. Other operations may be moved only when additional *compensation code* is inserted at an appropriate place in order to maintain original program semantics. Trace scheduling is quite general in this regard. Recall that a trace may be entered after the first instruction, and exited before the last instruction. In addition, trace scheduling allows operations in the region (trace) to move freely during scheduling relative to entries (join points) to and exits (split points) from the current trace. A separate *bookkeeping* step restores the original program semantics after trace compaction through the introduction of compensation code. It is this freedom of code motion during scheduling, and the introduction of compensation code between the scheduling of individual regions, that represents a major difference between trace scheduling and other acyclic scheduling techniques.



Trace Scheduling. Fig. 4 Basic scenarios for compensation code. In each diagram, the *left* part shows the selected trace, the *right* part shows the compacted code where operation *B* has moved above operation *A*

Since trace scheduling allows operations to move above join points as well as below split points (conditional branches) in the original program order, the bookkeeping process includes the following kinds of compensation. Note that a complete discussion of all the intricacies of compensation code is well beyond the scope of this entry; however, the following is a list of the simple concepts that form the basis of many of the compensation techniques used in compilers.

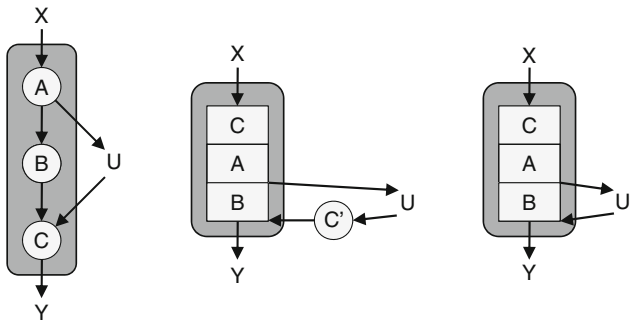
No compensation (Fig. 4a). If the global motion of an operation on the trace does not change the relative order of operations with respect to split and join points, no compensation code is needed. This covers the situation when an operation moves above a split, in which case the operation becomes *speculative*, and requires compensation depending on the recovery model of exceptions: in the case of *dismissible speculation*, no compensation code is needed; in the case of *recovery speculation*, the compiler has to emit a recovery block to guarantee the timely delivery of exceptions for correctly speculated operations.

Split compensation (Fig. 4b). When an operation *A* moves below a split operation *B* (i.e., a conditional branch), a copy of *A* (called A') must be inserted on the off-trace split edge. When multiple operations move below a split operation, they are all copied on the off-trace edge in source order. These copies are unscheduled, and hence will be picked and scheduled later during the trace scheduling of the program.

Join compensation (Fig. 4c). When an operation *B* moves above a join point *A*, a copy of *B* (called B') must be copied on the off-trace join edge. When multiple operations move above a join point, they are all copied on the off-trace edge in source order.

Join–Split compensation (Fig. 4d). When splits are allowed to move above join points, the situation becomes more complicated: when the split is copied on the rejoin edge, it must account for any split compensation and therefore introduce additional control paths with additional split copies.

These rules define the compensation code required to correctly maintain the semantics of the original



Trace Scheduling. Fig. 5 Compensation copy suppression. The *left* diagram shows the selected trace. The *middle* diagram shows the compacted code where operation *C* has moved above operation *A* together with the normal join compensation. The *right* diagram shows the result of compensation copy suppression assuming that *C* is available at *Y*

program. The following observations can be used to heuristically control the amount of compensation code that is generated.

To limit split compensation, the Multiflow Trace Scheduling compiler [12], the first commercial compiler to implement trace scheduling, required that all operations that precede a split on the trace precede the split on the schedule. While this limits the amount of available parallelism, the intuitive explanation is that a trace represents the most likely execution path; the on-trace performance penalty of this restriction is small; and off-trace the same operations would have to be executed in the first place. Multiflow's implementation excluded memory-store operations from this heuristic because in Multiflow's Trace architecture stores were unconditional and hence could not move above splits; they were allowed to move below splits to avoid serialization between stores and loop exits in unrolled loops. The Multiflow compiler also restricted splits to remain in source order. Not only did this reduce the amount of compensation code, it also ensured that all paths created by compensation code are subsets of paths (possibly rearranged) in the flow graph before trace scheduling.

Another observation concerns the possible suppression of compensation copies [8] (Fig. 5): sometimes an operation *C* that moves above a join point following an operation *B* actually moves to a position on the trace that dominates the join point. When this happens, and the result of *C* is still available at the join point, no copy of *C* is needed. This situation often arises when loops with internal branches are unrolled. Without copy suppression, such loops can generate large amounts of redundant compensation code.

Bibliographic Notes and Further Reading

The simplest form of a scheduling region is a region where all operations come from a single-entry single-exit straight-line piece of code (i.e., a basic block). Since these regions do not contain any internal control flow, they can be scheduled using simple algorithms that maintain the partial order given by data dependences. (For simplicity, it is best to require that operations that could incur an exception must end their basic block, allowing the exception to be caught by an exception handler.)

Traces and trace scheduling were the first region-scheduling techniques proposed. They were introduced by Fisher [6, 7] and described more carefully in Ellis' thesis [4]. By demonstrating that simple microcode operations could be statically compacted and scheduled on multi-issue hardware trace scheduling provided the basis for making VLIW machines practical. Trace scheduling was implemented in the Multiflow compiler [12]; by demonstrating that commercial codes could be statically compiled for multi-issue architectures, this work also greatly influenced and contributed to the performance of superscalar architectures. Today, ideas of trace scheduling and its descendants are implemented in most compilers (e.g., GCC, LLVM, Open64, Pro64, as well as commercial compilers).

Trace scheduling inspired several other global acyclic scheduling techniques. The most important linear acyclic region-scheduling techniques are presented next.

Superblocks

Hwu and his colleagues on the IMPACT project have developed a variant of trace scheduling called *superblock scheduling*. Superblocks are traces with the added restriction that the superblock must be entered at the top [2, 3]. Hence superblocks can be joined only before the first or after the last operation in the superblock. As such, superblocks are single-entry, multiple-exit traces.

Since superblocks do not contain join points, scheduling a superblock cannot generate any join or join–split compensation. By also prohibiting motion below splits, superblock scheduling avoids the need of generating compensation code outside the schedule region, and hence does not require a separate bookkeeping step. With these restrictions, superblock formation can be completed before scheduling starts, simplifying its implementation.

Superblock formation often includes a technique called *tail duplication* to increase the size of the superblock: tail duplication copies any operations that follow a rejoin in the original control flow graph and that are part of the superblock into the rejoin edge, thus effectively lowering the rejoin point to the end of the superblock. This is done at superblock formation time, before any compaction takes place [11].

A variant of superblock scheduling that allows speculative code motion is sentinel scheduling [14].

Hyperblocks

A different approach to global acyclic scheduling also originated with the IMPACT project. *Hyperblocks* are superblocks that have eliminated internal control flow using predication [13]. As such, hyperblocks are single-entry, multiple-exit traces (superblocks) that use predication to eliminate internal control flow.

Treegions

Treegions [9, 10] consist of the operations from a list of basic blocks $B_0, B_1, \ldots, B_n$ with the properties that:

- For each $j > 0$, B_j has exactly one predecessor.
- For each $j > 0$, the predecessor B_i of B_j is also on the list, where $i < j$.

Hence, treegions represent trees of basic blocks in the control flow graph. Since treegions do not contain any side entrances, each path through a treegion yields a superblock. Like superblock compilers, treegion compilers employ tail duplication and other region-enlarging techniques. More recent work by Zhou and Conte [16, 17] shows that treegions can be made quite effective without significant code growth.

Nonlinear Regions

Nonlinear region approaches include percolation scheduling [1] and DAG-based scheduling [15]. Trace scheduling-2 [5] extends treegions by removing the restriction on side entrances. However, its implementation proved so difficult that its proposer eventually gave up on it, and no formal description or implementation of it is known to exist.

Related Entries

►Modulo Scheduling and Loop Pipelining

Bibliography

1. Aiken A, Nicolau A (1988) Optimal loop parallelization. In: Proceedings of the SIGPLAN 1988 conference on programming language design and implementation, June 1988, pp 308–317
2. Chang PP, Warter NJ, Mahlke SA, Chen WY, Hwu WW (1991) Three superblock scheduling models for superscalar and superpipelined processors. Technical Report CRHC-91-29. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign
3. Chang PP, Mahlke SA, Chen WY, Warter NJ, Hwu WW (1991) IMPACT: an architectural framework for multiple-instruction-issue processors. In: Proceedings of the 18th annual international symposium on computer architecture, May 1991, pp 266–275
4. Ellis JR (1985) Bulldog: a compiler for VLIW architectures. PhD thesis, Yale University
5. Fisher JA (1993) Global code generation for instruction-level parallelism: trace scheduling-2. Technical Report HPL-93-43. Hewlett-Packard Laboratories
6. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction, IEEE Trans Comput, July 1981, 30(7):478–490
7. Fisher JA (1979) The optimization of horizontal microcode within and beyond basic blocks. PhD dissertation. Technical Report COO-3077-161. Courant Institute of Mathematical Sciences, New York University, New York, NY

8. Freudenberger SM, Gross TR, Lowney PG (1994) Avoidance and suppression of compensation code in a trace scheduling compiler, ACM Trans Program Lang Syst, July 1994, 16(4):1156–1214
9. Havanki WA (1997) Treegion scheduling for VLIW processors. MS thesis. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC
10. Havanki WA, Banerjia S, Conte TM (1998) Treegion scheduling for wide issue processors. In: Proceedings of the fourth international symposium on high-performance computer architecture, February 1998, pp 266–276
11. Hwu WW, Mahlke SA, Chen WY, Chang PP, Warter NJ, Bringmann RA, Ouellette RG, Hank RE, Kiyohara T, Haab GE, Holm JG, Lavery DM (May 1993) The superblock: an effective technique for VLIW and superscalar compilation. J Supercomput, 7(1–2):229–248
12. Lowney PG, Freudenberger SM, Karzes TJ, Lichtenstein WD, Nix RP, O'Donnell JS, Ruttenberg JC (1993) The Multiflow trace scheduling compiler, J Supercomput, May 1993, 7(1-2):51–142
13. Mahlke SA, Lin DC, Chen WY, Hank RE, Bringmann RA (1992) Effective compiler support for predicated execution using the hyperblock. In: Proceedings of the 25th annual international symposium on microarchitecture, 1992, pp 45–54
14. Mahlke SA, Chen WY, Bringmann RA, Hank RE, Hwu WW, Rau BR, Schlansker MS (1993) Sentinel scheduling: a model for compiler-controlled speculative execution, ACM Trans Comput Syst, November 1993, 11(4):376–408
15. Moon SM, Ebcioglu K (1997) Parallelizing nonnumerical code with selective scheduling and software pipelining, ACM Trans Program Lang Syst, November 1997, 19(6):853–898
16. Zhou H, Conte TM (2002) Code size efficiency in global scheduling for ILP processors. In: Proceedings of the sixth annual workshop on the interaction between compilers and computer architectures, February 2002, pp 79–90
17. Zhou H, Jennings MD, Conte TM (2001) Tree traversal scheduling: a global scheduling technique for VLIW/EPIC processors. In: Proceedings of the 14th annual workshop on languages and compilers for parallel computing, August 2001, pp 223–238

Trace Theory

Volker Diekert[1], Anca Muscholl[2]
[1]Universität Stuttgart FMI, Stuttgart, Germany
[2]Université Bordeaux 1, Talence, France

Synonyms

Partial computation; Theory of Mazurkiewicz-traces

Definition

Trace Theory denotes a mathematical theory of free partially commutative monoids from the perspective of concurrent or parallel systems. Traces, or equivalently, elements in a free partially commutative monoid, are given by a sequence of letters (or atomic actions). Two sequences are assumed to be equal if they can be transformed into each other by equations of type $ab = ba$, where the pair (a, b) belongs to a predefined relation between letters. This relation is usually called *partial commutation* or *independence*. With an empty independence relation, that is, without independence, the setting coincides with the classical theory of words or strings.

Discussion

Introduction

The analysis of sequential programs describes a run of a program as a sequence of atomic actions. On an abstract level such a sequence is simply a string in a free monoid over some (finite) alphabet of letters. This purely abstract viewpoint embeds program analysis into a rich theory of combinatorics on words and a theory of automata and formal languages. The approach has been very fruitful from the early days where the first compilers have been written until now where research groups in academia and industry develop formal methods for verification.

Efficient compilers use autoparallelization, which provides a natural example of independence of actions resulting in a partial commutation relation. For example, let $a; b; c; a; d; e; f$ be a sequence of arithmetic operations where:

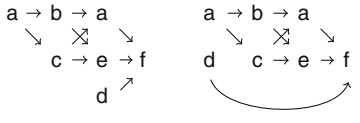
$$(a)\ x := x + 2y, \quad (b)\ x := x - z, \quad (c)\ y := y \cdot 5z$$

$$(d)\ w := 2w, \quad (e)\ z := y \cdot z, \quad (f)\ z := x + y \cdot w.$$

A concurrent-read-exclusive-write protocol yields a list of pairs of independent operations (a, d), (a, e), (b, c), (b, d), (c, d), and (d, e), which can be performed concurrently or in any order. The sequence can therefore be performed in four parallel steps $\{a\}; \{b, c\}; \{a, d, e\}; \{f\}$, but as d commutes with a, b, c the result of $a; b; c; a; d; e; f$ is equal to $a; d; b; c; a; e; f$, and two processors are actually enough to guarantee minimal parallel execution time, since another possible schedule is $\{a, d\}; \{b, c\}; \{a, e\}; \{f\}$. Trace theory yields a tool to do such (data-independent) transformations automatically.

Parallelism and concurrency demand for specific models, because a purely sequential description is neither accurate nor possible in all cases, for example, if asynchronous algorithms are studied and implemented. Several formalisms have been proposed in this context. Among these models there are Petri nets, Hoare's CSP and Milner's CCS, event structures, and branching temporal logics. The mathematical analysis of Petri nets is however quite complicated and much of the success of Hoare's and Milner's calculus is due to the fact that it stays close to the traditional concept of sequential systems relying on a unified and classical theory of words. Trace theory follows the same paradigm; it enriches the theory of words by a very restricted, but essential formalism to capture the main aspects of parallelism: In a static way a set I of independent letters (a, b) is fixed, and sequences are identified if they can be transformed into each other by using equations of type $ab = ba$ for $(a, b) \in I$. In computer science this approach appeared for the first time in the paper by Keller on *Parallel Program Schemata and Maximal Parallelism* published in 1973. Based on the ideas of Keller and the behavior of elementary net systems, Mazurkiewicz introduced in 1977 the notion of *trace theory* and made its concept popular to a wider computer science community. Mazurkiewicz's approach relies on a graphical representation for a trace. This is a node-labeled directed acyclic graph, where arcs are defined by the dependence relation, which is by definition the complement of the independence relation I.

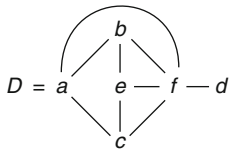
Thereby, a concurrent run has an immediate graphical visualization, which is obviously convenient for practice. The picture of the two parallel executions $\{a\}\,;\{b, c\}\,;\{a, d, e\}\,;\{f\}$ and $\{a, d\}\,;\{b, c\}\,;\{a, e\}\,;\{f\}$ can be depicted as follows, which represents (the Hasse diagrams of) isomorphic labeled partial orders:



Moreover, the graphical representation yields immediately a correct notion of *infinite trace*, which is not clear when working with partial commutations. In the following years it became evident that trace theory indeed copes with some important phenomena such as *true concurrency*. On the other hand it is still close to the classical theory of word languages describing sequential programs. In particular, it is possible to transfer the notion of finite sequential state control to the notion of asynchronous state control. This important result is due to Zielonka; it is one of the highlights of the theory. There is a satisfactory theory of recognizable languages relating finite monoids, rational operations, asynchronous automata, and logic. This leads to decidability results and various effective operations. Moreover, it is possible to develop a theory of asynchronous Büchi automata, which enables in trace theory the classical automata theory-based approach to automated verification.

Mathematical Definitions and Normal Forms

Trace theory is founded on a rigorous mathematical approach. The underlying combinatorics for partial commutation were studied in mathematics already in 1969 in the seminal Lecture Notes in Mathematics *Problèmes combinatoires de commutation et réarrangements* by Cartier and Foata. The mathematical setting uses a finite alphabet Σ of letters and the specification of a symmetric and irreflexive relation $I \subseteq \Sigma \times \Sigma$, called the *independence* relation. Conveniently, its complement $D = \Sigma \times \Sigma \subseteq I$ is called the *dependence* relation. The dependence relation has a direct interpretation as graph as well. For the dependency used in the first example above it looks as follows:



The intended semantics is that independent letters commute, but dependent letters must be ordered. Taking $ab = ba$ with $(a, b) \in I$ as defining relations one obtains a quotient monoid $\mathbb{M}(\Sigma, I)$, which has been called *free partial commutative monoid* or simply *trace monoid* in the literature. The elements are finite (Mazurkiewicz-)traces. For $I = \emptyset$, traces are just words in Σ^*; for a full independence relation, that is, $D = \mathrm{id}_\Sigma$, traces are vectors in some $\mathbb{N}^k$, hence Parikh-images of words. The general philosophy is that the extrema Σ^* and $\mathbb{N}^k$ are

well understood (which is far from being true), but the interesting and difficult problems arise when $\mathbb{M}(\Sigma, I)$ is neither free nor commutative.

For effective computations and the design of algorithms appropriate normal forms can be used. For the *lexicographic* normal form it is assumed that the alphabet Σ is totally ordered, say $a < b < c < \cdots < z$. This defines a lexicographic ordering on Σ^* exactly the same way words are ordered in a standard dictionary. The lexicographic normal form of a trace is the minimal word in Σ^* representing it. For example, if I is given by $\{(a,d),(d,a),(b,c),(c,a)\}$, then the trace defined by the sequence *badacb* is the congruence class of six words:

$$\{baadbc, badabc, bdaabc, baadcb, badacb, bdaacb\}.$$

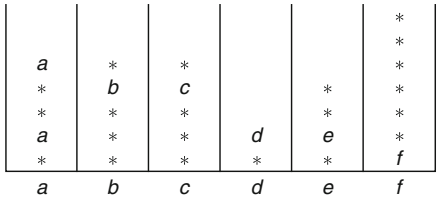
Its lexicographic normal form is the first word *baadbc*. An important property of lexicographic normal forms has been stated by Anisimov and Knuth. A word is in lexicographic normal form if and only if it does not contain a *forbidden pattern*, which is a factor *bua* where $a < b \in \Sigma$ and the letter a commutes with all letters appearing in $bu \in \Sigma^*$. As a consequence, the set of lexicographic normal forms is a regular language.

The other main normal is due to Foata. It is a normal form that encodes a maximal parallel execution. Its definition uses *steps*, where a step means here a subset $F \subseteq \Sigma$ of pairwise independent letters. Thus, a step requires only one parallel execution step. A step F yields a trace by taking the product $\Pi_{a\in F} a$ over all its letters in any order. The *Foata normal form* is a sequence of steps $F_1 \cdots F_k$ such that $F_1, \ldots, F_k$ are chosen from left to right with maximal cardinality. The sequence $\{a,d\};\{b,c\};\{a,e\};\{f\}$ above has been the Foata normal form of *abcadef*.

The graphical representation of a trace due to Mazurkiewicz can be viewed as a third normal form. It is called the *dependence graph representation*; and it is closely related to the Foata normal form. Say a trace t is specified by some sequence of letters $t = a_1 \cdots a_n$. Each index $i \in V = \{1, \ldots, n\}$ is labeled by the letter a_i. Finally, arcs $(i,j) \in E$ are introduced if and only if both $(a_i, a_j) \in D$ and $i < j$. In this way an acyclic directed graph $G(t)$ is defined which is another unique representation of t. The information about t is also contained in the induced partial order (i.e., the transitive closure of $G(t)$) or in its Hasse-diagram (i.e., removing all transitive arcs from $G(t)$).

Computation of Normal Forms

There are efficient algorithms that compute normal forms in polynomial time. A very simple method uses a stack for each letter of the alphabet Σ. An input word is scanned from right to left, so the last letter is read first. When processing a letter a it is pushed on its stack and a marker is pushed on the stack of all the letters b ($b \neq a$), which do not commute with a. Once the word has been processed its lexicographic normal form, the Foata normal form, and the Hasse-diagram of the dependence graph representation can be obtained straightforwardly. For example, the sequence $a;b;c;a;d;e;f$ (with a dependence relation as depicted above) yields stacks as follows:



Regular Sets

A fundamental concept in formal languages is the notion of a *regular set*. Kleene's Theorem says that a regular set can be specified either by a finite deterministic (resp. nondeterministic) automaton DFA (resp. NFA) or, equivalently, by a regular expression. Regular expressions are also called *rational expressions*. They are defined inductively by saying that every finite set denotes a rational expression and if R, S is rational, then $R \cup S$, $R \cdot S$, and R^* are rational expressions, too. The semantics of a rational expression is defined in any monoid M since the semantics of $R \cup S$, $R \cdot S$ is obvious, and R^* can be viewed as the union $\bigcup_{k\in\mathbb{N}} R^k$. For *star-free* expressions one does not allow the star-operation, but one adds complementation, denoted, for example, by $\overline{R}$ with the semantics $M \setminus R$.

In trace theory a direct translation of Kleene's Theorem fails, but it can be replaced by a generalization due to Ochmański. If (a,b) is a pair of independent letters, then $(ab)^*$ is a rational expression, but due to $ab = ba$ it represents all strings with an equal number

of a's and b's which is clearly not regular. With three pairwise independent letters $(abc)^*$ is not even context-free. A general formal language theory distinguishes between recognizable and rational sets. A subset L of a trace monoid is called *recognizable*, if its closure is a regular word language. Here the closure refers to all words in Σ^*, which represent some trace in L. A subset L is called *rational*, if L can be specified by some regular (and hence rational) expression. Using the algebraic notion of homomorphism this can be rephrased as follows. Let φ be the canonical homomorphism of Σ^* onto $\mathbb{M}(\Sigma, I)$, which simply means the interpretation of a string as its trace. Now, L is recognizable if and only if $\varphi^{-1}(L)$ is a regular word language, and L is rational if and only if $L = \varphi(K)$ for some regular word language K. As a consequence of Kleene's Theorem all recognizable trace languages are rational, but the converse fails as soon as there is a pair of independent letters, that is, the trace monoid is not free.

Given a recognizable trace language L, the corresponding word language $\varphi^{-1}(L)$ is accepted by some NFA (actually some DFA), which satisfies the so-called *I-diamond property*. This means whenever it holds $(a, b) \in I$ and a state p leads to a state q by reading the word ab, then it is in state p also possible to read ba and this leads to state q, too. NFAs satisfying the I-diamond property accept closed languages only. Therefore they capture exactly the notion of recognizability for traces.

It has been shown that the concatenation of two recognizable trace languages is recognizable, in particular *star-free languages* (i.e., given by star-free expressions) are recognizable. However, the example $(ab)^*$ above shows that the star-operation leads to non-recognizable sets as soon as the trace monoid is not free. Métivier and Ochmański have introduced a restricted version where the star-operation is allowed only when applied to languages L where all traces $t \in L$ are connected. This means the dependence graph $G(t)$ is connected or, equivalently, there is no nontrivial factorization $t = uv$ where all letters in u are independent of all letters in v. A theorem shows that L^* is still recognizable, if L is connected (i.e., all $t \in L$ are connected) and recognizable. Ochmański's Theorem yields also the converse: A trace language L is recognizable if and only if it can be specified by a rational expression where the star-operation is restricted to connected subsets. As word languages are always connected this is a proper generalization of the classical Kleene's Theorem. Yet another characterization of recognizable trace languages is as follows: They are in one-to-one correspondence with regular subsets inside the regular set LexNF $\subseteq \Sigma^*$ of lexicographic normal forms. The correspondence associates with $L \subseteq \mathbb{M}(\Sigma, I)$ the set $K = \varphi^{-1}(L) \cap \text{LexNF}$. A rational expression for K is a rational expression for L, where the star-operation is restricted to connected languages.

Decidability Questions

The Star Mystery

The *Star Problem* is to decide for a given recognizable trace language $L \subseteq \mathbb{M}(\Sigma, I)$ whether L^* is recognizable. It is not known whether the star problem is decidable, even if it is restricted to finite languages L. The surprising difficulty of this problem has been coined as the *star mystery* by Ochmański. It has been shown by Richomme that the Star Problem is decidable, if (Σ, I) does not contain any C_4 (cycle of four letters) as an induced subgraph.

Undecidability Results for Rational Sets

For rational languages (unlike as for recognizable languages) some very basic problems are known to be undecidable. The following list contains undecidable decision problems, where the input for each instance consists of an independence alphabet (Σ, I) and rational trace languages $R, T \subseteq \mathbb{M}(\Sigma, I)$ specified by rational expressions.

- **Inclusion** question: Does $R \subseteq T$ hold?
- **Equality** question: Does $R = T$ hold?
- **Universality** question: Does $R = \mathbb{M}(\Sigma, I)$ hold?
- **Complementation** question: Is $\mathbb{M}(\Sigma, I) \setminus R$ a rational?
- **Recognizability** question: Is R recognizable?
- **Intersection** question: Does $R \cap T = \emptyset$ hold?

On the positive side, if I is transitive, then all six problems above are decidable. This is also a necessary condition for the first five problems in the list. Transitivity of the independence alphabet means in algebraic terms that the trace monoid is a free product of free and free commutative monoids, like, for example, $\{a, b\}^* * \mathbb{N}^3$.

The intersection problem is simpler. It is known that the problem Intersection is decidable if and only if (Σ, I)

is a transitive forest. It is also well known that transitive forests are characterized by forbidden induced subgraphs C_4 and P_4 (cycle and path, resp., of four letters).

Asynchronous Automata

Whereas recognizable trace languages can be defined as word languages accepted by DFAs or NFAs with I-diamond property, there is an equivalent distributed automaton model called *asynchronous automata*. Such an automaton is a parallel composition of finite-state processes synchronizing over shared variables, whereas a DFA satisfying the I-diamond property is still a device with a centralized control. An asynchronous automaton $\mathcal{A}$ has, by definition, a distributed finite state control such that independent actions may be performed in parallel. The set of global states is modeled as a direct product $Q = \prod_{p \in P} Q_p$, where the Q_p are states of the local component $p \in P$ and P is some finite index set (a set of processors). For each letter $a \in \Sigma$ there is a *read domain* $R(a) \subseteq P$ and a *write domain* $W(a) \subseteq P$ where for simplicity $W(a) \subseteq R(a)$. Processors p and q share a variable a if and only if $p, q \in R(a)$. The transitions are given by a family of partially defined functions δ_p, where each processor p reads the status in the local components of its read domain and changes states in local components of its write domain. Accordingly to the read-and-write-conflicts being allowed, four basic types are distinguished:

- Concurrent-Read-Exclusive-Write (*CREW*), if $R(a) \cap W(b) = \emptyset$ for all $(a, b) \in I$.
- Concurrent-Read-Owner-Write (*CROW*), if $R(a) \cap W(b) = \emptyset$ for all $(a, b) \in I$ and $W(a) \cap W(b) = \emptyset$ for all $a \neq b$.
- Exclusive-Read-Exclusive-Write (*EREW*), if $R(a) \cap R(b) = \emptyset$ for all $(a, b) \in I$.
- Exclusive-Read-Owner-Write (*EROW*), if $R(a) \cap R(b) = \emptyset$ for all $(a, b) \in I$ and $W(a) \cap W(b) = \emptyset$ for all $a \neq b$.

The local transition functions $\left(\delta_p\right)_{p \in P}$ give rise to a partially defined transition function on global states $\delta : \left(\prod_{p \in P} Q_p\right) \times \Sigma \longrightarrow \prod_{p \in P} Q_p$.

If $\mathcal{A}$ is of any of the four types above, then the action of a trace $t \in \mathbb{M}(\Sigma, I)$ on global states is well defined. This allows to see an asynchronous automaton as an I-diamond DFA. There are effective translations from one model to the other. The most compact versions can be obtained by a CREW model, therefore it is of prior practical interest.

Zielonka has shown in his thesis (published in 1987) the following deep theorem in trace theory: Every recognizable trace language can be accepted by some finite asynchronous automaton. The proof of this theorem is very technical and complicated. Moreover, the original construction was doubly exponential in the size of an I-diamond automaton for the language L. Therefore, it is part of ongoing research to simplify its construction, in particular since efficient constructions are necessary to make the result applicable in practice. The best result to date is due to Genest et al. They provide a construction where the size of the obtained asynchronous automaton is polynomial in the size of a given DFA and simply exponential in the number of processes. They also show that the construction is optimal within the class of automata produced by Zielonka-type constructions, which yields a nontrivial lower bound on the size of asynchronous automata.

A rather direct construction of asynchronous automata is known for triangulated dependence alphabets, which means that all chordless cycles are of length 3. For example, complete graphs and forests are triangulated.

Infinite Traces

The theory of infinite traces has its origins in the mid-1980s when Flé and Roucairol considered the problem of serializability of iterated transactions in data bases. A suitable definition of an infinite trace uses the dependence graph representation due to Mazurkiewicz. Just as in the finite case an infinite sequence $t = a_1 a_2 \cdots$ of letters yields an infinite node-labeled acyclic directed graph $G(t)$, where now each $i \in V = \mathbb{N}$ is labeled by the letter a_i, and again arcs $(i, j) \in E$ are introduced if and only if both $(a_i, a_j) \in D$ and $i < j$. It is useful to consider finite and infinite objects simultaneously as an infinite trace may split into connected components where some of them might be finite. The notion of *real trace* has been introduced to denote either a finite or an infinite trace. If $t_1, t_2, \ldots$ is (finite or infinite) sequence of finite traces, then the product $t_1 t_2 \cdots$ is a well-defined real trace. It is a finite trace if almost all t_i are empty and an infinite trace otherwise. In particular, one can define the ω-product L^ω for every set L of finite traces and one enriches the set of rational expressions by this operation.

The set $\mathbb{R}(\Sigma, I)$ of real traces can be embedded into a monoid of *complex traces* where the *imaginary* component is a subset of Σ. This alphabetic information is necessary in order to define an associative operation of concatenation. (Over complex traces L^ω is defined for all subsets L.)

Many results from the theory of finite traces transfer to infinite traces according to the same scheme as for finite and infinite words.

Logics

MSO and First-Order Logic

Formulae in monadic second-order logic (MSO) are built up upon first-order variables $x, y, \ldots$ ranging over vertices and second-order variables $X, Y, \ldots$ ranging over subsets of vertices. There are Boolean constants *true* and *false*, the logical connectives $\vee, \wedge, \neg$, and quantification $\exists$, $\forall$ for the first- and second-order variables. In addition there are four types of atomic formulae:

$$x \in X,\ x = y,\ (x, y) \in E, \text{ and } \lambda(x) = a.$$

A first-order formula is a formula without any second-order variable. A *sentence* is a closed formula, that is, a formula without free variables. The semantics of an MSO-sentence is defined for every node-labeled graph $[V, E, \lambda]$ (here: V = set of vertices, E = set of edges, $\lambda : V \to \Sigma$ = vertex labeling). Identifying a trace t with its dependence graph $G(t)$, the truth value of $t \vDash \psi$ is therefore well defined for every sentence ψ. The trace language defined by a sentence ψ is $L(\psi) = \{t \in \mathbb{R}(\Sigma, I) \mid t \vDash \psi\}$. It follows a notion of first-order and second-order definability of trace languages.

Temporal Logic

Linear temporal logic, LTL, can be inductively defined inside first order as formulae with one free variable, as soon as the transitive closure $(x, y) \in E^*$ is expressible in first order (as it is the case for trace monoids). There are no quantifiers, but all Boolean connectives. The atomic formulae are $\lambda(x) = a$. If $\varphi(x), \psi(x)$ are LTL-formulae, then $\mathsf{EX}\,\varphi(x)$ and $(\varphi\ \mathsf{U}\ \psi)(x)$ are LTL-formulae. In temporal logic $(x, y) \in E^*$ means that y is in the future of the node x. The semantics of $\mathsf{EX}\,\varphi(x)$ is *exists next*, thus $\varphi(y)$ holds for a direct successor of x. The semantics of $(\varphi\ \mathsf{U}\ \psi)(x)$ reflects an *until operator*, it says that in the future of x there is some z that satisfies $\psi(z)$ and all y in the future of x but in the strict past of z satisfy $\varphi(y)$. Hence, condition φ holds until ψ becomes true. There are dual past-tense operators, but they do not add expressivity.

For LTL one can also give a syntax without any free variable and a *global semantics* where the evaluation is based on the prefix relation of traces. The local semantics as defined above is for traces a priori expressively weaker, but it was shown that both, the global and local LTL have the same expressive power as first-order logic. This was done by Thiagarajan and Walukiewicz in 1998 for global LTL and by Diekert and Gastin in 2006 for local LTL, respectively. Both results extend a famous result of Kamp from words to traces. The complexity of the satisfiability problem (or model checking) is however quite different. In global semantics it is nonelementary, whereas in local semantics it is in PSPACE (= class of problems solvable on a Turing machine in polynomial space.)

Fragments

For various applications fragments of first-order logics suffice. This has the advantage that simpler constructions are possible and that the complexity of model checking is possibly reduced. A prominent fragment is first-order logic with at most two names for variables. Two-variable logics capture the core features of XML navigational languages like XPath. Over words and over traces two variable logic $\mathrm{FO}^2[E]$ can be characterized algebraically via the variety of monoids DA (referring to the fact that regular $\mathcal{D}$-classes are aperiodic semigroups), in logic by *Next-Future* and *Yesterday-Past* operators, and in terms of rational expressions via unambiguous polynomials. It turns out that the satisfiability problem for two-variable logic is NP-complete (if the independence alphabet is not part of the input). The extension of these results from words to traces is due to Kufleitner.

Logics, Algebra, and Automata

The connection between logic and recognizability uses algebraic tools from the theory of finite monoids. If $h : \mathbb{M}(\Sigma, I) \to M$ is a homomorphism to a finite monoid M and $L \subseteq \mathbb{R}(\Sigma, I)$ is a set of real traces, then one says that h recognizes L, if for all $t \in L$ and factorizations $t = t_1 t_2 \cdots$ into finite traces t_i the following inclusion holds: $h^{-1}(t_1)h^{-1}(t_2)\cdots \subseteq L$. This allows to speak of

aperiodic languages if some recognizing monoid is aperiodic. A monoid M is *aperiodic*, if for all $x \in M$ there is some $n \in \mathbb{N}$ such that $x^{n+1} = x^n$. A deep result states that a language is first-order definable if and only if it is recognized by a homomorphism to a finite aperiodic monoid. Algebraic characterizations lead to decidability of fragments. For example, it is decidable whether a recognizable language is aperiodic or whether it can be expressed in two-variable first-order logic.

Another way to define recognizability is via Büchi automata. A Büchi automaton for real traces is an I-diamond NFA with a set of final states F and a set of repeated states R. It accepts a trace if the run stops in F or if repeated states are visited infinitely often. If its transformation monoid is aperiodic it is called aperiodic, too. There is also a notion of asynchronous (cellular) Büchi automaton, and it is known that every I-diamond Büchi automaton can be transformed into an equivalent asynchronous cellular Büchi automaton.

The main result connecting logic, recognizability, rational expressions, and algebra can be summarized by saying that the following statements in the first block (second block resp.) are equivalent for all trace languages $L \subseteq \mathbb{R}(\Sigma, I)$:

MSO definability:

1. L is definable in monadic second-order logic.
2. L is recognizable by some finite monoid.
3. L is given as a rational expression where the star is restricted to connected languages.
4. L is accepted by some asynchronous Büchi automaton.

First-order definability:

1. L is definable in first-order logic.
2. L is definable in LTL (with global or local semantics).
3. L is recognizable by some finite and aperiodic monoid.
4. L is star-free.

Automata-Based Verification

The automata theoretical approach to verification uses the fact that systems and specifications are both modeled with finite automata. More precisely, a system is given as a finite transition system $\mathcal{A}$, which is typically realized as an NFA without final states. So, the system allows finite and infinite runs. The specification is written in some logical formalism, say in the linear temporal logic LTL. So the specification is given by some formula φ, and its semantics $L(\varphi)$ defines the runs that obey the specification. Model checking means to verify the inclusion $L(\mathcal{A}) \subseteq L(\varphi)$. This is equivalent to $L(\mathcal{A}) \cap L(\neg\varphi) = \emptyset$. Once an automaton $\mathcal{B}$ with $L(\mathcal{B}) = L(\neg\varphi)$ has been constructed, standard methods yield a product automaton for $L(\mathcal{A}) \cap L(\mathcal{B})$. The check for emptiness becomes a reachability problem in directed graphs.

A main obstacle is the combinatorial explosion when constructing the automaton $\mathcal{B}$. But this works in practice nevertheless reasonable well, because typical specifications are simple enough to be understood (hopefully) by the designer, so they are short. From a theoretical viewpoint the complexity of model checking for MSO and first order is nonelementary, but for (local) LTL is still in PSPACE. This approach is mostly applied and very successful where runs can be modeled as sequences. Trace theory provides the necessary tools to extend these methods to asynchronous systems. A first step in this direction has been implemented in the framework of *partial order* reduction. Another application of trace theory is the analysis of communication protocols.

Traces and Asynchronous Communication

Trace automata like asynchronous ones model concurrency in the same spirit as Petri nets, using shared variables. A more complex model arises when concurrent processes cooperate over unbounded, fifo communication channels.

A *communicating automaton* is defined over a set P of processes, together with point-to-point communication channels $Ch \subseteq \{(p,q) \in P^2 \mid p \neq q\}$. It consists of a tuple of NFAs $\mathcal{A}_p$, one for each process $p \in P$. Each NFA $\mathcal{A}_p$ has a set of local states Q_p and transition relation $\delta_p \subseteq Q_p \times \Sigma_p \times Q_p$. The set Σ_p of local actions of process p consists of send-actions $p!q(m)$ (of message m to process q, $(p,q) \in Ch$) and receive-actions $p?r(m)$ (of message m from process r, $(r,p) \in Ch$), respectively. The semantics of such an automaton is defined through configurations consisting of a tuple of local states (one for each process) and a tuple of word contents (one for each channel). In terms of partial orders the semantics of runs corresponds to *message sequence charts*

(*MSCs*), a graphical notation for fifo message exchange. In contrast with asynchronous automata, communicating automata have an infinite state space and are actually Turing powerful; thus, most algorithmic questions about them are undecidable.

The theory of recognizable trace languages enjoys various nice results known from word languages, for example, in terms of logics and automata. Since communicating automata are Turing powerful, one needs restrictions in order to obtain, for example, logical characterizations. A natural restriction consists in imposing bounds on the size of the channels. Such bounds come in two versions, namely, as *universal* and *existential* bounds, respectively. The existential version of channel bounds is optimistic and considers all those runs that can be rescheduled on bounded channels. The universal version is pessimistic and considers only those runs that, independent of the scheduling, can be executed with bounded channels. Thus, communicating automata with an universal channel bound are finite state, whereas with an existential channel bound they are infinite state systems.

Kuske proposed an encoding of runs of communicating automata with bounded channels into trace languages. Using this encoding, the set of runs (MSCs) of a communicating automaton is the projection of a recognizable trace language (for a universal bound), respectively the set of MSCs generated by the projection of a recognizable trace language (for an existential bound). This correspondence has the same flavor as the distinction between recognizable and rational trace languages, respectively.

The logic MSO over MSCs is defined with an additional binary message-predicate relating matching send and receive events. Henriksen et al. and Genest et al., respectively, have shown that the equivalence between MSO and automata extends to communicating automata with universal and existential channel bound, respectively. Another equivalent characterization exists in terms of MSC-graphs, similar to star-connected expressions for trace languages. These expressiveness results are complemented by decidable instances of the model-checking problem.

Related Entries

►Asynchronous Iterative Algorithms
►CSP (Communicating Sequential Processes)
►Formal Methods–Based Tools for Race, Deadlock, and Other Errors
►Multi-Threaded Processors
►Parallel Computing
►Parallelization, Automatic
►Peer-to-Peer
►Petri Nets
►Reordering
►Synchronization
►Trace Scheduling
►Verification of Parallel Shared-Memory Programs, Owicki-Gries Method of Axiomatic

Bibliographic Notes and Further Reading

Trace theory has its origin in enumerative combinatorics when Cartier and Foata found a new proof of the MacMahon Master Theorem in the framework of partial commutation by combining algebraic and bijective ideas [2]. The Foata normal form was defined in this Lecture Note. In computer science the key idea to use partial commutation as tool to investigate parallel systems was laid by Keller [10], but it was only by the influence of the technical report of Mazurkiewicz [11] when these ideas were spread to a wider computer science community, in particular to the Petri-net community. It was also Mazurkiewicz who coined the notion *Trace theory* and who introduced the notion of dependence graphs as a visualization of traces. The characterization of lexicographic normal forms by forbidden pattern is due to Anisimov and Knuth [1].

The investigation of recognizable (regular, rational resp.) languages is central in the theory of traces. The characterization of recognizable languages in terms of star-connected regular expressions is due to Ochmański [13]. The notion of *asynchronous automaton* is due to Zielonka. The major theorem showing that all recognizable languages can be accepted by asynchronous automata is his work (built on his thesis) [15]. The research on asynchronous automata is still an important and active area. The best constructions so far are due to Genest et al., where also nontrivial lower bounds were established [8].

The theory of infinite traces has its origin in the mid-1980s. A definition of a real trace as a prefix-closed and directed subset of real traces and its characterization by dependence graphs is given in a survey by

Mazurkiewicz [12]. The theory of recognizable real trace languages has been initiated by Gastin in 1990. The generalization of the Kleene–Büchi–Ochmański Theorem to real traces is due to Gastin, Petit, and Zielonka [7]. Diekert and Muscholl gave a construction for deterministic asynchronous Muller automata accepting a given recognizable real trace language.

Ebinger initiated the study of LTL for traces in his thesis in 1994. But it took quite an effort until Diekert and Gastin were able to show that LTL (in local semantics) has the same expressive power as first-oder logic [3]. The advantage of a local LTL is that model checking in PSPACE, whereas in its global semantics it becomes nonelementary by a result of Walukiewicz [14]. The PSPACE-containment has been shown for a much wider class of logics by Gastin and Kuske [6]. Diekert, Horsch, and Kufleitner [4] give a survey on fragments of first-order logic in trace theory. The Büchi-like equivalence between automata and MSO for existentially bounded communicating automata has been shown by Genest, Kuske, and Muscholl [9]. The translation from MSO into automata uses the equivalence for trace languages, but needs some additional, quite technical construction specific to communicating automata.

Very much of the material used in the present discussion can be found in The Book of Traces, which was edited by Diekert and Rozenberg [5]. The book surveys also a notion of *semi-commutation* (introduced by Clerbout and Latteux), and it provides many hints for further reading. Current research efforts concentrate on the topic of distributed games and controller synthesis for asynchronous automata.

Bibliography

1. Anisimov AV, Knuth DE (1979) Inhomogeneous sorting. Int J Comput Inf Sci 8:255–260
2. Cartier P, Foata D (1969) Problèmes combinatoires de commutation et réarrangements. Lecture notes in mathematics, vol 85. Springer, Heidelberg
3. Diekert V, Gastin P (2006) Pure future local temporal logics are expressively complete for Mazurkiewicz traces. Inf Comput 204:1597–1619. Conference version in LATIN 2004, LNCS 2976: 170–182, 2004
4. Diekert V, Horsch M, Kufleitner M (2007) On first-order fragments for Mazurkiewicz traces. Fundamenta Informaticae 80:1–29
5. Diekert V, Rozenberg G (eds) (1995) The book of traces. World Scientific, Singapore
6. Gastin P, Kuske D (2007) Uniform satisfiability in pspace for local temporal logics over Mazurkiewicz traces. Fundam Inf 80(1–3): 169–197
7. Gastin P, Petit A, Zielonka WL (2007) An extension of Kleene's and Ochmański's theorems to infinite traces. Theoret Comput Sci 125:167–204, x
8. Genest B, Gimbert H, Muscholl A, Walukiewicz I (2010) Optimal Zielonka-type construction of deterministic asynchronous automata. In: Abramsky S, Gavoille C, Kirchner C, Meyer auf der Heide F, Spirakis PG (eds) ICALP (2). Lecture notes in computer science, vol 6199. Springer, pp 52–63
9. Genest B, Kuske D, Muscholl A (2006) A Kleene theorem and model checking algorithms for existentially bounded communicating automata. Inf Comput 204:926–956. http://dx.doi.org/10.1016/j.ic.2006.01.005;DBLP, http://dblp.uni-trier.de
10. Keller RM (1973) Parallel program schemata and maximal parallelism I. Fundamental results. J Assoc Comput Mach 20(3):514–537
11. Mazurkiewicz A (1977) Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus
12. Mazurkiewicz A (1987) Trace theory. In: Brauer W et al. (eds) Petri nets, applications and relationship to other models of concurrency. Lecture notes in computer science, vol 255. Springer, Heidelberg, pp 279–324
13. Ochmański E (Oct 1985) Regular behaviour of concurrent systems. Bull Eur Assoc Theor Comput Sci (EATCS) 27:56–67
14. Walukiewicz I (1998) Difficult configurations – on the complexity of LTrL. In: Larsen KG, et al. (eds) Proceedings of the 25th International Colloquium Automata, Languages and Programming (ICALP'98), Aalborg (Denmark). Lecture notes in computer science, vol 1443. Springer, Heidelberg, pp 140–151
15. Zielonka WL (1987) Notes on finite asynchronous automata. R.A.I.R.O. Informatique Théorique et Applications 21:99–135

Tracing

►Performance Analysis Tools
►Scalasca
►TAU

Transactional Memories

Maurice Herlihy
Brown University, Providence, RI, USA

Synonyms

Locks; Monitors; Multiprocessor synchronization

Introduction

Transactional memory (TM) is an approach to structuring concurrent programs that seeks to provide better scalability and ease-of-use than conventional approaches based on locks and conditions. The term is

commonly used to refer to ideas that range from programming language constructs to hardware architecture. This entry will survey how transactional memory affects each of these domains.

The major chip manufacturers have, for the time being, given up trying to make processors run faster. Moore's law has not been repealed: Each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. Instead, attention has turned toward *chip multiprocessing* (CMP), in which multiple computing cores are included on each processor chip. In the medium term, advances in technology will provide increased parallelism, but not increased single-thread performance. As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must learn to make more effective use of increasing parallelism.

This adaptation will not be easy. Conventional programming practices typically rely on combinations of locks and conditions, such as monitors [1], to prevent threads from concurrently accessing shared data. Locking makes concurrent programming possible because it allows programmers to reason about certain code sections as if they were executed atomically. Nevertheless, the conventional approach suffers from a number of shortcomings.

First, programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock, and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is relatively easy to use, but permits little or no concurrency, thereby preventing the program from exploiting multiple cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlock when acquiring multiple locks. Such designs are further complicated because the most efficient engineering solution may be platform dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

Second, locking provides poor support for code composition and reuse. For example, consider a lock-based queue that provides atomic `enq()` and `deq()` methods. Ideally, it should be easy to transfer an item atomically from one queue to another, but such elementary composition simply does not work. It is necessary to lock both queues at the same time to make the transfer atomic. If the queue methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the queues export their locks, then modularity and safety are compromised, because the integrity of the objects depends on whether their users follow *ad hoc* conventions correctly.

Finally, such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make concurrent programs too difficult to develop, debug, understand, and maintain.

The Transactional Model

A *transaction* is a sequence of steps executed by a single thread. Transactions are *atomic*: Each transaction either commits (it takes effect) or aborts (its effects are discarded). Transactions are linearizable [2]: They appear to take effect in a one-at-a-time order. Transactional memory supports a computational model in which each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. If the commit succeeds, the transaction's operations take effect; otherwise, they are discarded.

Sometimes we refer to these transactions as *memory transactions*. Memory transactions satisfy the same formal serializability and atomicity properties as the transactions used in conventional database systems, but they are intended to address different problems.

Unlike database transactions, memory transactions are short-lived activities that access a relatively small number of objects in primary memory. Database transactions are *persistent*: When a transaction commits, its changes are backed up on a disk. Memory transactions need not be persistent, and involve no explicit disk I/O.

To illustrate why memory transactions are attractive from a software engineering perspective, consider the problem of constructing a concurrent FIFO queue that permits one thread to enqueue items at the tail of

the queue at the same time another thread dequeues items from the head of the queue, at least while the queue is non-empty. Any problem so easy to state, and that arises so naturally in practice, should have an easily devised, understandable solution. In fact, solving this problem with locks is quite difficult. In 1996, Michael and Scott published a clever and subtle solution [3]. It speaks poorly for fine-grained locking as a methodology that solutions to such simple problems are challenging enough to be publishable.

By contrast, it is almost trivial to solve this problem using transactions. Figure 1 shows how the queue's enqueue method might look in a language that provides direct support for transactions. It consists of little more than enclosing sequential code in a transaction

```
class Queue<T> {
  QNode head;
  Qnode tail;
  public void enq(T x) {
    atomic {
      Qnode q = new Qnode(x);
      if (tail == null) {   // empty queue
        head = tail = q;
      } else {
        tail.next = q;
        tail = q;
    }
  }
  public T deq() {
    atomic {
      if (head == null)
        retry;
      T item = head.item;
      head = head.next;
      if (head == null)
        tail = null;
      return item;
    }
  }
  ...
}
```

Transactional Memories. Fig. 1 Transactional queue code fragment

```
atomic {
  x = q0.deq();
} orElse {
  x = q1.deq();
}
```

Transactional Memories. Fig. 2 The orElse statement: waiting on multiple conditions

block. In practice, of course, a complete implementation would include more details (such as how to respond to an empty queue), but even so, this concurrent queue implementation is a remarkable achievement: It is not, by itself, a publishable result.

Conditional synchronization can be accomplished in the transactional model by means of the retry construct [4]. As illustrated in Fig. 1, if a thread attempts to dequeue from an empty queue, it executes retry, which rolls back the partial effects of the atomic block, and re-executes that block later when the object's state has changed. The retry construct is attractive because it is not subject to the *lost wake-up* bug that can arise using monitor conditions.

Transactions also admit compositions that would be impossible using locks and conditions. Waiting for one of several conditions to become *true* is impossible using objects with internal monitor condition variables. A novel aspect of retry is that such composition becomes easy. Figure 2 shows a code snippet illustrating the orElse statement, which joins two or more code blocks. Here, the thread executes the first block. If that block calls retry, then that subtransaction is rolled back, and the thread executes the second block. If that block also calls retry, then the orElse as a whole pauses, and later reruns each of the blocks (when something changes) until one completes.

Motivation

TM is commonly used to address three distinct problems: first, a simple desire to make highly concurrent data structures easy to implement; second, a more ambitious desire to support well-structured large-scale concurrent programs; and third, a pragmatic desire to make conventional locking more concurrent. Here is a survey of each area.

Lock-Free Data Structures

A data structure is *lock-free* if it guarantees that infinitely often *some* method call finishes in a finite number of steps, even if some subset of the threads halt in arbitrary places. A data structure that relies on locking cannot be lock-free because a thread that acquires a lock and then halts can prevent non-faulty threads from making progress.

Lock-free data structures are often awkward to implement using today's architectures which typically rely on *compare-and-swap* for synchronization. The *compare-and-swap* instruction takes three arguments, and *address a*, an *expected* value *e*, and an *update* value *u*. If the value stored at *a* is equal to *e*, then it is atomically replaced with *u*, and otherwise it is unchanged. Either way, the instruction sets a flag indicating whether the value was changed.

Often, the most natural way to define a lock-free data structure is to make an atomic change to several fields. Unfortunately, because *compare-and-swap* allows only one word (or perhaps a small number of contiguous words) to be changed atomically, designers of lock-free data structures are forced to introduce complex multistep protocols or additional levels of indirection that create unwelcome overhead and conceptual complexity. The original TM paper [5] was primarily motivated by a desire to circumvent these restrictions.

Software Engineering

TM is appealing as a way to help programmers structure concurrent programs because it allows the programmer to focus on what the program should be doing, rather than on the detailed synchronization mechanisms needed. For example, TM relieves the programmer of tasks such as devising specialized locking protocols for avoiding deadlocks, and conventions associating locks with data.

A number of programming languages and libraries have emerged to support TM. These include Clojure [6], .Net [7], Haskell [4]. Java [8, 9], C++ [10], and others.

Several groups have reported experiences converting programs from locks to TM. The TxLinux [11] project replaced most of the locks in the Linux kernel with transactions. Syntactically, each transaction appears to be a lock-based critical section, but that code is executed speculatively as a transaction (see Section 3.3). If an I/O call is detected, the transaction is rolled back and restarted using locks. Using transactions primarily as an alternative way to implement locks minimized the need to rewrite and restructure the original application.

Damron et al. [12] transactionalized the Berkeley DB lock manager. They found the transformation more difficult than expected because simply changing critical sections into atomic blocks often resulted in a disappointing level of concurrency. Critical sections often shared data unnecessarily, usually in the form of global statistics or shared memory pools. Later on, we will see other work that reinforces the notion the need to avoid *gratuitous conflicts* means that concurrent transactional programs must be structured differently than concurrent lock-based programs.

Pankratius et al. [13] conducted a user study where twelve students, working in pairs, wrote a parallel desktop search engine. Three randomly chosen groups used a compiler supporting TM, and three used conventional locks. The best TM group were much faster to produce a prototype, the final program performed substantially better, and they reported less time spent on debugging. However, the TM teams found performance harder to predict and to tune. Overall, the TM code was deemed easier to understand, but the TM teams did still make some synchronization errors.

Rossbach et al. [14] conducted a user study in which 147 undergraduates implemented the same programs using coarse-grained and fine-grained locks, monitors, and transactions. Many students reported they found transactions harder to use than coarse-grain locks, but slightly easier than fine-grained locks. Code inspection showed that students using transactions made many fewer synchronization errors: Over 70% of students made errors with fine-grained locking, while less than 10% made errors using transactions.

Lock Elision

Transactions can also be used as a way to implement locking. In *lock elision* [15], when a thread requests a lock, rather than waiting to acquire that lock, the thread starts a speculative transaction. If the transaction commits, then the critical section is complete. If the transaction aborts because of a synchronization conflict, then the thread can either retry the transaction, or it can actually acquire the lock.

Here is why lock elision is attractive. Locking is conservative: A thread must acquire a lock if it *might* conflict with another thread, even if such conflicts are rare. Replacing lock acquisition with speculative execution enhances concurrency if actual conflicts are rare. If conflicts persist, the thread can abandon speculative execution and revert to using locks. Lock elision has the added advantage that it does not require code to be restructured. Indeed, it can often be made to work with legacy code.

Azul Systems [16] has a JVM that uses (hardware) lock elision for contended Java locks, with the goal of accelerating "dusty deck" Java programs. The run-time system keeps track of how well the hardware transactional memory (HTM) is doing, and decides when to use lock elision and when to use conventional locks. The results work well for some applications, modestly well for others, and poorly for a few. The principal limitation seems to be the same as observed by Damron et al. [12]: many critical sections are written in a way that introduces gratuitous conflicts, usually by updating performance counters. Although these are not real conflicts, the HTM has no way to tell. Rewriting such code can be effective, but requires abandoning the goal of speeding up "dusty deck" programs.

Hardware Transactional Memory

Most hardware transactional memory (HTM) proposals are based on straightforward modifications to standard multiprocessor cache-coherence protocols. When a thread reads or writes a memory location on behalf of a transaction, that cache entry is flagged as being transactional. Transactional writes are accumulated in the cache or write buffer, but are not written back to memory while the transaction is active. If another thread invalidates a transactional entry, a data conflict has occurred, that transaction is aborted and restarted. If a transaction finishes without having had any of its entries invalidated, then the transaction commits by marking its transactional entries as valid or as dirty, and allowing the dirty entries to be written back to memory in the usual way.

One limitation of HTM is that in-cache transactions are limited in size and scope. Most hardware transactional memory proposals require programmers to be aware of platform-specific resource limitations such as cache and buffer sizes, scheduling quanta, and the effects of context switches and process migrations. Different platforms provide different cache sizes and architectures, and cache sizes are likely to change over time. Transactions that exceed resource limits or are repeatedly interrupted will never commit. Ideally, programmers should be shielded from such complex, platform-specific details. Instead, TM systems should provide full support even for transactions that cannot execute directly in hardware.

Techniques that substantially increase the size of hardware transactions include signatures [17] and permissions-only caches [18]. Other proposals support (effectively) unbounded transactions by allowing transactional metadata to overflow caches, and for transactions to migrate from one core to another. These proposals include TCC [19], VTM [20], OneTM [18], UTM [21], TxLinux [11], and LogTM [17].

Software Transactional Memory

Software transactional memory (STM) is an alternative to direct hardware support for TM. STM is a software system that provides programmers with a transactional model through a library or compiler interface. In this section, we describe some of the questions that arise when designing an STM system. Some of these questions concern *semantics*, that is, how the STM behaves, and other concern *implementation*, that is, how the STM is structured internally.

Weak vs Strong Isolation

How should threads that execute transactions interact with threads executing non-transactional code? One possibility is *strong isolation* [22] (sometimes called *strong atomicity*), which guarantees that transactions are atomic with respect to non-transactional accesses. The alternative, *weak isolation* (or *weak atomicity*), makes no such guarantees. HTM systems naturally provide strong atomicity. For STM systems, however, strong isolation may be too expensive.

The distinction between strong and weak isolation leaves unanswered a number of other questions about STM behavior. For example, what does it mean for an unhandled exception to exit an atomic block? What does I/O mean if executed inside a transaction? One appealing approach is to say that transactions behave as if they were protected by a *single global lock* (SGL) [19, 23, 24].

One limitation of the SGL semantics is that it does not specify the behavior of *zombie* transactions: transactions that are doomed to abort because of synchronization conflicts, but continue to run for some duration before the conflict is discovered. In some STM implementations, zombie transactions may see an inconsistent state before aborting. When a zombie aborts, its effects are rolled back, but while it runs, observed inconsistencies could provoke it to pathological behavior that may be difficult for the STM system to protect against, such as dereferencing a null pointer or entering an infinite loop. *Opacity* [25] is a correctness condition that guarantees that all uncommitted transactions, including zombies, see consistent states.

I/O and System Calls

What does it mean for a transaction to make a system call (such as I/O) that may affect the outside world? Recall that transactions are often executed speculatively, and a transaction that encounters a synchronization conflict may be rolled back and restarted. If a transaction creates a file, opens a window, or has some other external side effect, then it may be difficult or impossible to roll everything back.

One approach is to allow *irrevocable* transactions [11, 18, 26] that are not executed speculatively, and so never need to be undone. An irrevocable transaction cannot explicitly abort itself, and only one such transaction can run at a time, because of the danger that multiple irrevocable transactions could deadlock.

An alternative approach is to provide a mechanism to escape from the transactional system. Escape actions [27] and open nested transactions citeNi07 allow a thread to execute statements outside the transaction system, scheduling application-specific commit and abort handlers to be called if the enclosing transaction commits or aborts. For example, an escape action might create a file, and register a handler to abort that file if the transaction aborts. Escape mechanisms can be misused, and often their semantics are not clearly defined. Using open nested transactions, for example, care must be taken to ensure that abort handlers do not deadlock.

Exploiting Object Semantics

STM systems typically synchronize on the basis of *read/write conflicts*. As a transaction executes, it records the data items it read in a *read set*, and the data items it wrote in a *write set*. Two transactions *conflict* if one transaction's read or write set intersects the other's write set. Conflicting transactions cannot both commit.Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: It can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

This problem can be addressed by open nested transactions, as described above in Section 5.2, but open nested transactions are difficult to use correctly, and lack the expressive power to deal with certain common cases [28].

Another approach is to use type-specific synchronization and recovery to exploit concurrency inherent in an object's high-level specification. One such mechanism is *transactional boosting* [28], which allows thread-safe (but non-transactional) object implementations to be transformed into highly concurrent transactional implementations by allowing method calls to proceed in parallel as long as their high-level specifications are *commutative*.

Eager vs Lazy Update

There are two basic ways to organize transactional data. In an *eager* update system, data objects are modified in place, and each transaction maintains an *undo log* allowing it to undo its changes if it aborts. The dual approach is *lazy* (or deferred) update, where each transaction computes optimistically on its local copy of the data, installing the changes if it commits, and discarding them if it aborts. An eager system makes committing a transaction more efficient, but makes it harder to ensure that zombie transactions see consistent states.

Eager vs Lazy Conflict Detection

STM systems differ according to when they detect conflicts. In *eager* conflict detection schemes, conflicts are detected before they arise. When one transaction is about to create a conflict with another, it may consult a contention manager, defined below, to decide whether to pause, giving the other transaction a chance to finish, or to proceed and cause the other to abort. By contrast,

a *lazy* conflict detection scheme detects conflicts when a transaction tries to commit. Eager detection may abort transactions that could have committed lazily, but lazy detection discards more computation, because transactions are aborted later.

Contention Managers

In many STM proposals, conflict resolution is the responsibility of a *contention manager* [29] module. Two transactions *conflict* if they access the same object and one access is a write. If one transaction discovers it is about to conflict with another, then it can pause, giving the other a chance to finish, or it can proceed, forcing the other to abort. Faced with this decision, the transaction consults a contention management module that encapsulates the STM's conflict resolution policy.

The literature includes a number of contention manager proposals [29–32], ranging from exponential backoff to priority-based schemes. Empirical studies have shown that the choice of a contention manager algorithm can affect transaction throughput, sometimes substantially.

Visible vs Invisible Reads

Early STM systems [29] used either *invisible reads*, in which each transaction maintains per-read metadata to be revalidated after each subsequent read, or *visible reads*, in which each reader registers its operations in shared memory, allowing a conflicting writer to identify when it is about to create a conflict. Invisible read schemes are expensive because of the need for repeated validation, while visible read schemes were complex, expensive, and not scalable.

More recent STM systems such as TL2 [33] or SKYSTM [34] use a compromise solution, called *semi-visible reads*, in which read operations are tracked imprecisely. Semi-visible reads conservatively indicate to the writer that a read-write conflict might exist, avoiding expensive validation in the vast majority of cases.

Privatization

It is sometimes useful for a thread to *privatize* [35] a shared data structure by making it inaccessible to other threads. Once the data structure has been privatized, the owning thread can work on the data structure directly, without incurring synchronization costs. In principle, privatization works correctly under SGL semantics, in which every transaction executes as if it were holding a "single global lock." Unfortunately, care is required to ensure that privatization works correctly. Here are two possible hazards. First, the thread that privatizes the data structure must observe all changes made to that data by previously committed transactions, which is not necessarily guaranteed in an STM system where updates are lazy. Second, a doomed ("zombie") transaction must not be allowed to perform updates to the data structure after it has been privatized.

Bibliographic Notes and Further Reading

The most comprehensive TM survey is the book *Transactional Memory* by Larus and Rajwar [23]. Of course, this area changes rapidly, and the best way to keep up with current developments is to consult the the *Transactional Memory Online* web page at: http://www.cs.wisc.edu/trans-memory/.

Bibliography

1. Hoare CAR (1974) Monitors: an operating system structuring concept. Commun ACM 17(10):549–557
2. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM T Progr Lang Sys 12(3):463–492
3. Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, Philadelphia. ACM, New York, pp 267–275
4. Harris T, Marlow S, Peyton-Jones S, Herlihy M (2005) Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming, Chicago. ACM, New York, pp 48–60
5. Herlihy M, Moss JEB (May 1993) Transactional memory: architectural support for lock-free data structures. In: International symposium on computer architecture, San Diego
6. Hickey R (2008) The clojure programming language. In: DLS '08: Proceedings of the 2008 symposium on dynamic languages, Paphos. ACM, New York, pp 1–1
7. Microsoft Corporation. Stm.net. http://msdn.microsoft.com/en-us/devlabs/ee334183.aspx
8. Korland G Deuce STM. http://www.deucestm.org/
9. S. Microsystems. DSTM2. http://www.sun.com/download/products.xml?id=453fb28e
10. Intel Corporation. C++ STM compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/
11. Rossbach CJ, Hofmann OS, Porter DE, Ramadan HE, Aditya B, Witchel E (2007) TxLinux: using and managing hardware transactional memory in an operating system. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, Stevenson. ACM, New York, pp 87–102

12. Damron P, Fedorova A, Lev Y, Luchangco V, Moir M, Nussbaum D (2006) Hybrid transactional memory. In: ASPLOS-XII: Proceedings of the 12th international conference on architectural support for programming languages and operating systems, Boston. ACM, New York, pp 336–346
13. Pankratius V, Adl-Tabatabai A-R, Otto F (Sept 2009) Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe
14. Rossbach CJ, Hofmann OS, Witchel E (Jun 2009) Is transactional memory programming actually easier? In: Proceedings of the 8th annual workshop on duplicating, deconstructing, and debunking (WDDD), Austin
15. Rajwar R, Goodman JR (2001) Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on microarchitecture, Austin. IEEE Computer Society, Washington, DC, pp 294–305
16. Click C (Feb 2009) Experiences with hardware transactional memory. http://blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html
17. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) LogTM-SE: decoupling hardware transactional memory from caches. In: HPCA '07: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture, Phoenix. IEEE Computer Society, Washington, DC, pp 261–272
18. Blundell C, Devietti J, Lewis EC, Martin M (Jun 2007) Making the fast case common and the uncommon case simple in unbounded transactional memory. In: International symposium on computer architecture, San Diego
19. Hammond L, Carlstrom BD, Wong V, Hertzberg B, Chen M, Kozyrakis C, Olukotun K (2004) Programming with transactional coherence and consistency (TCC). ACM SIGOPS Oper Syst Rev 38(5):1–13
20. Rajwar R, Herlihy M, Lai K (Jun 2005) Virtualizing transactional memory. In: International symposium on computer architecture, Madison
21. Ananian CS, Asanović K, Kuszmaul BC, Leiserson CE, Lie S (Feb 2005) Unbounded transactional memory. In: Proceedings of the 11th international symposium on high-performance computer architecture (HPCA'05), San Franscisco, pp 316–327
22. Blundell C, Lewis EC, Martin MMK (Jun 2005) Deconstructing transactions: the subtleties of atomicity. In: Fourth annual workshop on duplicating, deconstructing, and debunking, Wisconsin
23. Larus J, Rajwar R (2007) Transactional memory (Synthesis lectures on computer architecture). Morgan & Claypool, San Rafael
24. Menon V, Balensiefer S, Shpeisman T, Adl-Tabatabai A-R, Hudson RL, Saha B, Welc A (2008) Single global lock semantics in a weakly atomic STM. SIGPLAN Notices 43(5):15–26
25. Guerraoui R, Kapalka M (2008) On the correctness of transactional memory. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City. ACM, New York, pp 175–184
26. Welc A, Saha B, Adl-Tabatabai A-R (2008) Irrevocable transactions and their applications. In: SPAA '08: Proceedings of the twentieth annual symposium on parallelism in algorithms and architectures, Munich. ACM, New York, pp 285–296
27. Moravan MJ, Bobba J, Moore KE, Yen L, Hill MD, Liblit B, Swift MM, Wood DA (2006) Supporting nested transactional memory in logTM. SIGPLAN Notices 41(11):359–370
28. Herlihy M, Koskinen E (2008) Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, Salt Lake City. ACM, New York, pp 207–216
29. Herlihy M, Luchangco V, Moir M, Scherer W (Jul 2003) Software transactional memory for dynamic-sized data structures. In: Symposium on principles of distributed computing, Boston
30. Guerraoui R, Herlihy M, Pochon B (2005) Toward a theory of transactional contention managers. In: PODC '05: Proceedings of the twenty-fourth annual ACM symposium on principles of distributed computing, Las Vegas. ACM, New York, pp 258–264
31. Scherer WN III, Scott ML (Jul 2004) Contention management in dynamic software transactional memory. In: PODC workshop on concurrency and synchronization in java programs, St. John's
32. Attiya H, Epstein L, Shachnai H, Tamir T (2006) Transactional contention management as a non-clairvoyant scheduling problem. In: PODC '06: Proceedings of the twenty-fifth annual ACM symposium on principles of distributed computing, Denver. ACM, New York, pp 308–315
33. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: Proceedings of the 20th international symposium on distributed computing, Stockholm
34. Lev Y, Luchangco V, Marathe V, Moir M, Nussbaum D, Olszewski M (2009) Anatomy of a scalable software transactional memory. In: TRANSACT 2009, Raleigh
35. Spear MF, Marathe VJ, Dalessandro L, Scott ML (2007) Privatization techniques for software transactional memory. In: PODC '07: Proceedings of the twenty-sixth annual ACM symposium on principles of distributed computing, Portland. ACM, New York, pp 338–339

Transactions, Nested

J. Eliot B. Moss
University of Massachusetts, Amherst, MA, USA

Synonyms

Multi-level transactions; Nested spheres of control

Definition

Nested Transactions extend the traditional semantics of transactions by allowing meaningful nesting of one or

more child transactions within a parent transaction. Considering the traditional ACID properties of transactions, atomicity, consistency, isolation, and durability, a child transaction possesses these properties in a relative way, such that a parent transaction effectively provides a universe within which its children act similarly to ordinary transactions in a non-nested system. In parallel computation, the traditional property of durability in the face of various kinds of system failures may not be required.

Discussion

Transactions

Transactions are a way of guaranteeing atomicity of more or less arbitrary sequences of code in a parallel computation. Unlike locks, which identify computations that may need serialization according to the identity of held and requested locks, transaction serialization is based on the specific data accessed by a transaction while it runs. In some cases, it is possible usefully to pre-declare the maximal set of data that a transaction might access, but it is not possible in general. Transactions *specify semantics*, while locks express an implementation of serialization. The usual semantics of transactions are that concurrent execution of a collection of transactions must be equivalent to execution of those transactions one at a time, in some order. This property is called *serializability*.

The ACID properties capture transaction semantics in a slightly different way. *Atomicity* requires that transactions are all-or-nothing: Either all of a transaction's effects occur, or the transaction fails and has no effect. *Consistency* requires that each transaction take the state of the world from one consistent state to another. *Isolation* requires that no transaction perceive any state in the middle of execution of another transaction. *Durability* requires that the effects of any transaction, once the transaction is accepted by the system, not disappear.

In general, in parallel computing, as opposed to database systems where transactions had their origins, the consistency and durability properties are often less important. Consistency may be ignored in that most commonly there are no explicitly stated consistency constraints and no mechanism to enforce them. However, a correct transaction system is still required not to leave effects of partially executed or failed transactions. Durability is often ignored in that a parallel computing system may have no permanent state. If the system has distributed memory, then it may achieve significant durability by keeping multiple copies of data in different units of the distributed memory. Of course, parallel systems can also use one or more nonvolatile copies to achieve durability, according to a system's durability requirements.

It is important to distinguish transactions from concurrency-safe data structures. A concurrency-safe data structure generally offers a guarantee of *linearizability:* If two actions on the data structure by different threads overlap in their execution, then the effect is as if the actions are executed in one order or the other. That is, a set of concurrent actions by different threads appears to occur in some linear order. This is *similar* to serializability, but what transactions and serializability add is the possibility for a given thread to execute a whole *sequence* of actions a_1, a_2, ..., a_n without any intervening actions of other threads. A concurrency-safe data structure guarantees only that the individual a_i execute correctly as defined by the data type, but permits actions of other threads to interleave between the a_i.

In discussing transactions and their nesting, some additional terms will be useful. Transactions are said to *commit* (succeed) or *abort* (fail). They may fail for many reasons, one of them being serialization conflicts with other transactions. Transactional concurrency control may be *pessimistic*, also called early conflict detection, or *optimistic*, also called late conflict detection. Pessimistic conflict detection usually employs some kind of locks, while optimistic generally uses some kind of version numbers or timestamps on data and transactions to determine conflicts. It is even possible to maintain multiple versions so as to allow more transactions to commit, while still enforcing serializability. In general, locking schemes require some kind of deadlock avoidance or detection protocol. However, if a set of transactions is in deadlock, the system has a way out: It can abort one of the transactions to break the deadlock. Thus, deadlock is not a fatal problem as it is when using just locks for synchronization.

A system may update data *in place*, which requires an *undo log* to support removing the effects of a failed transaction. Alternatively, a system may create *new copies* of data, and install them only if a transaction commits, which in general requires a *redo log*. Updating in-place requires early conflict detection if the system guarantees that a transaction will not see effects of other uncommitted transactions. It is possible to allow such effects to be visible, but serializability then requires that the observing transaction commit only if the transaction it observed also commits. However, the system must still prevent two transactions from observing each other's effects, since then they cannot be serialized. More advanced models have also been explored but are not discussed here.

Semantics of Nesting

The simplest motivation for nesting is to make it easy to compose software components into larger systems. If a library routine uses transactions, and the programmer wishes to use that routine within an application transaction, then there will be nesting of transaction begin/end pairs. One can simply treat this as one large transaction, effectively ignoring the inner transaction begin/end pairs. However, it is also possible to attribute transactional semantics to them, as follows.

Consider a transaction T and a transaction U contained with it, i.e., whose begin and end are between the begin and end of T. T is a *parent* transaction and U a *child transaction* or *subtransaction* of T. If T is not contained within any enclosing transaction, it is *top-level*. Only proper nesting of begin/end transaction pairs is legal, so a top-level transaction and subtransactions at all depths form a tree.

Conflict semantics of non-nested transactions extend to nested transactions straightforwardly in terms of relationships in the forest of transaction trees. If action A conflicts with action B when executed by two different non-nested transactions T1 and T2, then A conflicts with B in the nested setting if neither of T1 and T2 is an ancestor of the other. Why is there no conflict in the case where, say, T1 is an ancestor of T2? It is because T1 is providing an environment or universe *within which* additional transactions can run, compete, and be serialized. It is simplest, however, to envision nesting where a parent does not execute actions directly, but rather always creates a child transaction to perform them. Alternatively, a parent might perform actions directly, but only when it has no active subtransactions. This model leads to serializability among the subtransactions of any given transaction T. However, as viewed from outside of the transaction tree that includes T and its descendants, T and its subtransactions form a single transaction that must itself be serializable with transactions outside of T.

As with non-nested transactions, a nested transaction can fail (abort). In that case, it is as if the failing transaction, and all of its descendants, never ran. Thus, commit of a child transaction is not final, but only relative to its parent, while abort of the parent *is* final and aborts all descendants, even those that have (provisionally) committed.

Example Closed Nesting Implementation Approach

Consider adding support for nesting to a non-nested transaction implementation that employs in-place update and early conflict detection based on locking. As each transaction runs, it accumulates a set of locks and a list of undos. If the transaction aborts, the system applies the transaction's undos (in reverse order) and then discards the transaction's locks. Note that a transaction T can be granted a lock L provided that the only conflicting holders of L are ancestors of T. Also note that discarding a child's lock does not discard any ancestor's lock on the same item. If a child transaction commits, then the system adds the child's locks to those held by the parent, and appends the child's undo list to that of the parent. If a top-level transaction commits, the system simply discards its held locks and its undo list. Moss [5, 6] described this protocol. It is also possible to devise timestamp-based approaches, possibly supporting late conflict detection, as articulated by Reed [10].

Notice that these nesting schemes use the same set of possible actions at each level of nesting. They provide *temporal grouping* of actions, and in a distributed

system can also be used for *spatial grouping* within temporal groups.

Motivations for Closed Nesting

There are two primary advantages of closed nesting. One is that failure of a child does not require immediate failure of its parent. Thus, if a transaction desires to execute an action that has higher than usual likelihood of causing failure, it can execute that action within a child transaction and avoid immediate failure of itself should the action cause an abort. In a centralized system aborts might most likely be caused by conflicts with other transactions, but in a decentralized system, failure of a remote node or communication link is also possible. Thus, remote calls are natural candidates to execute as subtransactions. If a child does fail, the parent can retry it, which may often make sense, or the parent can perform some alternate action. For example, in a distributed system, if one node of a replicated database is down, the parent could try another one.

A possibly stronger motivation for closed nesting is safe transaction execution when the application desires to exploit concurrency *within* a transaction. It is easy to see this by considering that if there is concurrency within a transaction, then proper semantics and synchronization or serialization within the transaction present the same issues that led to proposing transaction mechanisms in the first place. Even if, at first blush, it appears that the space of data that concurrent actions might update is disjoint, and thus that there can be no conflict, that property can be a delicate one, and difficult to enforce in complex software systems having many layers. For example, transaction T at node A might make apparently disjoint concurrent remote calls to nodes B and C. However, B and C, unknown to T, use a common service at node D, and should have their actions properly serialized there. If the work at B and C is not performed in distinct concurrent child transactions of T, the work at D might not be properly serialized.

Open Nesting

While closed nested transactions indeed support safe concurrency within transactions, and also offer limited recoverability from partial failure, they have a significant limitation: Transactions that are "big," either in terms of how long they run or the volume of data they access, tend to conflict with other transactions. While this cannot always be avoided, many conflicts are *false conflicts* at the level of application semantics. For example, consider a transaction T that adds a number of new records to a data structure organized as a B-tree. *Logically* speaking, if other transactions do not access these records or otherwise inquire directly or indirectly about their presence or absence in the data structure, then they do not conflict with T. However, straightforward mechanisms for guaranteeing safe transactional access to the B-tree might acquire locks on B-tree nodes and hold them until T commits. Thus, other transactions could be locked out of whole nodes of the tree, even though they are not (logically) affected by the changes T is making.

The solution discovered in the context of databases applies also to the case of parallel computing. It is to make a distinction between different *levels of semantics*, and requires recognizing certain data as being part of a coherent and distinct data abstraction. For example, in the case of a B-tree, each B-tree node is part of a given B-tree, and should be visible and manipulated only by actions on that B-tree. That is, the B-tree nodes are *encapsulated* within their owning B-tree. This allows B-tree actions to apply conflict management and undo or redo to B-tree nodes, during execution of those actions, and for the system to switch to *abstract* concurrency control and *abstract* undo or redo once a B-tree action is complete.

How does this solve the problem? The concurrency control and undo/redo on B-tree nodes allows safe concurrent (transactional) execution of B-tree actions themselves. This could also be achieved by non-transactional locking, lock-free or wait-free algorithms, or any other means that guarantees linearizability, but open nesting is generally taken to refer to the recursive use of transaction-like mechanisms, while wrapping a not necessarily transactional data type with abstract concurrency control and recovery is called *transactional boosting* [4]. More significantly, though, the conflicts between full B-tree *actions* will be much fewer than the (internal, temporary) conflicts on B-tree nodes during

those actions. For example, looking up record r_1 and adding record r_2 do not conflict logically, but if they lie in the same B-tree node, there will be a (physical) conflict on that node. While open nesting is by no means restricted to use with such collection data types, it is certainly very useful in allowing higher concurrency for them.

An Example Open Nesting Protocol

A fleshed-out example protocol for open nesting may be helpful to build understanding. This example uses in-place update and employs locks for early conflict detection, but other protocols are possible for other approaches to update and conflict detection. For understanding the protocol, a specific example data structure is instructive. Consider a `Set` abstraction implemented using a linked list. Suppose it supports actions `add(x)` and `remove(x)` to manipulate whether `x` is in the set, `size()` to return the set's cardinality, and `contains(x)` to test whether `x` is in the set.

Suppose the items `a`, `b`, and `c` have been added to the set in that order, and that `add` appends new elements to the end (since it must scan to the end anyway in order to avoid entering duplicates). Assume these elements are committed and there are no transactions pending against the set. Now suppose that a transaction adds a new member `d` and continues with other work. During the `add` operation, the transaction observes `a`, `b`, and `c` and the list links, then it creates a new list node containing `d` and modifies `c`'s link to refer to the new node. Suppose that another transaction concurrently queries whether the set contains `e`. At the physical level this `contains` query conflicts with the uncommitted `add`, because the query will read the link value in `c`'s list node, etc. Likewise a transaction that tries to remove `b` will conflict with both the `add` and the `contains` actions because it will try to modify the link in `a`'s node, to unchain `b`'s node from the list. To guarantee correct manipulation of the list each operation can acquire transactional locks on list nodes, acquiring an exclusive (X) mode lock when modifying a node and a share (S) mode lock when only observing it. Two S mode locks on the same object do not conflict, but all other mode combinations conflict. The pointer to the first node is likewise protected with the same kind of locks. This locking protocol works fine for closed nesting, but it is easy to see that it leads to many needless conflicts.

Open nesting requires identifying the *abstract* conflicts between operations. This example protocol uses *abstract locks*. These locks include an S and X mode lock for each possible element of the set, and an additional lock with modes Read (R) and Modify (M) for the cardinality of the set. Two R mode locks do not conflict, and two M mode locks also do not conflict, but R and M mode conflict with each other. Here is a table showing the abstract locks acquired by each action on a `Set`:

`add(x)`	X mode on `x`; M mode on cardinality
`remove(x)`	X mode on `x`; M mode on cardinality
`contains(x)`	S mode on `x`
`size()`	R mode on cardinality

This can be refined to acquire only an S mode lock on `x` if `add` or `remove` does not actually change the membership of the set, and in that case also not to acquire the M mode lock on the cardinality.

Assuming that each action on the set is run as an open nested transaction, then before an action completes it must acquire the specified abstract locks. If it cannot do that, it is in conflict and some transaction must be aborted. Once the action is complete *and* holds the abstract locks, the nested transaction commits and releases the lower-level locks on list nodes. The parent transaction will hold the abstract locks until it itself commits. Similar to closed nesting, if an uncommitted open nested transaction aborts, it can simply unwind back to where it started and try again. Often such cases arise because of temporary conflicts on the physical data structure. However, if the conflict is because of an abstract lock, then retry is not likely to help – either other transactions need to complete or abort to get out of this transaction's way, or this transaction needs to abort higher up in the nested transaction tree.

Abstract locks are just one way of implementing detection of abstract conflicts. In general what is required is an encoding of *abstract conflict predicates* into conflict checking code. These conflict predicates

indicate which actions on a data type conflict with other actions. Here is an example table for `Set`:

	`add(y)`	`remove(y)`	`contains(y)`	`size()`
`add(x)`	`x = y`	`x = y`	`x = y`	`true`
`remove(x)`	`x = y`	`x = y`	`x = y`	`true`
`contains(x)`	`x = y`	`x = y`	`false`	`false`
`size()`	`true`	`true`	`false`	`false`

In this table, the left action is considered to have been performed by one transaction, and the right action is requested by another. The entry in the table indicates the condition under which the new request conflicts with the older, not yet committed, action. The table above is expressed in terms of the operations and their arguments. However, it is possible to refine these predicates if they can refer to the *state* of the set. In general this might include the state after the first operation as well as the state before it. The refined table below uses references to the state `S` before the first operation:

	`add(y)`	`remove(y)`	`contains(y)`	`size()`
`add(x)`	$x = y \wedge x \notin S$	$x = y$	$x = y \wedge x \notin S$	$x \notin S$
`remove(x)`	$x = y$	$x = y \wedge x \in S$	$x = y \wedge x \in S$	$x \in S$
`contains(x)`	$x = y \wedge x \notin S$	$x = y \wedge x \in S$	`false`	`false`
`size()`	$y \notin S$	$y \in S$	`false`	`false`

Open nesting involves more than just conflict detection. Until an open nested action commits, aborting it works like aborting a closed nested action: Simply apply its (lower level) undos in reverse order and release its (lower level) locks. However, once an open nested action commits, it does not work to undo it using the list of lower level undos it accumulated while it ran. The lower level undos are guaranteed to work properly only if the lower level locks are still held. To undo a *committed* open nested action, the system applies an *abstract undo*, also called an *inverse* or *compensating action*. Here is a table of inverses for actions on the `Set` abstraction; a — entry means that no inverse is needed (the action did not change the state):

Action	`add(x)`	`remove(x)`	`contains(x)`	`size()`
Inverse	if $x \notin S$ then `remove(x)`	if $x \in S$ then `add(x)`	—	—

Notice that the appropriate inverse can depend on the state in which the original action ran. If `add` or `remove` does not actually change the state, then they can simply omit adding an inverse.

These inverses are added to the parent transaction's undo list, to apply if the parent transaction needs to abort. The abstract concurrency control will guarantee that these inverses still make sense when they are applied. They should be run as open nested transactions, and if they fail, it will only be because of temporary conflicts on the physical data structure, so they should simply be retried until they succeed.

Coarse-Grained Transactions

Transactions at the more abstract level, employing abstract concurrency control, and, if using in-place update, abstract undos, can be more generally termed *coarse-grained transactions*. As previously noted, the individual actions need not be run as transactions under a transaction mechanism – all that is required is that they are linearizable. However, using a transaction mechanism does offer the advantage of being able to abort and retry an action in case of conflict, while other approaches must guarantee absence of conflict. Thus, if the system does not use nested transactions to implement the actions, then, if using in-place update, it will need to acquire abstract locks *before* running the action. This implies that it cannot base the lock acquisition on the state of the data abstraction or on the result of the action. However, if executing the action reveals that the originally acquired abstract lock is stronger than necessary, the implementation can then downgrade the lock.

As noted before, the underlying implementation might use non-transactional locks to synchronize (for example, one mutual exclusion lock on the whole data structure will work, at the cost of reduced concurrency), or might use lock-free, obstruction-free, or wait-free techniques to obtain linearizability.

Upon first consideration, in-place updates may appear more complicated, since they require specifying, implementing, tracking, and applying undos. However, providing new copies of an abstract data structure has its own problems. One difficulty is being clear as to what needs to be copied. A second problem is cost. To reduce cost, a coarse-grained transaction implementation might use *Bloom filters*, which record

and examine an ongoing transaction's changes in a side data structure, private to the transaction. Transactions must also record additional information even for read-only actions, in order to check for conflicts later.

Correct Abstract Concurrency Control

The tables above gave conflict predicates without indicating how to derive or verify them. A conflict predicate is *safe* if the actions commute when the predicate is false. [Some make a distinction between actions moving to the left and to the right, but in most practical cases actions either commute (move both ways) or they do not (neither way).] Two actions commute if executing them in either order allows them to return the same results, and also does not affect the outcome of any future actions. Assuming that all relevant aspects of the state are observable, then this can be rephrased as: Actions commute if, when executed in either order, they produce the same results and the same final state.

There are some subtleties lurking in this definition. First, the "same state" means the same *abstract* state. For example, in the case of `Set` implemented as a linked list, the abstract state consists in what elements are in, and not in, the set. The order in which the members occur on the linked list does not matter. Therefore, even though `add(x)` and `add(y)` result in a different linked list when executed in the opposite order, *abstractly* there is no difference. Thus it is important to have clarity about what the abstract state *is*.

Second, interfaces vary in what they reveal. For example, `add(x)` might return nothing, not revealing whether `x` was previously in the set. As far as concurrency control goes, the less revealing interface reduces conflicts: two transactions could both do `add(x)` without conflict as perceived via this interface.

Third, if the system uses undos, then the undo added to a transaction's undo list is part of the result to consider when determining conflicts. So, if `x` is not initially in a set, and then two transactions each invoke `add(x)`, even if the `add` actions return no result, the actions conflict since the undo for the first one is `remove(x)` and the undo for the second is "do nothing." In this respect, late conflict detection sometimes allows more concurrency. However, in general it requires making a copy (at least an effective copy) of the data structure, and if any transaction commits changes to the data structure while transaction T is running, T's actions must be redone on the primary copy of the data structure rather than directly installing the new state that T constructed.

Fourth, certain non-mutating operations entail concurrency control obligations that may at first seem surprising. For example, if `x` is not in a set and transaction T runs the query `contains(x)`, the set must guarantee that any other transaction that adds `x` will conflict with T. Thus, if the system uses abstract locking, `contains(x)` must in this case lock the *absence* of `x`. Hence, an abstract lock is not necessarily a lock attached to some piece of the original data structure. (In some database implementations the locks are mixed with the actual records, and the system creates a new record for a lock like this, a record that goes away at the end of the transaction. This is called a *phantom record*.) A similar case occurs with an ordered set abstraction when a call to `getNextHigher(x)` returns `y`: The transaction must lock the fact that the ordered set has no value between `x` and `y`. Thus, read-only actions still require checking and recording, and this applies equally to late conflict detection as to early detection.

Extended Semantics

Another use for open nesting is to break out of strict serializability (at the programmer's risk, of course). It is sometimes useful, even necessary, to keep some effects of a transaction even if it is aborted. For example, in processing a commercial transaction, a system might discover that the credit card presented is on a list of stolen cards. While most effects of the purchase should be undone, information about the attempted use of the card should go to a log that will definitely *not* be undone. This is easy to do by giving the log action's inverse as "do nothing." (This is harder to do in a system that is not doing in-place updates, and would require a special notation.) In this way open nesting can be abused to achieve irrevocable effects. Similarly, a programmer can understate conflict predicates and allow communication between transactions. The "extended semantics" of open nesting abused in these ways may depend on the underlying implementation.

Nesting in Transactional Memory

Both closed and open nesting have been proposed for use with transactional memory (TM), for both software (STM) and hardware (HTM) approaches. The primary difficulty in implementing closed nesting for TM is its more complex conflict rule. It no longer suffices to check for equality or inequality of transaction identifiers – the test must distinguish an ancestor transaction from a non-ancestor. (This assumes that only transactions that are currently leaves of the transaction tree can execute.) HTM designs must also deal with the reality that hardware resources are always limited, and thus, there may be hard limits on the nesting depth, for example. HTM will also not be aware of abstract locks and abstract concurrency control; they will always be implemented in software. However, the number of conflict checks required for abstract concurrency control is strictly less than for physical units such as words or cache lines.

It is particularly more complex to check for conflict between concurrent subtransactions running under nested TM. However, if a transaction has at most one child at once, and only leaf transactions can execute, then the implementation is only slightly more complex than for non-nested transactions. Because the transaction tree in this case consists of a single line of descent from a top-level transaction, it is called *linear nesting*. Linear nesting admittedly forgoes one of the strong advantages of nesting, namely concurrent sibling subtransactions, but it retains partial rollback and thus remains potentially more useful than non-nested transactions.

Bibliographic Notes and Further Reading

The early exposition of nested transactions is marked by Davies [2], Reed [10], and Moss [5, 6]. Open nesting (also called *multi-level transactions*) was articulated by Beeri et al. [1], Moss et al. [8], and Weikum and Schek [11]. Nested transactions for hardware transactional memory are explored in Yen et al. [12] and Moss and Hosking [7], and Ni et al. [9] describe a prototype that supports open nesting in software transactional memory. Transactional boosting was introduced by Herlihy and Koskinen [4]. Gray and Reuter [3] provide comprehensive coverage of transaction processing concepts and techniques.

Bibliography

1. Beeri C, Bernstein PA, Goodman N (1983) A concurrency control theory for nested transactions. In: Proceedings of the ACM Symposium on Principles of Distributed Computing. ACM Press, New York, pp 45–62
2. Davies CT Jr (1973) Recovery semantics for a DB/DC system. In: ACM '73 Proceedings of the ACM Annual Conference. ACM, New York, pp 136–141. doi: http://doi.acm.org/10.1145/800192.805694
3. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Data Management Systems. Morgan Kaufmann, Los Altos, CA
4. Herlihy M, Koskinen E (2008) Transactional boosting: a methodology for highly-concurrent transactional objects. In: Chatterjee S, Scott ML (eds) Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, 20–23 Feb 2008. ACM, New York, pp 207–216, ISBN 978-1-59593-795-7
5. Moss JEB (1985) Nested transactions: an approach to reliable distributed computing. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (Also published as MIT Laboratory for Computer Science Technical Report 260)
6. Moss JEB (1985) Nested transactions: an approach to reliable distributed computing. MIT Press, Cambridge, MA
7. Moss JEB, Hosking AL (2006) Nested transactional memory: model and architecture sketches. Sci Comput Progr 63: 186–201
8. Moss JEB, Griffeth ND, Graham MH (1986) Abstraction in recovery management. In: Proceedings of the ACM Conference on Management of Data. Washington, DC. ACM SIGMOD, ACM Press, New York, pp 72–83
9. Ni Y, Menon V, Adl-Tabatabai AR, Hosking AL, Hudson RL, Moss JEB, Saha B, and Shpeisman T (2007) Open nesting in software transactional memory. In: ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming, San Jose, CA. ACM, New York, pp 68–78
10. Reed DP (1978) Naming and synchronization in a decentralized computer system. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (Also published as MIT Laboratory for Computer Science Technical Report 205)
11. Weikum G, Schek HJ (1992) Concepts and applications of multilevel transactions and open nested transactions. Morgan Kaufmann, Los Altos, CA, pp 515–553, ISBN 1-55860-214-3
12. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) LogTM-SE: decoupling hardware transactional memory from caches. In: Proceedings of the 13th International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10–14 February 2007, Phoenix, Arizona, USA, IEEE Computer Society, Washington, DC, pp 261–272

Transpose

►All-to-All

TStreams

►Concurrent Collections Programming Model

Tuning and Analysis Utilities

►TAU