Optimal Task Scheduling to Minimize Inter-Tile Latencies [*]

Fabrice Rastello
LIP, Ecole Normale
Supérieure de Lyon
46 allée d'Italie
69364 Lyon Cedex 07, France
fabrice.rastello@ens-lyon.fr

Amit Rao
PO Box 210030
Dept. of ECECS
University of Cincinnati,
Cincinnati, OH 45221
arao@ececs.uc.edu

Santosh Pande
PO Box 210030
Dept. of ECECS
University of Cincinnati,
Cincinnati, OH 45221
santosh@ececs.uc.edu

Abstract

This work addresses the issue of exploiting intra-tile parallelism by overlapping communication with computation removing the restriction of atomicity of tiles. The effectiveness of tiling is then critically dependent on the execution order of tasks within a tile. In this paper we present a theoretical framework based on equivalence classes that provides an optimal task ordering under assumptions of constant and different permutations of tasks in individual tiles. Our framework is able to handle constant but compile-time unknown dependences by generating optimal task permutations at run-time and results in significantly lower loop completion times. Our solution is an improvement over previous approaches [2, 6] and is optimal for all problem instances. We also propose efficient algorithms that provide the optimal solution. The framework has been implemented as an optimization pass in the SUIF compiler and has been tested on a distributed memory system using a message passing model. We show that the performance improvement over previous results is substantial.

1. Introduction

Loop tiling is one of the most popular techniques to partition uniform loop nests and map tasks to processors in a multicomputer. The notion of loop tiling was introduced by Wolfe [11] and the issue of legality of tiling was formalized by Irigoin and Triolet [7]. Usually tiles are considered to be atomic *i.e.*, interprocessor communication is considered to take place only after the end of computation in each tile. Atomic tile considerations are justified when the cache size is small since tile size is chosen to fit the underlying data in the cache. However, the assumption of atomic tiles leads to loss of potential parallelism when targeting parallel machines such as distributed shared memory systems (DSMs) that are capable of overlapping communication with computation. In fact, intra-tile optimizations, given that computation and communication can overlap, open opportunities for exploiting more parallelism.

Hollander [5] extracts parallelism in nested loops by identifying and labeling independent subsets of iterations using unimodular transformations. However, this work is not suited to DSMs as it involves complex data partitioning and assignment of iterations to processors. Tiling is more attractive in such situations where data and iteration space partitioning can be kept simple and efficient. Many approaches [1, 3, 8, 10] use the assumption of atomic tiles to derive the optimal tile size and shape. These approaches involve re-distribution of data among processors in order to improve data locality thereby addressing the issue of memory to tile latencies. On the other hand, we perform loop tiling and attempt to minimize inter-tile latencies by re-ordering individual tasks within a tile.

Previous approaches by Chou & Kung [2] and Dion et al [6] to the problem of finding an optimal task ordering within loop tiles have relied on heuristics that do not yield the optimal solution for all instances of the problem even in one dimension. Thus, finding the optimal solution in one dimension is still an open issue. The central contribution of this work is the development of a new formulation of this problem along with a framework based on equivalence classes to derive optimal permutations. Using this framework we also develop efficient algorithms that result in an optimal solution for all problem instances in one dimension. Further, we take into account the possibility of having different task permutations in each tile and give the optimal solution for this case too.

The remainder of the paper is organized as follows. Section 2 presents the problem statement. Section 3 introduces definitions and notations used in this paper. Section 4 discusses previous work. Section 5 formulates the problem based on equivalence classes. In section 6 we develop the theoretical framework and show optimality results. Section 7 presents the proposed constant permutation algorithm. Section 8 discusses the solution when different task permutations are considered in each tile. In section 9, we present and discuss the results of our implementation. Finally, section 10 provides concluding remarks.

2. Statement of the problem

Consider a loop of N iterations that carries a dependence of distance l. Assume that the loop is tiled with tile size n and mapped on to P processors. The N tasks are partitioned into

[*]This work is supported in part by the National Science Foundation through grant no. CCR-9696129; by the CNRS–ENS Lyon–INRIA project *ReMaP*; and by the Eureka Project *EuroTOPS*

$\lceil \frac{N}{n} \rceil$ tiles such that for $0 \leq i < \lceil \frac{N}{n} \rceil - 1$, the tasks contained in tile T_i are $\{t_{ni}, t_{ni+1}, ..., t_{n(i+1)-1}\}$ and $T_{\lceil \frac{N}{n} \rceil - 1} = \{t_{n(\lceil \frac{N}{n} \rceil - 1)}, ..., t_{N-1}\}$. Tile T_i is executed on processor P_i for every i. The problem is to find an optimal ordering (σ) of tasks in a tile, so as to minimize the overall completion time (T_{tot}) of the loop respecting the dependences imposed. Since tiling the loop and mapping the tiles are dictated by different considerations, the number of processors and tiles need not be the same.

We now present some of the definitions and notations used in the paper.

3. Terms and Definitions

Definition 1 *Let us denote $\tau = \{t_0, t_1, ..., t_{N-1}\}$ as the task space. Each element of τ is a task to be executed.*

Definition 2 *The iteration space is mapped onto the tile space. Each tile contains n tasks ; more precisely, a tile is a set $T_j = \{t_{nj}, t_{nj+1}, t_{nj+2}, ..., t_{nj+n-1}\} \cap \tau$ with $j \in \{0, ..., \lceil \frac{N}{n} \rceil - 1\}$. All the tasks of one tile are to be executed on one processor only.*

Let τ_{calc} be the time to perform one task with one processor, and let τ_{comm} be the time for one communication between two adjacent processors. The g.c.d of n and l is denoted by $n \wedge l$. P denotes the number of processors. k denotes n mod l while p denotes $\lfloor \frac{n}{l} \rfloor$. Finally,

$$x \equiv y[l] \iff x \bmod l = y \bmod l$$

Definition 3 *T is called the period of ordering such that for each i, tile T_i is executed from time iT to time $iT + n\tau_{calc}$.*

Since, the total loop completion time depends linearly on the value of T, the problem reduces to minimizing T.
In each tile, the permutation of tasks is modulo n. Let σ_0 denote the permutation of tasks in tile T_0.

$$\{0, 1, ..., n\} \overset{\sigma_0}{\to} \{0, 1, ..., n\}$$

Definition 4 *A constant task permutation is one in which for all $i \in \{0, 1, ..., N-1\}$, t_i is executed by the processor $j = \lfloor \frac{i}{n} \rfloor$ between time $\tau_i = jT + \sigma_0(i - nj)\tau_{calc}$ and the time $\tau_i + \tau_{calc}$.*

Definition 5 *A non-constant task permutation is one in which for all $i \in \{0, 1, ..., N-1\}$, t_i is executed by the processor $j = \lfloor \frac{i}{n} \rfloor$ between time $\tau_i = jT + \sigma_j(i - nj)\tau_{calc}$ and the time $\tau_i + \tau_{calc}$.*

Definition 6 *For a given n, and a given l, and τ_{calc} and τ_{comm} being fixed, the minimum period of ordering T_{min} is the smallest reachable value of T satisfying the above constraints with an appropriate permutation within a tile.*

4. Previous Work

The problem of determining optimal permutations of tasks within a tile to reduce inter-tile latencies and thereby minimize loop completion time has been attempted by Chou & Kung [2] and by Dion et. al. [6].

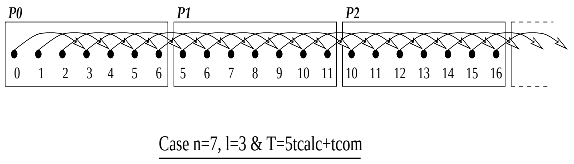


Figure 1. Chou & Kung's solution for the case $n = 7$ and $l = 3, T = 5$ (numbers below tasks represent times at which they are executed, considering $\tau_{calc} = 1$ & $\tau_{comm} = 0$).

4.1. Chou & Kung's Solution

Chou and Kung propose the following ordering of tasks on each tile (processor),
For all $i \in \{0, .., N-1\}$, $j = \lfloor \frac{i}{n} \rfloor$ (we have $nj \leq i \leq nj + n - 1$), t_i is executed by processor P_j between time $\tau_i = (i - nj) \times \tau_{calc} + j \times T$ and $\tau_i + \tau_{calc}$. The period of ordering is given by,

$$T = (n - l + 1) \times \tau_{calc} + \tau_{comm}$$

Hence, for all i, the processor P_i starts working at time $i \times T$ and ends working at time $i \times T + n \times \tau_{calc}$. Chou & Kung's solution for the case $n = 7$ and $l = 3$ is presented in Figure 1. As one can see their approach is non-optimal. Dion improved over Chou and Kung's solution by considering a better ordering of tasks within each tile so as to decrease the period (T) of ordering.

4.2. Dion's solution

First Dion et al. show the following results,

Lemma 1 *The best local permutation that permits reaching the minimum period T_{min} is independent of the values of τ_{calc} and τ_{comm}. Moreover, if $T^{1,0}_{min}$ is the minimum period for $\tau_{calc} = 1$ and $\tau_{comm} = 0$ then $T^{\tau_{calc}, \tau_{comm}}_{min} = T^{1,0}_{min} \times \tau_{calc} + \tau_{comm}$.*

Lemma 2 *If $n \wedge l \neq 1$, the problem is equivalent to a smaller problem with $n' = \frac{n}{n \wedge l}$ and $l' = \frac{l}{n \wedge l}$.*

Hence, from now on we can choose n and l such that $n \wedge l = 1$, τ_{calc} will be equal to 1, and τ_{comm} will be 0, so that the discussion is simplified.

Dion et al's permutation leads to a smaller period of ordering than Chou & Kung's solution. In fact, their solution is optimal for the special case $l = 2$. For example, for the case $n = 9$ and $l = 2$ Dion et al's solution leads to $T = 7$ which is the optimal solution, while Chou and Kung's solution leads to $T = 8$. Larger the value of n, greater will be the difference between the two solutions and thus, the loop completion times. The solutions are presented in Figure 2.

The following theorem summarizes Dion et al's contribution for the special case $l = 2$,

Theorem 1 *For $n = 2k + 1$, $k > 0$ and $l = 2$, the optimal ordering has a period of ordering,*

$$T_{opt} = \lceil \frac{3n - 1}{4} \rceil$$

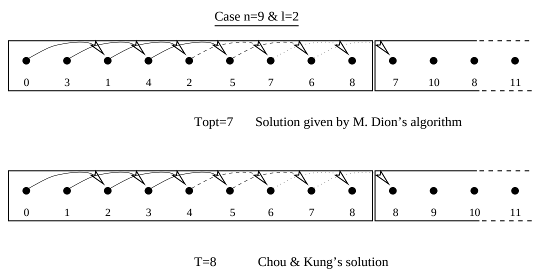

Case n=9 & l=2
0 3 1 4 2 5 7 6 8 7 10 8 11
Topt=7 Solution given by M. Dion's algorithm
0 1 2 3 4 5 6 7 8 8 9 10 11
T=8 Chou & Kung's solution

Figure 2. Comparison of Dion et al's solution and Chou & Kung's solution for $n = 9$ and $l = 2$.

For $l \geq 3$, Dion et al. give an algorithm called a *cyclic* algorithm that gives a correct permutation or schedule with $T = 2\lfloor \frac{n}{l} \rfloor + 2$. However, this is not an optimal solution. This is one of the major limitations of their solution. The other limitation is that they have used the simplifying assumption of a constant permutation in every processor similar to Chou and Kung's work. In later sections we show that the solution can be greatly improved if we remove this restriction.

However, determining the optimal solution for both constant and non-constant permutations is a non-trivial combinatorial problem. We propose a suitable framework that determines optimal task permutations in $O(n)$. Our framework derives the optimal period of ordering T_{min}, for all cases of l, in terms of n, k and l, where $n = lp + k$ & $0 \leq k < l$. The framework also leads to efficient algorithms that always reach the optimal period of ordering (T_{min}).

5. Formulation of the Problem using Equivalence Classes

We introduce the following notations and definitions for our subsequent discussion,

1. The set of tasks of the first tile is given by $\Omega = \{t_0, t_1, ..., t_{n-1}\}$. Further, we say that $t_i \equiv t_j$ iff $i \equiv j[l]$. Hence, by using a simplified notation, we denote task t_i by the integer i.
2. The operator $\equiv$ is an equivalence relation that defines equivalence classes within $\{0, 1, ..., n-1\}$. This equivalence relation defines l components or equivalence classes, $X_0, X_1, ..., X_{l-1}$. We denote the set of equivalence classes by $\Psi = \{X_0, X_1, ..., X_{l-1}\}$.
3. We say that, $X \rightarrow Y$ if,
$$\exists (x, y) \in (X, Y) \cdot x \equiv y + n[l]$$
4. The indices for the equivalence classes are chosen in a manner such that,
   - $X_0 = \{i \in \Omega \cdot i \equiv 0[l]\}$
   - $X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_{l-1} \rightarrow X_0$

From the following definitions we have,

$$\forall X_i \in \Psi, p \leq |X_i| \leq p + 1$$

Moreover, there are k classes of size $(p+1)$ & $(l-k)$ classes of size p.

Definition 7 *For all i in $\{0, 1, \cdots, l-1\}$, we define $\lambda_i = |X_i| - p$. In other words, if $|X_i| = p + 1$ then $\lambda_i = 1$, else $|X_i| = p$ and $\lambda_i = 0$.*

Definition 8 *For all $i \geq l$, $\lambda_i = \lambda_{i \bmod l}$*

Therefore, the $\lambda-$string can be extended to an infinite string (periodic with period l).
Example 2 : Consider $n = 7$ and $l = 3$. We have,

$$X_0 = \{0, 3, 6\}, \lambda_0 = 1$$
$$X_1 = \{2, 5\}, \lambda_1 = 0$$
$$X_2 = \{1, 4\}, \lambda_2 = 0$$

The task permutation and therefore the period of ordering depends on the relative values of λ_i *i.e.,* the property of the string $\lambda_0\lambda_1\lambda_2 \cdots \lambda_{l-1} \cdots$. We derive the optimal solution in the next section.

6. Optimal Solution

As shown in last section, the size of an equivalence class X_i is $p + \lambda_i$, with $p = \lfloor \frac{n}{l} \rfloor$. We claim that the optimum period of ordering depends on the relative values of λ_i, more specifically λ_i and λ_{i+1}. We, thus, first quantify the property of the string in theorem 2 and then use it to determine the optimum period of ordering in theorem 3. Specifically, theorem 2 is analyzes the property of the string $\lambda_0, \lambda_1, ..., \lambda_{l-1}$..... in terms of the tile size (n) and the dependence distance (l). This relationship is quantified in terms of Λ presented below.

Theorem 2 *If $l \geq 3, n \wedge l = 1$ and $\Lambda = \min_{i \in \mathbb{N}}[\max_{i+1 \leq j, j+1 \leq i+l-1}(\lambda_j + \lambda_{j+1})]$, then*

- *If $l = 4$ & $k = 3$ then*
$$\Lambda = \max_{2 \leq j, j+1 \leq 4}(\lambda_j + \lambda_{j+1}) = 1$$
- *Else*
$$\Lambda = \max_{1 \leq j, j+1 \leq l-1}(\lambda_j + \lambda_{j+1}) = \left| \begin{array}{ll} 0 & \text{if } k = 1 \\ 1 & \text{if } 1 < k \leq \lceil \frac{l}{2} \rceil \\ 2 & \text{if } k > \lceil \frac{l}{2} \rceil \end{array} \right.$$

$\square$

As seen above, the definition of Λ is a min-max definition. The above theorem gives the values of Λ for all cases of tile size and dependence distance.

We now illustrate the result of theorem 2 through some examples.

Example 3 : Consider $n = 7$, $l = 3$ ($k = 1$)
$\lambda -$ string $= 1\mathbf{00}100100...$
$\Lambda = 0$.
Example 4 : Consider $n = 7$, $l = 4$ ($k = 3$)
$\lambda -$ string $= 11\mathbf{101}110...$
$\Lambda = 1$

Example 5 : Consider $n = 7,\ l = 5\ (k = 2 < \lceil\frac{5}{2}\rceil)$
$\lambda -$ string $=$ 1**0100**10100...
$\Lambda = 1$
Example 6 : Consider $n = 9,\ l = 5\ (k = 4 > \lceil\frac{5}{2}\rceil)$
$\lambda -$ string $=$ 1**1110**11110...
$\Lambda = 2$

The Λ values calculated above for different tile sizes and dependence distances allow us to determine the optimal period of ordering as per theorem 3. In order to prove theorem 2 we need a few results.

Lemma 3 gives us a working definition for X_i.

Lemma 3 *If* $X'_i = \{x \in \Omega \mid \exists\alpha \in I\!N \cdot (x \equiv \alpha l[n]) \wedge (in \leq \alpha l < (i+1)n)\}$, *then* $\forall i,\ X'_i = X_i$.

Proof Refer [9].

The next lemma states the condition that the tile size and the dependence distance should satisfy in order to have a sub-string 11 within the λ-string.

Lemma 4

$$\begin{aligned} \exists i \in \{1, ..., l-1\} \mid \lambda_i\lambda_{i+1} = 11 \quad &\Longleftrightarrow \quad 2k > l+1 \\ &\Longleftrightarrow \quad k > \lceil\frac{l}{2}\rceil \end{aligned}$$

Proof Refer [9].

In order to prove theorem 2, we have to differentiate between the cases $\Lambda = 1$ and $\Lambda = 2$ (the case $\Lambda = 0$ is trivial). To achieve this task we need to discuss whether there exists i in $\{2, \cdots, l-1\}$ such that $\lambda_i\lambda_{i+1} = 11$ or not. Indeed, we will see that if $k > \lceil\frac{l}{2}\rceil$ then $\lambda_0\lambda_1 = 11$ and that $\lambda_{l-1} = 0$.

To formalize it, we need to introduce a new concept - the property of a string to be well balanced.

Let $\sum = \{0, 1\}$ be the alphabet of the λ-string.

Definition 9 (sub-string) *v is said to be a sub-string of u, if there exist two strings α & β, such that $u = \alpha v\beta$.*

Definition 10 (length) *The length of a string $u = u_1u_2u_3....u_n$ is the integer n denoted by $|u|$.*

Definition 11 (weight) *Let $\alpha \in \sum$. If $u = u_1u_2.....u_n \in \sum^*$ is a string of length n, then $|u|_\alpha = |\{i \in \{1, ..., n\}, u_i = \alpha\}|$.*

Definition 12 (well-balanced) *Let $u \in \sum^*$. u is said to be well-balanced if for any pair of sub-strings of u, (v, v'),*

$$|v| = |v'| \Longrightarrow ||v|_1 - |v'|_1| \leq 1$$

Lemma 5 *The infinite string $\lambda = \lambda_0\lambda_1\lambda_2.....$ is well-balanced.*

Proof Refer [9].
Now we can easily prove *theorem 2,*
Theorem 2 proof: First, note that $\lambda_0 = 1$. We have,
$|X_0| = |X'_0| = |\{\alpha \in I\!N, 0 \leq \alpha l < n\}| = p + 1$.
Also $\lambda_{l-1} = 0$.

$$\begin{aligned} |X_{l-1}| &= |\{\alpha \in I\!N, (l-1)n \leq \alpha l < ln\}| \\ &= |\{\alpha \in I\!N, (l-1)n \leq \alpha l \leq ln\}| - 1 \\ &\leq (p+1) - 1 \end{aligned}$$

1. <u>Case $k = 1$</u>:
   - As $k = |\lambda_0\lambda_1....\lambda_{l-1}|_1 = 1$
     & $\lambda_0 = 1$ therefore,
     $\Lambda \leq \max_{1\leq p,p+1\leq l-1}(\lambda_p + \lambda_{p+1}) = 0$
   - Consequently, $\Lambda = 0$.

2. <u>Case $1 < k \leq \lceil\frac{l}{2}\rceil$</u>:
   - $\forall i, |\lambda_i\lambda_{i+1}....\lambda_{i+l-1}| = k \geq 2$, therefore,
     $\forall i, \exists p \in \{i, ..., l+i-2\} \mid \lambda_p = 1$
     Hence, $\Lambda \geq 1$.
   - From the proof of *lemma 4*, we have

     $$\Lambda \leq \min_{1\leq p,p+1\leq l-1} \lambda_p + \lambda_{p+1} \leq 1$$

   - Consequently, $\Lambda = 1$.

3. <u>$l > k > \lceil\frac{l}{2}\rceil$</u>:
   - We have $2n = 2pl + 2k \geq (2p+1)l + 1$. Therefore,

     $$\begin{aligned} |X'_0| + |X'_1| &= |\{\alpha \in I\!N, 0 \leq \alpha l < 2n\}| \\ &= 2p + 2 \end{aligned}$$

     Hence, $\lambda_0\lambda_1 = 11$
   - Suppose that *there exists i in* $\{2, 3, ..., l-1\}$ such that $\lambda_i\lambda_{i+1} = 11$. Since $i \neq l-1$ this will lead to $\Lambda = 2$
   - Suppose that *there does not exist* $i \in \{2, 3, ..., l-1\}$ such that $\lambda_i\lambda_{i+1} = 11$.
     Then $\lambda_{l-1}\lambda_0\lambda_1\lambda_2 = 0111$ ($\lambda_2 = 0$ violates $k > \lceil\frac{l}{2}\rceil$).
     Hence, $l \geq 4$. Let $l > 4$.
     From *lemma 5* we have, $\lambda_0\lambda_1.....\lambda_{l-1}$ does not contain the substring 00.
     So, l is necessarily even $(l \geq 6)$

     $$\begin{aligned} \lambda_0\lambda_1....\lambda_{l-1} &= 111(01)^{\frac{l-4}{2}}0 \\ &= 111010(10)^{\frac{l-6}{2}} \end{aligned}$$

     But this violates *lemma 5*, because of the sub-strings 111 & 010.
     Finally, $l = 4$ & $k = 3$ gives
     $\lambda_0\lambda_1\lambda_2\lambda_3\lambda_4\lambda_5\lambda_6\lambda_7 = 11101110$,
     and $\Lambda = \max_{2\leq p,p+1\leq 4} \lambda_p + \lambda_{p+1} = 1$

□

We now state the theorem that determines the lower bound on the period of ordering in case of constant task permutations in tiles.

Theorem 3 *For constant task permutations in tiles and $l > 2$, $T \geq 2p + \Lambda$.* □

Proof Refer [9].

7. Constant Permutation Algorithm

7.1. Case $n \wedge l = 1$

Using the framework developed in the previous section we are now able to devise an algorithm that will compute the optimal ordering of tasks under the assumption that the ordering is the same in every tile. *Algorithm 1* gives us a correct permutation of tasks with period of ordering $(T) = 2p + \Lambda$. Recall that $l \geq 3$ and $n \wedge l = 1$.

Algorithm 1

```
procedure ComputeOrderingCst ($n$, $l$)
Input:      $n$ (tile size)
            $l$ (dependence distance)
Precondition: $l \geq 3$
              $n \wedge l = 1$
Output: $permutation[0 : n-1]$ (task ordering)
{ Initialize variables $p$ and $k$ }
  $p := \lfloor \frac{n}{l} \rfloor$
  $k := n \bmod l$
{ Assign the first task $f$ to be executed }
  if ($l = 4$ and $k = 3$) then
    $f := l - k$ { because $0 = (f + n)[l]$ }
  else $f := 0$
{ Execute the first $p$ tasks of $X_f$ }
  $t := f$
  for $i := 0$ to $p - 1$ do
     $permutation[i] := t$
     $t := t + l$
  endfor
{ Equivalence classes are executed in the
  opposite order to ($\longrightarrow$) }
  $t := (t + n) \bmod l$
  while $t \neq f$
       repeat
          $permutation[i] := t$
          $t := t + l$
          $i := i + 1$
       until $t \geq n$
       $t := (t + n) \bmod l$
  endwhile
{ Execute the last task of $X_f$ }
  $permutation[i] := f + p \times l$
end ComputeOrderingCst
```

The permutation given by this algorithm clearly obeys the constraints of dependences in the tile. Algorithm 1 has time complexity $O(n)$. Please refer to [9] for the proof that the permutation given by algorithm 1 gives the minimum period of ordering, $T_{min} = 2p + \Lambda$.

7.2. General Case

We illustrate how our approach finds an optimal task permutation when we have a single constant dependence vector (l) and $n \wedge l = d \geq 1$.

Case $n = 14$ & $l = 6$															
Tile 0	Tasks	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Times	0	7	2	9	4	11	1	8	3	10	5	12	6	13
Tile 1	Tasks	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	Times	4	11	6	13	8	15	5	12	7	14	9	16	10	17
Tile 2	Tasks	28	29	30	31	32	33	34	35	36	37	38	39	40	41
	Times	8	15	10	17	12	19	9	16	11	18	13	20	14	21

Table 1. Optimal ordering with constant permutation in each tile ($n = 14, l = 6$).

Let us consider a correct ordering with T as the period. Consider the sub-set of tasks $(t'_i)_{i \in [0,n-1]}$ such that $\forall i, t'_i = t_{id}$. Let us call the tile consisting of the tasks (t'_i) a derived tile. The length of the derived tile is $n' = \frac{n}{d}$. The derived dependence distance is $l' = \frac{l}{d}$. We assume that the derived permutations in each tile are constant (constant permutation).

Hence, according to the last part $T \geq T_{min}(n', l')$.

Now, let us show that $T_{min}(n, l) = 2\lfloor \frac{n'}{l'} \rfloor + \Lambda(n', l')$

Let $\sigma' : \{0, \cdots, n'-1\} \rightarrow \{0, \cdots, n'-1\}$ be the permutation of tasks that gives the period T'_{min}.
Then, consider $\forall v \in \{0, \cdots, d-1\}, \forall i \in \{0, \cdots, n'-1\}$ $\sigma(v + id) = v + \sigma'(i)d$
Clearly, this permutation permits to reach the period $T = T'_{min}$. Thus, algorithm 1 generates the optimal permutation for this case also simply by using n' and l' as inputs instead of n and l.

The task execution times given by our algorithm for $n = 14$ and $l = 6$ is shown in Table 1. Since $n \wedge l = 2$, tile 0 has two components *viz.*, {0,2,4,6,8,10,12} and {1,3,5,7,9,11,13}. Each of them form derived tiles with $n' = 7$ and $l' = 3$. The first component has equivalence classes $X_0 = \{0, 6, 12\}, X_1 = \{4, 10\}, X_2 = \{2, 8\}$. First, p tasks of X_0 are executed followed by all the tasks of X_2 and X_1 in that order. Finally the last task of X_0 is executed. The second component is then executed with a similar task ordering.

The period of ordering reached by our algorithm is 4 while that reached by Dion's algorithm is 6.

8. Optimal Solution with Non-constant Permutations

We can further optimize the solution if we relax the constraint of maintaining a constant permutation in every tile. We also show that computing the optimal permutation in every tile does not result in any overhead because the optimal permutation in each tile is a simple shift of the permutation in the previous tile. By removing the constraint, we can reach the optimal period of ordering which is half smaller than that in the case of constant permutation.

Let $n \wedge l = d \geq 1$. Recall that from section 4, we have $X_0 \rightarrow X_1 \rightarrow \cdots \rightarrow X_{l-1} \rightarrow X_0$. The algorithm computes task ordering such that it leads to the following execution order. The first processor executes tasks belonging to X_0 then X_1, X_2 and finally X_{l-1}. The second processor starts working $|X_0|\tau_{calc} + \tau_{comm}$ units of time after the first one, and executes the tasks of X_1 then $X_2, X_3 \cdots X_{l-1}$ and finally X_0. The third processor starts working $|X_1|\tau_{calc} + \tau_{comm}$ units of time after the second one, and executes the tasks of X_2 then $X_3, X_4 \cdots X_0$ and finally

X_1. Clearly, we can see that all dependences are satisfied. This leads to the following algorithm.

Algorithm 2

```
procedure ComputeOrderingNoncstGen ($n,\ l,\ i$)
Input:    $n$ (tile size)
          $l$ (dependence distance)
          $i$ (tile number)
Output: $permutation[0 : n-1]$ (task ordering)
{ Initialize variables $d, n_d$ and $l_d$ }
  $d$ := gcd($n, l$)
  $n_d := \frac{n}{d}$
  $l_d := \frac{l}{d}$
{ Initialize variables $m$ and $f$ }
  $m := \lceil \frac{n_d i}{l_d} \rceil$
  $f := (m l_d)$ mod $n_d$
  for $i_d := 0$ to $d-1$ do
{ Assign the first task $f$ to be executed }
      $x := 0$
      $permutation[x] := fd + i_d$
{ Execute each equivalence class in the
  order of $(\longrightarrow)$}
      $j := (f + l_d)$ mod $n_d$
      while $j \neq f$ do
          $x := x + 1$
          $permutation[x] := jd + i_d$
          $j := (j + l_d)$ mod $n_d$
     endwhile
  endfor
end ComputeOrderingNoncstGen
```

In the above algorithm we have,

- i is the ordinal of the tile being executed.
- i_d is the ordinal of the derived tile being executed.
- m is the smallest integer where $n_d i \leq m l_d < n_d(i+1)$.
- f is the first task of the derived tile i_d to be executed.

The time offset O_i for each tile is the number of time units between the start of execution of tile i and tile $i+1$. The offset O_i as opposed to the period of ordering need not be the same for every tile. The offset O_i (in computational-time units) corresponds to the size of the following set,

$$X_i = \{m \in N, in \leq ml < (i+1)n\}$$

Hence,

$$O_i = \lceil \frac{n(i+1) - \lceil \frac{ni}{l} \rceil l}{l} \rceil \tau_{calc} + \tau_{comm}$$

Consider the example $n = 5$ and $l = 3$. We have the following times when tasks are executed.

$$\begin{array}{ccccc|ccccc|ccccc}
0 & 2 & 4 & 1 & 3 & 5 & 2 & 4 & 6 & 3 & 5 & 7 & 4 & 6 & 8\cdots \\
\bullet & \circ & \times & \bullet & \circ & \times & \bullet & \circ & \times & \bullet & \circ & \times & \bullet & \circ & \times\cdots
\end{array}$$

Table 2[1] illustrates that the ordering in each tile is a circular shift of the ordering in the previous tile.

It is easy to show that the period of ordering reached by *algorithm 2* is optimal. Consider two tiles T_j and T_{j+1}. Suppose that equivalence class X_i starts executing on T_j at $t = t_0$. Since $X_i \rightarrow X_{i+1}$ the earliest time at which X_{i+1} can start execution on T_{j+1} is $t = t_0 + |X_i|$. This is precisely the time offset between two consecutive tiles obtained through *algorithm 2*. Thus, latency between two tiles using non constant permutations is $\approx p$.

[1]The subscripts of tasks are modulo n.

Case $n = 5$ & $l = 3$	
Tile 0	Order 0 3 1 4 2
	Offset 2
Tile 1	Order 1 4 2 0 3
	Offset 2
Tile 2	Order 2 0 3 1 4
	Offset 1

Table 2. Optimal ordering with non-constant permutation in each tile ($n = 5, l = 3$).

9 Results

9.1 Performance Evaluation

The performance evaluation of the proposed methods was carried out using several signal processing applications consisting of matrix transformations. We tested our proposed algorithms using a sample test routine shown below,

```
Do i: 0 -> N
   Task(i)
EndDo
```

In the above loop, `Task()` exhibits a compile-time unknown dependence distance l in the outermost loop. In general, `Task()` can represent a loop nest or a function call or a group of statements. We tiled the above loop using the tiling transformation provided by the *SUIF* compiler. Since the dependence distance is a compile-time unknown in the above loop, the code must be generated which computes the task permutation at run time. In order to enforce synchronization between tiles imposed by the dependence relation, data is passed between left and right processing elements (PEs) using message passing library (MPI) calls. The complete framework has been incorporated in the *SUIF* compiler as an optimization pass.

The code generated by the compiler's optimization pass is sketched in figure 3. In case of constant permutations (figure 3(a)), permutations are generated at the entry point of the tile loop (`i_tile`). In case of non-constant permutations (figure 3(b)), permutations are generated at the entry point of the element loop (`i`). The element loop is executed following the owner computes rule. At run time `permute_tile` will generate an optimal task permutation in each tile using the appropriate algorithm (1 or 2). This permutation is saved in the array `perm` which is used to order the task execution. The final transformed code was targeted on Cray T3E.

We performed experiments by varying the tile size (n) and the dependence distance (l). The metric used to evaluate the performance of our algorithms in comparison to previous algorithms was

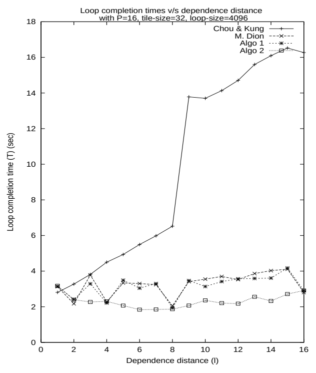
```
permute_tile(n,l); <generates pemutation
                   using algo 1>
Do i_tile: 0 -> N by n
  if i_tile maps onto PE then
    Do i: i_tile -> min(N,i_tile+n-1)
  <If needed, receive data from left PE>
      Task(perm[i]);
  <If needed, send data to right PE>
    EndDo
  endif
EndDo
```

(a)

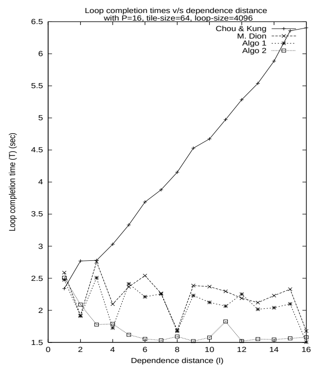
```
Do i_tile: 0 -> N by n
  if i_tile maps onto PE then
    permute_tile(n,l,i_tile); <generates
                  permutation using algo 2>
    Do i: i_tile -> min(N,i_tile+n-1)
   <If needed, receive data from left PE>
       Task(perm[i]);
   <If needed, send data to right PE>
    EndDo
  endif
EndDo
```

(b)

Figure 3. (a) Code generating constant permutations (b) Code generating non-constant permutations.



(a)



(b)

Figure 4. Loop completion time v/s dependence distance on Cray T3E for (a) P = 16, n = 32 and N = 4096 and (b) P = 16, n = 64 and N = 4096.

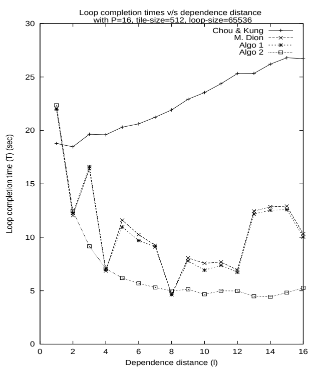
the total loop completion time. The tests were carried out using 16 processing elements with a cyclic distribution of tiles on processors. Figure 4 compares the performance of the final transformed code using the proposed algorithms (Algorithm 1 and 2 presented in sections 7 and 8) in comparison to previous approaches on the Cray T3E for small tile sizes. Figure 5 presents results obtained on the Cray T3E for larger tile sizes.

As seen in Figures 4 and 5 the performance obtained by using the two proposed algorithms is superior to that obtained by using algorithms proposed by Dion and Chou & Kung. This is true in the case of small tile sizes as well as large tile sizes.

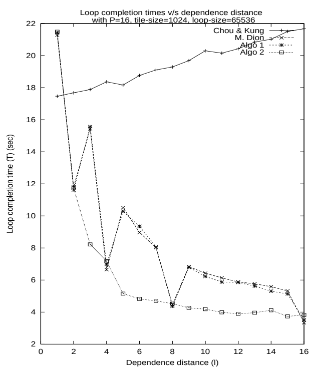
Figures 4 and 5 show that when $gcd(n, l)$ is l, Dion's algorithm and the proposed algorithms yield very similar results. This is because in this case l' is 1 causing all three algorithms to generate identical permutations.

Since task permutations are generated at run-time we need to investigate the execution efficiency of the proposed algorithms in comparison to previous approaches. Table 3 presents the times taken by the code that generates task permutations using our and Dion's algorithms indicating the following hierarchy in the *time complexity* of the permutation generation algorithms.
Dion's algorithm $<$ Algorithm 1 $<$ Algorithm 2

One can see that although Dion's algorithm is more time efficient, better loop completion times result from algorithm 1 & 2 due



(a)



(b)

Figure 5. Loop completion time v/s dependence distance on Cray T3E for (a) P = 16, n = 512 and N = 65536 and (b) P = 16, n = 1024 and N = 65536.

l	Time (micro-sec)		
	Dion	Algo 1	Algo 2
3	11937	17153	19451
5	13108	18352	19546
7	13923	18422	19711
9	14966	18783	19627

Table 3. Permutation generation times $(N = 256, n = 64, P = 4)$.

to superior task permutations. Thus, it is clear that the overhead of generating more complex task permutations does not nullify the performance gain achieved by those permutations. The results indicate the following *performance hierarchy* of the algorithms proposed in this paper. For $gcd(n, l) \neq l$,
Algorithm 2 $>$ Algorithm 1 $>$ Dion's algorithm

Algorithm 2 is also the most natural and efficient algorithm. For all tile sizes the results indicate that Algorithm 2 yields the best solution which is superior to solutions obtained by all constant permutation algorithms.

9.2 Effect of tile size

An interesting issue is to study the effect of variation of tile size on performance. In case of the multi-dimensional tiling problem $n_1 \times n_2 \times \cdots \times n_d$, the gain of our method using Algorithm 1 over Dion's algorithm will be proportional to $\prod_{i=1}^{d-1} n_i$.

Desprez et al [4] have addressed the issue of finding the optimal grain size that minimizes the execution time by improving pipeline communications on parallel computers. The following discussion presents the effect of tile size (n) on the total loop completion time (T_{tot}) in light of the framework developed in this paper.

Using Algorithm 2, we obtain the following period of ordering,

$$T \approx \frac{n}{l}\tau_{calc} + \tau_{comm}$$

The total loop completion time is given by,

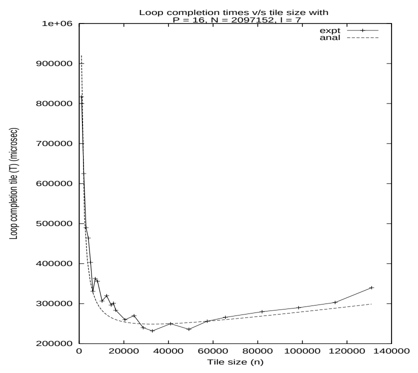


Figure 6. Loop completion time v/s tile size on SGI Power Challenge for P = 16, l = 7 and N = 2097152.

$$\begin{aligned} T_{tot} &\approx \frac{N}{n}T + n\tau_{calc} \\ &= \frac{N\tau_{calc}}{l} + n\tau_{calc} + \frac{N\tau_{comm}}{n} \\ &= A + Bn + \frac{C}{n} \end{aligned}$$

Minimizing the above expression we get,

$$n_{opt} = \sqrt{\frac{C}{B}} = \sqrt{\frac{N\tau_{comm}}{\tau_{calc}}}$$

Let $\frac{\tau_{comm}}{\tau_{calc}} = c$. Therefore,

$$n_{opt} \approx \sqrt{cN}$$

Also we have,

$$\frac{n}{l}\tau_{calc} > \tau_{comm}$$

This leads to,

$$n > lc$$

In order to compare the above analytical solution with experimental results we observed the total loop completion time (T_{tot}) varying the tile size (n) keeping P fixed. Figure 6 presents this comparison for $N = 2097152$ and $l = 7$. Figure 6 shows that the analytical expression derived for the loop completion time closely matches the experimental results. The knee of the analytical solution curve corresponds to the optimum tile size that yields the minimum loop completion time.

10. Conclusions

The effectiveness of loop tiling is critically dependent on the execution order of tasks within a tile. In this work, we have addressed the problem of finding an optimal ordering of tasks within tiles executed on multicomputers for constant but compile-time unknown dependences. We remove the restriction of atomicity on tiles and exploit the internal parallelism within each tile by overlapping computation with communication. We have formulated the problem and developed a new framework based on equivalence classes and show optimality results for single dimensional tiles with single constant dependences. Using the framework we have also developed two efficient algorithms that provide the optimal solution in both cases,

1. Same (constant) task ordering in tiles,
2. Different (non-constant) task ordering in tiles.

We have shown that the two proposed algorithms yield superior results to the previous approaches when tested on distributed memory systems. We also show that the non-constant permutations in our approach significantly reduce the loop completion time unlike the constant permutations in previous approaches. Finally, we have investigated the relationship between tile size and the loop completion time and developed a methodology to obtain optimal tile size given our framework.

References

[1] A. Agarwal, D. Kranz, and V. Natrajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, Sept. 1995.

[2] W. Chou and S. Kung. Scheduling Partitioned Algorithms on Processor Arrays with Limited Communication Supports. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP)*, pages 53–64, 1993.

[3] F. Desprez et al. Scheduling Block-Cyclic Array Redistribution. In *Parallel Computing '97 (ParCo97)*. North-Holland, September 1997.

[4] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 165–172. Springer Verlag, August 1996.

[5] E. D'Hollander. Partitioning and Labeling of Loops by Unimodular Transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, July 1992.

[6] M. Dion, T. Risset, and Y. Robert. Resource-constrained Scheduling of Partitioned Algorithms on Processor Arrays. In *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*, pages 571–580, 1995.

[7] F. Irigoin and R. Triolet. Supernode Partitioning. In *15th Symposium on Principles of Programming Languages (POPL XV)*, pages 319–329, 1988.

[8] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[9] F. Rastello, A. Rao, and S. Pande. Optimal Task Ordering in Linear Tiles for Minimizing Loop Completion Time. Technical Report TR 213/02/98/ECECS, University of Cincinnati, Jan. 1998.

[10] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[11] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, pages 357–361, 1987.