USING MESSAGE PASSING FOR DEVELOPING COARSE-GRAINED APPLICATIONS IN OPENMP

Bielecki Wlodzimierz and Palkowski Marek
Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st., 71-210 Szczecin, Poland

Keywords: Coarse-grained parallelism, send-receive mechanism, synchronization, agglomeration, locks.

Abstract: A technique for extracting coarse-grained parallelism in loops is presented. It is based on splitting a set of dependence relations into two sets. The first one is to be used for generating code scanning slices while the second one permits us to insert send and receive functions to synchronize the slices execution. Codes of send and receive functions based on both OpenMP and POSIX locks functions are presented. A way of proper inserting and executing send and receive functions is demonstrated. Using agglomeration and free-scheduling are discussed for the purpose of improving program performance. Results of experiments are presented.

1 INTRODUCTION

Extracting coarse-grained parallelism available in loops is of great importance to develop shared memory parallel programs. To develop such programs, the **OpenMP** standard can be used (OpenMP, 2007). OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. The API supports C/C++ and Fortran on multiple architectures, including UNIX & Windows NT. It permits for specifying parallel regions, work sharing, both barrier and mutual exclusion synchronization. However OpenMP does not support directly message passing (send and receive functions) among threads.

In this paper, we show the need for massage passing while we develop an OpenMP coarse-grained parallel program. Then we present how such a mechanism can be implemented on the basis of both OpenMP and POSIX locks. Agglomeration and free-scheduling are discuss to improve program performance. Results of experiments are presented.

In this paper, we deal with *affine loop nests* (Darte 2000). For extracting coarse-grained parallelism, we use an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a dependence if and only if it actually exists. To implement our algorithms, we use the dependence analysis proposed by Pugh and Wonnacott (Pugh 1998).

2 MOTIVATING EXAMPLE

Let us consider the following loop.

```
for( i=1; i<=n; i++)
 for( j=1; j<=n; j++)
  a(i,j) = a(i,j-1) + a(i-1,1);
```

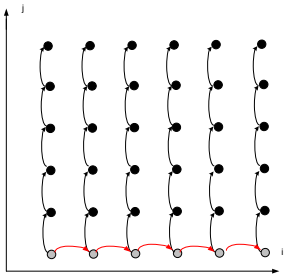

Figure 1: Dependences in the motivating example.

Using the Petit tool (Kelly 1997), we can extract the following dependence relations for this loop:

R1 = {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n},
R2 = {[i,1] -> [i+1,1] : 1 <= i < n && j=1}.

Figure 1 shows the loop iteration space and dependences for n=m=6. To produce a coarse-grained parallel program, we may proceed as follows. Using relation R1, we can generate code scanning synchronization-free slices. For this

Wlodzimierz B. and Marek P. (2008).
USING MESSAGE PASSING FOR DEVELOPING COARSE-GRAINED APPLICATIONS IN OPENMP.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 145-152
DOI: 10.5220/0001873801450152
Copyright © SciTePress

purpose, we can apply any well-known technique, for example those presented in (Beletska 2007, Bielecki 2008). The following loop, scanning synchronization-free slices being described by relation R1(see Figure 2), is generated by the Omega Calculator *codegen* function (Kelly 1995).

```
if(n >=2)
for(t1 = 1; t1 <= n; t1++) {
    a(t1,1) = a(t1,0) + a(t1-1,1);
    if (n >= t1 && t1 >= 1) {
      for(t2 = 2; t2 <= n; t2++) {
         a(t1,t2) = a(t1,t2-1) +
                    a(t1-1,1);
      }
}
```

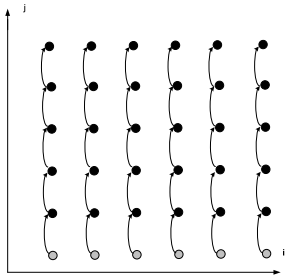

Figure 2: Synchronization-free slices described by R1.

To made this code to be semantically the same as the working loop, we may insert in it send and receive functions responsible for preserving dependences represented by R2. To properly insert and execute send and receive functions, the values of the iteration index vector have to be checked. Let us suppose that a pair of associated send and receive functions are executed in such a way that the receive function will wait until the execution of the correspondent send function is completed. Then for the motivating example, we may apply the following way of inserting and executing send and receive functions.

If the iteration index vector, I, belongs to the set SET_RECV = (Domain(R1) union Range(R1)) intersection Range(R2), comprising all the dependence sources and destinations described by R1 that are simultaneously dependence destinations described by R2, then a receive function must be inserted and executed prior the first loop statement that executes at iteration I. Its arguments are coordinates of the vector $J= R2^{-1}(I)$.

The correspondent send function must be inserted and executed after the last loop statement executing at iteration $J= R2^{-1}(I)$. Its arguments are also coordinates of J. That is, we may form a set SET_SEND as $R2^{-1}$(SET_RECV). It comprises all the iterations, after which execution, send functions must be executed.

To preserve the presented rules, we may insert send and receive functions as the body of *if* condition statements.

Using the Omega Calculator, we get the following sets for the working example

SET_RECV={[i,1]:2<=i<=n},
SET_SEND={[i,1] : 1 <= i <= n-1}.

Below we present the loop being formed taking into account the above presented rules. Figure 3 illustrates for which iterations send and receive functions should be executed and what are their arguments.

```
if (n >= 2) {
par for(t1 = 1; t1 <= n; t1++) {
    if(2 <= t1 && t1 <= n)
       recv(t1-1,1);
    a(t1,1) = a(t1,0) + a(t1-1,1);
    if(1 <= t1 && t1 < n)
       send(t1,1);
    if (n >= t1 && t1 >= 1) {
      for(t2 = 2; t2 <= n; t2++) {
         a(t1,t2) = a(t1,t2-1) +
                    a(t1-1,1);
      }
    }
}
```

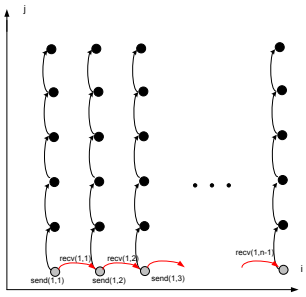

Figure 3: Parallel threads with synchronization.

3 ALGORITHM OF GENERATING COARSE-GRAINED LOOPS WITH SEND AND RECEIVE FUNCTIONS

Below we present an algorithm permitting for proper inserting and executing send and receive functions.

Algorithm. Inserting send and receive functions.

Input. 1) set S1 including dependence relations originating at least two synchronization-free slices; 2) set S2 comprising dependence relations to be used for the synchronization of the execution of slices produced by set S1(set S1 and set S2 together comprise all dependence relations extracted for the source loop); 3) relations Rel1 and Rel2 representing the unions of all the relations being comprised in set S1 and set S2, respectively; for sets S1 and S2, the following limitation is satisfied: Domain (Rel2) union Range (Rel2) ∈ Domain (Rel1) union Range (Rel1).

Output. Coarse-grained parallel code.

Method.
1. Using relations in set S1, generate code scanning synchronization-free slices applying any well-known technique, for example (Beletska 2007, Bielecki 2008) and extract all sets, I_j, representing the iteration space of statement st_j, j=1,2,...,q; q is the number of the loop body statements.
2. Calculate set SET_RECV_i = (Domain(Rel1) union Range(Rel1)) intersection Range($R2_i$) for each relation $R2_i$ in S2, i=1,2,...,n; n is the number of relations in S2.
3. Calculate set SET_SEND = $Rel2^{-1}$(Domain(Rel1) union Range(Rel1)) intersection Range(Rel2).
4. Before each statement, st_j, of the code generated in step 1 and for each relation $R2_i$, if (SET_RECV_i intersection I_j) ≠ FALSE insert the following code

```
if(I ∈ SET_RECVi)
  recv(R2i-1(I)); /*I is the
iteration vector being defined by the
loops surrounding stj */
```

5. After each statement st_j of the code generated in step 1, if (SET_SEND intersection I_j) ≠ FALSE insert the following code:

```
if(I ∈ SET_SEND)
  send(I); /*I is the iteration
vector being defined by the loops
surrounding stj */
```

Let us generate now code for the motivating example according to the presented algorithm.

Input.
S1 = {R1}, Rel1 = {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n},
S2= {R2}, Rel2 = {[i,1] -> [i+1,1] : 1 <= i < n && j=1}.

Step 1.

```
if (n >= 2) {
for(t1 = 1; t1 <= n;  t1++) {
   s(t1,1) ; /* a(t1,1) = a(t1,0) +
a(t1-1,1); */
   if (n >= t1 && t1 >= 1) {
     for(t2 = 2; t2 <= n; t2++) {
        s(t1,t2);  /* a(t1,t2) =
         a(t1,t2-1) + a(t1-1,1); */
     }
   }
```

I_1 = {[t1,1] : t1 >= 1 && t1 <=n },
I_2 = {[t1,t2 : t1 >= 1 && t1 <=n && t2 >= 2 && t2 <=n}

Steps 2,3.
SET_RECV_1={[i,1]:2<=i <= n}, SET_SEND={[i,1] : 1 <= i <= n-1}.

Steps 4,5.

```
if (n >= 2) {
for(t1 = 1; t1 <= n;  t1++) {
   if(2 <= t1 && t1 <= n)
     recv(t1-1,1);
   s(t1,1) ;          // st1
   if(1 <= t1 && t1 < n)
     send(t1,1);
   if (n >= t1 && t1 >= 1) {
     for(t2 = 2; t2 <= n; t2++) {
       s(t1,t2);  //st2
     }
} }
```

For I_2, we have SET_RECV_2 intersection I_2 = FALSE, SET_SEND_2 intersection I_2 = FALSE, therefore send and receive functions are not inserted for statement st_2.

4 IMPLEMENTATION OF SEND AND RECEIVE FUNCTIONS

In order to implement send and receive functions, we use lock functions. A lock function takes a lock variable as an argument. Firstly we present send and receive functions based on OpenMP lock functions, then we demonstrate how they can be implemented on the basis of POSIX lock functions.

4.1 Send and Receive Functions based on OpenMP Lock Functions

For each vector I, belonging to set S_SEND or set S_RECV, a structure, *s_synch,* is created. The structure consists of a binary released lock *mutex* and a boolean variable *executed* initialized with the false value. Below we present the code of the send

function.

```
void send(int t1, int t2,...,int tn)
{
 struct s_synch *set_c =
           &SET_SEND[t1][t2]..[tn];
  omp_set_lock(&set_c->mutex);
  set_c->executed = 1;
  omp_unset_lock(&set_c->mutex);
}
```

The arguments of the send function are coordinates of vector I, n is the number of coordinates of I. Firstly the function gets the reference to a correspondent structure. The function *omp_set_lock* acquires the correspondent lock and waiting until it becomes available, if necessary. The next statement sets the variable *executed* to true. The function *omp_unset_lock* releases the lock *mutex,* resuming a waiting thread.

Below is the code of the receive function.

```
void recv(int t1, int t2,...,int tn)
{
 struct s_comm *set_c =
          &SET_RECV[t1][t2]...[tn];
  while(1)
  {
     omp_set_lock(&set_c->mutex);
     if(!(set_c->executed))
         omp_unset_lock(
                  &set_c->mutex);
     else {
         omp_unset_lock(
                  &set_c->mutex);
         break;
     }
  }
}
```

The arguments of the receive function are coordinates of vector I. Firstly the function gets the reference to a correspondent structure. In order to check the variable *executed,* the lock has to be set by means of the function *omp_set_lock*. A thread executes the while loop as long as the value of the variable *executed* is false (at each iteration, it releases the lock and next locks the lock again).

When the value of the variable *executed* becomes true(a correspondent send function was executed), the execution of the receive function is terminated allowing a waiting thread to be resumed.

4.2 Send and Receive Functions based on POSIX Lock Functions

The disadvantage of the OpenMP lock functions is that waiting is active, causing wasting processor cycles. The send and receive functions, implemented with POSIX Threads (Posix 2004), do not posses such a disadvantage.

A condition variable in POSIX allows threads to wait until some event is occurred without wasting CPU cycles. Several threads may wait on the same condition variable until some other thread signals this condition variable (sending a notification). Then, one of the threads waiting on this condition variable wakes up. It is possible also to wake up all threads waiting on a condition variable by using a broadcast method on this variable.

It is worth to note that a condition variable does not provide locking. Thus, a lock is used along with a condition variable, to provide necessary locking when accessing this condition variable.

The code of the send function is as follows.

```
void send(int t1, int t2,...,int tn)
{
 struct s_comm *set_c =
&SET_SEND[t1][t2]...[tn];
 pthread_mutex_lock(&set_c->mutex);
 set_c->executed = 1;
     pthread_cond_broadcast(
                      &set_c->cond);
 pthread_mutex_unlock(
                     &set_c->mutex);
}
```

Firstly, the function *pthread_mutex_lock* acquires the lock on the *mutex* variable. If the *mutex* variable is already locked by another thread, the call of this function will block the calling thread until the mutex variable is unlocked. Then the variable *executed* is set to true, signaling that the iteration defined by vector I=(t1,t2,…,tn) has been executed. The POSIX function *pthread_cond_brodcast* signals all waiting threads on the condition variable *cond* and causes their awaking up. Finally, the function *pthread_mutex_unlock* unlocks the *mutex* variable. Below the code of the receive function is presented.

```
void recv(int t1, int t2,...,int tn)
{
 struct s_comm *set_c =
       &SET_RECV[t1][t2]...[tn];
 pthread_mutex_lock(&set_c->mutex);
 if(!(set_c->executed))
      pthread_cond_wait(
       &set_c->cond, &set_c->mutex);
 pthread_mutex_unlock(
                     &set_c->mutex);
}
```

Firstly the lock is locked. Then the variable *executed* is checked. If its value is false, the thread, calling the receive function, is put to sleep. The function *pthread_cond_wait* blocks the calling thread until the specified condition, *cond,* is signaled. This function is called while the mutex variable is locked, and it will automatically release the mutex variable while it waits. After the signal is received and the tread is awakened, the mutex variable will be automatically locked for use by the tread. Finally the thread releases the lock calling the function *pthread_mutex_unlock*. If the variable *executed* was set to *true,* the thread only releases the *mutex* variable and continues computing.

5 SLICES AGGLOMERATION AND ORDER OF ITERATIONS SCANNING

To reduce the volume of synchronization among threads, it is worth to agglomerate several slices into one macro-slice to be executed by an OpenMP thread. This permits for eliminating synchronization among slices within a macro-slice. For this purporse, we can combine several slices into one macro-slice. Let us suppose that applying agglomeration we produced N macro-slices. Then we add an additional parallel loop to scan those N macro-slices. The remaining inner loops with new lower and upper bounds *lb, ub*(their values depend on N) are executed serialy and scan iterations of a particular macro-slice.

To avoid inner synchronization in a macro-slice, additional constraints in *if* statements must be added (step 4, 5 in the algorithm presented in Section 3). We have to check for each vector I whether iterations $R2_i^{-1}(I)$ and R2(I) belong to the iterations of a macro-slice. If this is not true, synchronization is required.

Program performance will depend not only on the volume of synchronization but also on how we scan iterations of macro-slices. Figure 4 represents two different ways to scan iterations of macro-slices. On the left-hand side, iterations are scanned in lexicographic order, for example, for the first macro-slice the order of the iteration execution is (1,1), (1,2),...., (2,1), (2,2),.... On the right-hand side, iterations are executed according to the free-scheduling policy (Darte 2000) that supposes that in each step we execute those iterations whose input data is already completed, i.e., in each step the iterations between a pair of skewing neighborhood lines are executed (the order of their execution can be arbitrary because they all have completed data). There is barrier synchronization in the end of each step. For example, for the first macro-slice the following order of the iterations execution is valid: step1: (1,1), step2: (2,1) , (1,2), step3: (3,1), (2,2), (1,3) and so on. It is obvious that applying the free-scheduling policy improves the program performance because it reduces the waiting time. For example, applying lexicographic scanning, the second macro-slice can start its execution after the execution of 13 iterations(see Figure 4) while applying the free-scheduling policy, we can start the execution of the second macro-slice after the execution of 4 iterations only.

In paper (Bielecki 2008), we proposed how to generate code scanning iterations according to the free-scheduling policy. For scanning macro-slices, we should add a tuple to a set holding iterations to be executed at the next step (Bielecki 2008), only if an iteration $R2_i(I)$ belongs to the iterations of a macro-slice.

Below, we present the pseudo-code that scans iterations of macro-slices for our working example according to the free-schedule policy. This code is generated on the basis of the approach presented by us in paper (Bielecki 2008), and the described above way to insert send and receive functions. The additional constraints permitting for avoiding inner synchronization are bolded in the pseudocode below.

```
par for w = 1 to N {
  //  calculate  upper  and  lower
  //bounds, ub and lb as follows
 // pack = ((o_ub-1)+1)/N,
 //lb = pack *(w-1)+o_lb;
 // ub = (w==N) ? n : pack *w;
 // where o_lb, o_ub are the lower
//and upper bounds of the outermost
//loop generated in step 1 of the
//algorithm presented in Section 3
  S' = Ø;
  for t=lb to ub {
      I = [t,1]
      Add I to S';
  }
  while(S' != Ø) {
    S_tmp= Ø;
    foreach(vector I=[i,j] in S') {
       if(j==1 && 2 <= i && i <= n &&
!(i-1>=lb  &&  i-1  <=  ub)) /*
receive  message  from  another
macro-slice */
        recv(i-1,1);
      s1(I);
      if(j==1 && 1 <= i && i < n &&
( !(i+1>=lb && i+1 <= ub) ) )
```

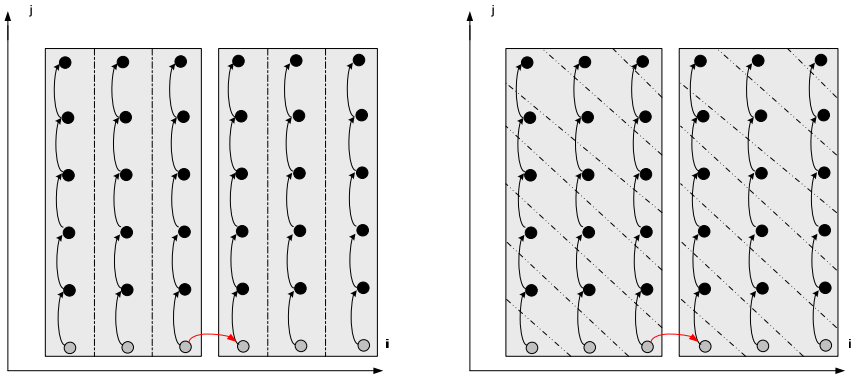

Figure 4: Illustrating slices agglomeration and the order of iterations scanning.

Table 1: Loops for experiments.

Loop 1	Loop 2	Loop3

for i=1 to n do
for j=1 to n do
a(i,j) = a(i,j-1) + a(i-1,1)/1.1
endfor
endfor
	for i=1 to n do
	a(i, 1) = a(1, i)
	for j=1 to n do
	a(i,j) = a(i,j+1)/1.001
	endfor
	endfor
		for i=1 to n do
		for j=1 to n do
		if(i==2 and j==1) then
		a(i,j) = a(i-1,j)
		endif
		a(i,j) = a(i,j-1)/1.1 + a(i-2,j+1)/1.1
		endfor
		endfor

```
   /*send message to another macro-
slice */
      send(i,1);

  if(1 <= j && j < n && 1 <= i &&
  i <= n){    /* if R1(I ∈ domain
  R1*/
     add J=[i,j+1] to S_tmp;
  /* J=[ip,jp]= R2(I) */

  if(j==1 && 1<=i && i<n &&
  lb<=i+1 && i+1>=ub) /*if
  R2(I) ∈domain R2
  and I∈ Iterations of macro-slice
  */
     add J=[i+1,j] to S_tmp;
  /* J=[ip,jp]= R2(I) */
 }
 S' = S_tmp;  }
```

6 EXPERIMENTS

To evaluate the performance of programs being formed by the proposed approach, we have tested three loops (see Table 1). OpenMP parallel coarse-grained codes were produced by us on the basis of presented approach. The POSIX library (POSIX threads C+ access library2.2.1-2) was included in OpenMP sources to permit us to use send and receive functions based on POSIX lock functions. We have carried experiments on the machine Intel Xeon 1.6 Ghz, 8 processors (2 x 4-core CPU, cache 4 MB), 2 GB RAM, Ubuntu Linux.

The results are presented in Table 2, where time is presented in seconds, S and E denote speed-up and efficiency, respectively. The second column in Table 2 indicates whether agglomeration was used (+) or not(-), F means that free scheduling was used, N is the number of macro-slices.

For Loop 1 and Loop 2, we examined how agglomeration impacts speed-up and efficiency. Loop 3 originates only two slices, hence agglomeration was not considered.

Experiments with Loop 1 and Loop 2 demonstrate that i) increasing the volume of calculations executing by a thread(increasing n) increases S and E; ii) the positive impact of agglomeration on S and E grows with increasing the number of processors; iii) free scheduling is important for the number of processors greater than 2. However it is not obvious what is the value of N that optimizes program performance. As N grows, the waiting time to start the macro-slice execution decreases while the volume of synchronization increases.

Table 2: Results of experiments (POSIXfunctions).

	Aggl.	CPU	1		2			4			8		
Loop		n	seq. for	synch	time	S	E	time	S	E	time	S	E
Loop1	-	1500	0,0441	0,0462	0,0296	1,490	0,745	0,0231	1,281	0,320	0,0288	1,028	0,128
		2000	0,0782	0,0814	0,0475	1,646	0,823	0,0346	2,263	0,566	0,0390	2,005	0,251
		2500	0,1222	0,1271	0,0713	1,714	0,857	0,0461	2,651	0,663	0,0680	1,797	0,225
Loop1	+ / F N = 200	1500	0,0441	0,0470	0,0299	1,475	0,737	0,0213	2,070	0,518	0,0272	1,621	0,203
		2000	0,0782	0,0861	0,0479	1,633	0,816	0,0350	2,234	0,559	0,0312	2,506	0,313
		2500	0,1222	0,1300	0,0721	1,695	0,847	0,0487	2,509	0,627	0,0461	2,651	0,331
Loop2	+ N = 16	1500	0,0446	0,0452	0,0308	1,449	0,724	0,0251	1,777	0,444	0,0247	1,806	0,226
		2000	0,0793	0,0801	0,0516	1,537	0,768	0,0314	2,525	0,631	0,0318	2,494	0,312
		2500	0,1238	0,1249	0,0720	1,719	0,860	0,0458	2,703	0,676	0,0382	3,241	0,405
Loop 3	-	1500	0,0578	0,0580	0,0391	1,478	0,739	-	-	-	-	-	-
		2000	0,0843	0,0844	0,0527	1,600	0,800	-	-	-	-	-	-
		2500	0,1148	0,1153	0,0648	1,772	0,886	-	-	-	-	-	-

Table 3: Influence of the number of macro-slices on program performance (POSIX functions).

CPU	1		2			4			8		
N	seq. for	synch	time	S	E	time	S	E	time	S	E
32	0,0819	0,0908	0,0507	1,615	0,808	0,0428	1,914	0,478	0,0370	2,214	0,277
64	0,0819	0,0912	0,0498	1,645	0,823	0,0356	2,301	0,575	0,0346	2,367	0,296
128	0,0819	0,0878	0,0514	1,593	0,797	0,0324	2,528	0,632	0,0307	2,668	0,333
256	0,0819	0,0876	0,0538	1,522	0,761	0,0342	2,395	0,599	0,0364	2,250	0,281

Table 4: POSIX and OpenMP functions.

	1 CPU, s				2 CPU, s			4 CPU, s			8 CPU, s		
N	Seq.	OpenMP	POSIX	WS	OpenMP	POSIX	WS	OpenMP	POSIX	WS	OpenMP	POSIX	WS
512	0,0819	0,0894	0,0866	0,0837	0,0587	0,0516	0,0496	0,0392	0,0381	0,0378	0,0375	0,036	0,0359
	(6,8%;3,5%)				(18,3%;4%)			(3,7%;0,8%)			(4,5%;0,3%)		
256	0,0819	0,0888	0,0853	0,0852	0,0568	0,0501	0,0499	0,0345	0,0309	0,0306	0,0386	0,036	0,0317
	(4,2%;3,5%)				(13,8%; 0,4%)			(12,7%; 1,0%)			(21,8%; 13,6%)		
128	0,0819	0,0875	0,0866	0,0856	0,0576	0,0521	0,0512	0,0356	0,0318	0,0316	0,0365	0,0319	0,032
	(2,2%; 1,2%)				(12,5%; 1,8%)			(12,7%; 0,6%)			(14,1%; 0%)		
32	0,0819	0,1001	0,0965	0,0845	0,0585	0,056	0,0509	0,0431	0,0401	0,0371	0,0372	0,0343	0,0311
	(18,5%; 14,2%)				(14,9%; 10%)			(16,2%;8,1%)			(19,6%;10,3%)		

The size of a macro-slice impacts also program locality – a very important factor defining program performance. Table 3 presents how the number of macro-slices (Loop 1, free-scheduling with agglomeration), N, impacts S and E. For each number of processors, there exists a particular value of N optimizing program performance.

Table 4 presents differences in execution times of programs being written with OpenMP and POSIX lock functions for Loop 1, n=2048. The columns named as WS present program execution times when send and receive functions are removed from code. In the brackets, the percentage of the program execution time is inserted that characterizes synchronization time on the basis of OpenMP and POSIX functions, respectively. The data in Table 4 demonstrates that POSIX functions permit for better program performance in comparison with OpenMP functions.

7 RELATED WORK

Unimodular loop transformations (Banerjee 1990), permitting the outermost loop in a nest of loops to be parallelized, find synchronization-free parallelism. However when there exist dependences between slices, that approach fails to extract coarse-grained parallelism.

The affine transformation framework (polyhedral approach), considered in many papers, for example, in papers (Darte 2000, Feautrier 1994, Lim 1999) unifies a large number of previously proposed loop transformations. It permits for producing affine time and space partitioning and correspondent parallel codes. For the motivating example, there does not exist any affine space partitioning permitting for extracting coarse-grained parallelism; we are able to find only time partitioning to extract fine-grained parallelism.

Our contribution consists in demonstrating how to generate coarse-grained code on the basis of two sets of dependence relations, the first of them permits for extracting synchronization-free slices (our previous results: Beletska 2007, Bielecki 2008) while the second one is used for inserting send and receive functions. The proposed approach extracts coarse-grained parallelism when in a loop there does not exist synchronization-free slices.

8 CONCLUSIONS AND FUTURE WORK

We introduced the approach to extract coarse-grained parallelism in loops. Send and receive functions are used to synchronize the slices execution. Our future research direction is to derive techniques permitting for automatic splitting a set of dependence relations into two sets such that the first one is to be used for generating code scanning slices while the second one is to insert send and receive functions.

REFERENCES

OpenMP Application Program Interface 2007. In *http://www.openmp.org*

Darte A., Robert Y., Vivien F., 2000. *Scheduling and Automatic Parallelization,* Birkhäuser Boston.

Pugh W., Wonnacott D., 1998. *Constraint-based array dependence analysis.* In ACM Trans. on Programming Languages and Systems.

Beletska A., Bielecki W., San Pietro P., 2007 *Extracting Coarse-Grained Parallelism in Program Loops with the Slicing Framework.* In Proceedings of ISPDC'2008.

Bielecki W., Beletska A., Pałkowski M., San Pietro P., 2008. *Finding synchronization-free parallelism represented with trees of dependent operations.* In Proceedings of ICA3PP'2008.

Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott, D., 1995. *The omega library interface guide*. Technical Report CS-TR-3445, University of Maryland.

The POSIX 1003.1-2001, 2004. *Standard of an application programming interface (API) for writing multithreaded applications. In* http:// www.unix.org/version3/ieee_std.html

Banerjee U., 1990. *Unimodular transformations of double loops.* In Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing, pp. 192-219.

Feautrier P., 1994. *Toward automatic distribution.* Journal of Parallel Processing Letters 4, pp. 233-244.

Lim W., Cheong G.I., Lam M.S., 1999. *An affine partitioning algorithm to maximize parallelism and minimize communication.* In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing.

Kelly W., Pugh W., Rosser E. and Shpeisman T., 1996. *Transitive Closure of Infinite Graphs and its Applications.* International Journal of Parallel Programming, v. 24, n. 6, pp. 579-598.

Allen, R, Kennedy, K. 2001. *Optimizing Compilers for Modern Architectures*, pages 790, Morgan Kaufmann.