Maciej POLIWODA
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE

The efficiency of parallel OpenMP loop code produced by the hyperplane method

Dr inż. Maciej POLIWODA

Od 1998r zatrudniony na Wydziale Informatyki Politechniki Szczecińskiej. W roku 2003 uzyskał stopień naukowy doktora.

e-mail: mpoliwoda@wi.zut.edu.pl

Abstract

The efficiency of loops parallelized by the hyperplane method is considered in the paper. The improvement of the parallel loop code efficiency was explored across improvement the locality of calculations. The main goal of presented research is disclosing whether is it possibly and what is the area of the hyperplane method parallelize loops, and how the improvement of data locality influences the improvement of the parallel loop code efficiency.

Keywords: loop parallelization, OpenMP, hyperplane method.

Efektywność kodu pętli automatycznie zrównoleglonego metodą hiperpłaszczyzn

Streszczenie

W artykule przedstawiono wyniki badań efektywności kodu pętli zrównoleglonego metodą hiperpłaszczyzn w odniesieniu do kodu pętli zrównoleglonego innymi metodami, z uwzględnieniem efektywności wynikającej z użycia różnych kompilatorów. Dodatkowo przeprowadzono badania poprawy efektywności zrównoleglenia poprzez zwiększenie lokalności obliczeń. Celem przeprowadzonych badań było określenie czy i w jakim obszarze kod zrównoleglony metodą hiperpłaszczyzn może być efektywny i w jakim stopniu zwiększenie lokalności obliczeń wpływa na poprawę efektywności kodu.

Słowa kluczowe: zrównoleglenie pętli, OpenMP, metoda hiperpłaszczyzn.

1. Introduction

The process of the transformation of sequential code to parallel one is difficult and time-consuming. The aim of the parallelization process is to find in sequential code program fragments, which can be executed by independently working processors of a multiprocessor system. The main goal of the transformation of sequential code to parallel one is the reduction of the execution time in a multiprocessor system.

Parallelizing a large part of program instructions is the condition for the obtaining the effective parallel code. This condition can be satisfied by parallelizing program loop code, which executes large parts of program instructions.

The hyperplane method for loops parallelization was implemented as a result of my previous research. An implemented tool extracts dependencies between loop iterations and generates different versions of parallel loop code. The next step is the finding such a version of parallelized loop code that improves data locality of calculations to effectively use processors of a multiprocessor system.

This paper presents the efficiency of loop code parallelized by the hyperplane method and other well known methods, and being compiled by different compilers. The experiments show the improvement of the parallel loop code efficiency due to the improvement of data locality.

2. Dependencies between loop iterations

In this paper the loop code presented in Fig. 1 executing matrix multiplication is considered. The first step in the parallelization process is to describe dependencies between loop iterations. The dependencies description allows finding independent iterations, which can be executed by threads. The dependency analysis methods are described in [1, 3, 10, 12].

The dependency vector between the iteration described by vector $\overline{I} = [i_1, \ldots, i_n]^T$, and the iteration described by $\overline{J} = [j_1, \ldots, j_n]^T$, where iteration $\overline{J}$ is depended on $\overline{I}$, is defined by $\overline{K} = \overline{J} - \overline{I} = [j_1 - i_1, \ldots, j_n - i_n]^T$. If the iterations $\overline{I}$ and $\overline{J}$ satisfy the lexicographical order $\overline{I} \prec \overline{J}$, the k exists for $1 \le k \le n$, where $i_1 = j_1, \ldots, i_{k-1} = j_{k-1}$, and $i_k < j_k$, the dependency vector is lexicographically greater then a zero vector, then $\overline{K} \succ \overline{0}$.

```
for (i = 0; i < s1; i++)
    for (j = 0; j < s2; j++)
        for (k = 0; k < s3; k++)
        a[i][j] += b[i][k] * c[k][j];
```

Fig. 1. Loop example
Rys. 1. Przykładowa pętla

The iteration $\overline{J}$ is depended on $\overline{I}$, when the iterations $\overline{J}$ and $\overline{I}$ refer the same memory area and one of this references is a write operation.

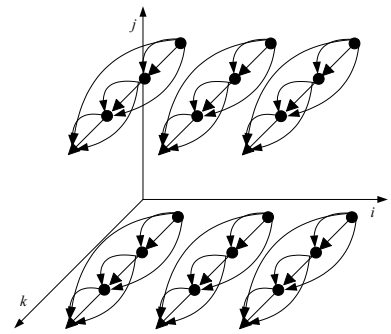


Fig. 2. Loop iteration space with dependencies
Rys. 2. Przestrzeń iteracji z zależnościami

In Fig. 2 flow, anti, and output dependencies between iterations in the loop of Fig. 1, are presented. They are described by the dependency vector $K=[0\ 0\ 1]^T$.

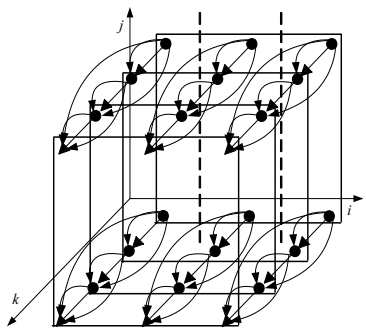
3. The hyperplane method

The "hyperplane method" called as the „wave-front-method" or „Lamport method" is described in [6, 7, 11]. The rows of $m \times n$ matrix C, where $m \le n$, describe hyperplane. The row vector $\overline{C}_i$ for $i=1,2,\ldots m$ describes a hyperplane on i-level. For all dependency vectors in an m-dimensional loop where $\overline{K} \succ \overline{0}$, the matrix C must satisfy the condition $C\overline{K} > 0$ to keep the lexicographically order of dependent iterations. For matrix C, where $\overline{C}\overline{K} > 0$, iterations $\overline{I}$ such that $C\overline{I} = t$, belong to the same hyperplane and can be executed in parallel. For the loop of Fig. 1, the matrix C satisfying

the condition $C\overline{K} > 0$, for dependency vectors on Fig. 2, is as follows:

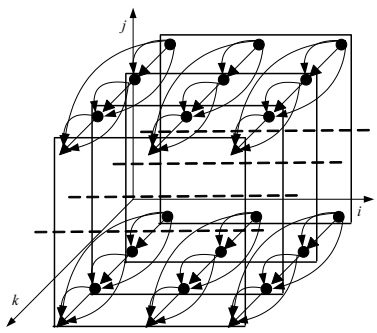
$$C = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

For this matrix describing hyperplane, there exist two versions of loop parallel code after the sequential loop transformation [4, 10, 11], where inner loops can be executed in parallel, which are described in Fig. 3 and Fig. 4.




```
#pragma omp parallel private(k,j) shared(i,a,b,c)
{
        for(k=0;k<size;k++){
#pragma omp for schedule(runtime)
                for(i=0;i<size;i++){
                        for(j=0;j<size;j++){
                                a[i][j]+=b[i][k]*c[k][j];
                        }
                }
        }
}
```

Fig. 3. Parallel code created with the hyperplane method (hpari)
Rys. 3. Kod pętli zrównoleglony metodą hiperpłaszczyzn (hpari)

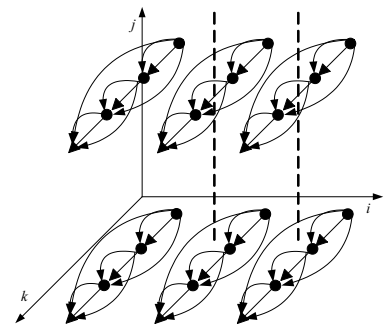



```
#pragma omp parallel private(k,i) shared(j,a,b,c)
{
        for(k=0;k<size;k++){
                for(i=0;i<size;i++){
#pragma omp for schedule(runtime)
                        for(j=0;j<size;j++){
                                a[i][j]+=b[i][k]*c[k][j];
                        }
                }
        }
}
```

Fig. 4. Parallel code created with the hyperplane method (hpari)
Rys. 4. Kod pętli zrównoleglony metodą hiperpłaszczyzn (hparj)

4. Alternative methods

The alternative parallel codes for a loop of Fig. 1 are presented in Fig. 5, where the outer loop is parallelized, Fig. 6a, where the inner loop was parallelized, and Fig. 6b, where the most inner loop was parallelized by the reduction method.




```
#pragma omp parallel for private(j,k,sum) schedule(runtime)
        for(i=0;i<size;i++){
                for(j=0;j<size;j++){
                        sum = 0;
                        for(k=0;k<size;k++){
                                sum+=b[i][k]*c[k][j];
                        }
                        a[i][j]+=sum;
                }
        }
```

Fig. 5. Code with the parallel outer loop (pari)
Rys. 5. Zrównoleglenie iteracji pętli zewnętrznej (pari)

a)

```
#pragma omp parallel private(i,sum) shared(j,k,a,b,c)
{
        for(i=0;i<size;i++){
#pragma omp for nowait private(k) schedule(runtime)
                for(j=0;j<size;j++){
                        sum = 0;
                        for(k=0;k<size;k++){
                                sum+=b[i][k]*c[k][j];
                        }
                        a[i][j]+=sum;
                }
        }
}
```

b)

```
#pragma omp parallel private(i,j,sum) shared(a,b,c)
{
        for(i=0;i<size;i++){
                for(j=0;j<size;j++){
                        sum = 0;
#pragma omp for nowait schedule(runtime)
                        for(k=0;k<size;k++){
                                sum+=b[i][k]*c[k][j];
                        }
#pragma omp critical
                        a[i][j]+=sum;
                }
        }
}
```

Fig. 6. Code with the parallel inner loops a) (parj), b) (park)
Rys. 6. Zrównoleglenie pętli wewnętrznych a) (parj), b) (park)

5. Data locality improvement

To improve data locality the blocking method was used. The iteration space was divided between iteration blocks such that calculations in iterations from one block were executed with the use of local memory. The function assigning iterations between blocks is described in Fig. 7. The data locality for *m* - dimensional arrays, which are referenced by the linear index expression, is described in [15]. The block size calculations [16] dependents on a system architecture are presented in Section 7. The loop code with blocking for the parallelized outer loop is presented in Fig. 8, and for the loop parallelized with hyperplane in Fig. 9.

```
intgetIterationBlock
(intiterator,int*start,int*end,intloopInit,intloopEnd,intblockSize){
      *start=loopInit+iterator*bloockSize;
      *end=*start+blockSize-1;
      *end=(((*end)>(loopEnd))?(loopEnd):(*end));
      return*start<=*end;
}
```

Fig. 7. Function for assigning the loop iteration space
Rys. 7. Kod funkcji dzielącej przestrzeń iteracji pętli

```
#pragma omp parallel private(k, j, i, ii, is, ie, jj, js, je, kk, ks, ke) shared(a, b, c)
  {
    for(ii = 0; getIterationBlock(ii,&is,&ie,0, size - 1, blockSize); ii + +){
      for(jj = 0; getIterationBlock(jj,&js,&je,0, size - 1, blockSize); jj + +){
        for(kk = 0; getIterationBlock(kk,&ks,&ke,0, size - 1, blockSize); kk + +){
#pragma omp for schedule (runtime)
          for(i = is; i <= ie; i + +){
            for(j = js; j <= je; j + +){
              for(k = ks; k <= ke; k + +){
                a[i][j] + = b[i][k] * c[k][j];
  } } } } } } }
```

Fig. 8. Outer loop parallelized together with blocking (pari with blocking)
Rys. 8. Zrównoleglenie pętli zewnętrznej z podziałem przestrzeni iteracji

```
#pragma omp parallel private(k, j, i, ii, is, ie, jj, js, je) shared(a, b, c)
  {
    for(ii = 0; getIterati onBlock(ii ,&is,&ie,0 , size - 1, blockSize) ; ii + +){
      for(jj = 0; getIterati onBlock(jj ,&js,&je,0 , size - 1, blockSize) ; jj + +){
        for(k = 0; k < size; k + +){
#pragma omp for schedule (runtime)
          for(i = is; i <= ie; i + +){
            for(j = js; j <= je; j + +){
              a[i][j] + = b[i][k] * c[k][j];
  } } } } } } }
```

Fig. 9. Loop parallelized by the hyperplane together with blocking (hpari with blocking)
Rys. 9. Zrównoleglenie metodą hiperpłaszczyzn z podziałem przestrzeni iteracji

6. Environment

To estimate the efficiency of parallelized loop code, the speedup $S(P)=\frac{T(1)}{T(P)}$ were calculated for parallel code presented in the previous section. Here $T(1)$ - execution time of sequential loop, $T(P)$ - execution time of parallel loop on P processors. The experiments were conducted on the system with the following parameters:

Hardware:
2 x Intel Quad Core Xeon E5310 1,60 GHz 1U
The Intel Workstation Board S5000Xvn 1.97 GB RAM

Software:
Configuration I
Operating Linux Ubuntu
Compilers [17, 18]:
Omni OpenMP Compiler (omni)
OMPi OpenMP C Compiler (ompi)

Configuration II
Windows XP Professional x64 Edition Ver.2003
Compilers [19, 20]:
Intel ® C++ Compiler for Windows* - Evaluation (icc)
Microsoft Visual Studio 2005 (vs2005)

7. Experiment results

The execution time of the loop of Fig. 1, compiled with the compilers [17, 18, 19, 20], is presented on Fig. 10. The best results were obtained for the loop compiled by the compilers ompi and vs2005.

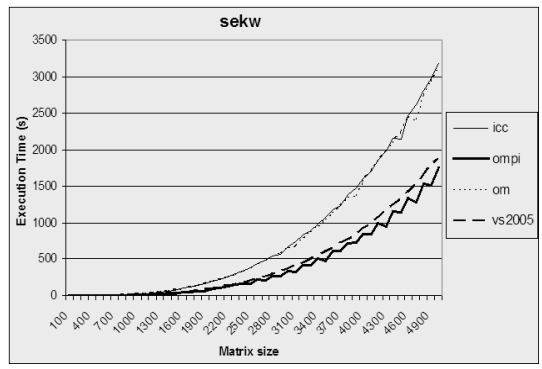


Fig. 10. Execution time of sequential loop
Rys. 10. Czas sekwencyjnego wykonania pętli

For the loops of Fig. 3, Fig. 4, Fig. 5, Fig. 6, the speedup is presented in Fig. 11 and Fig. 12, for the compilers vs2005 and ompi. The speedups retrieved by the other compilers have the same values.

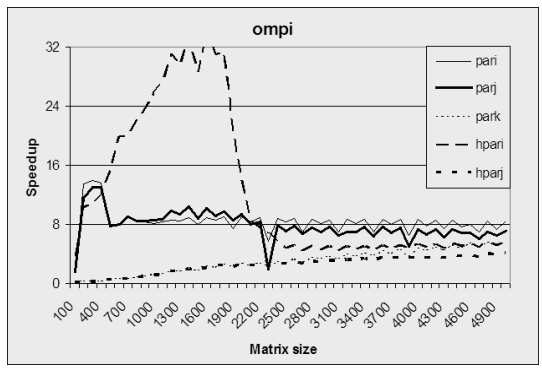


Fig. 11. Loop speedup, the ompi compiler
Rys. 11. Przyspieszenie pętli, kompilator ompi

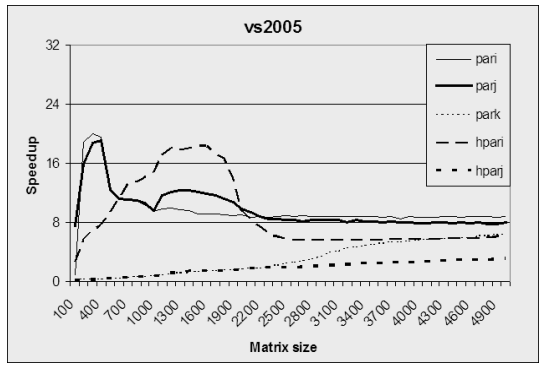


Fig. 12. Loop speedup, vs2005 compiler
Rys. 12. Przyspieszenie pętli, kompilator vs2005

The speedup takes maximal values due to high data locality, for the loop of Fig. 3 (hpari), where the matrix size is between 400 and 2200, and the loops of Fig. 5 (pari), Fig. 6a (parj), where the matrix size is between 200 and 400.

To improve data locality, the iteration space can be divided between iteration blocks like in the loops of Fig. 8 and Fig. 9.The dependence between the parallel execution time for the loops of Fig. 8 and Fig. 9, for the matrix size N=3000, and the size of the iteration block is presented in Fig. 13 and Fig. 14, respectively.

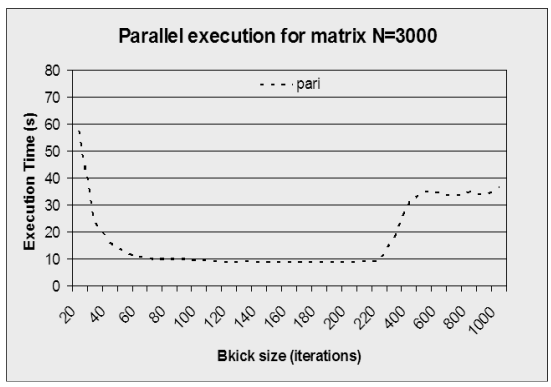


Fig. 13. Parallel execution loop of Fig. 8 for N=3000
Rys. 13. Wykonanie równoległe pętli z rys. 8 dla N=300

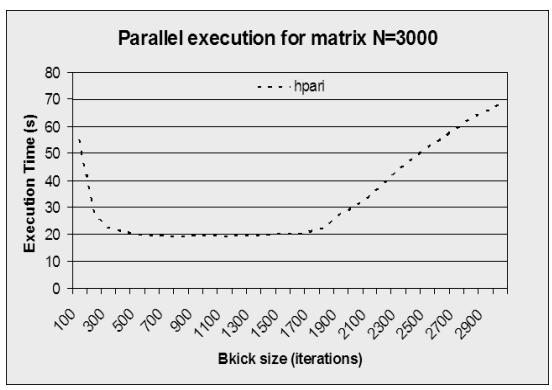


Fig. 14. Parallel execution loop of Fig. 9 for N=3000
Rys. 14. Wykonanie równoległe pętli z rys. 9 dla N=300

For the loop of Fig. 8 the best iteration block size is B=160, and for the loop of Fig. 9 the best iteration block size is B=800. The iteration blocks sizes for the loop of Fig. 8 and Fig. 9 were used to other matrix size, Fig. 15 and Fig. 16.

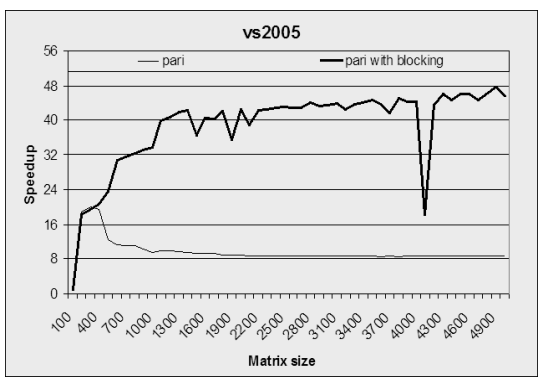


Fig. 15. Speedup for the loops of Fig. 5 and Fig. 8
Rys. 15. Przyspieszenie dla pętli z rys. 5 i rys. 8

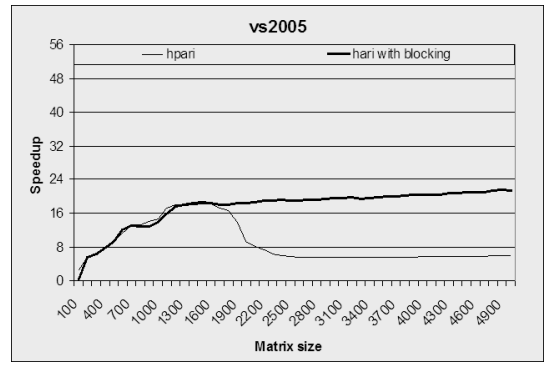


Fig. 16. Speedup for the loops of Fig. 3 and Fig. 9
Rys. 16. Przyspieszenie dla pętli z rys. 3 i rys. 9

8. Conclusion

The efficiency of loops parallelized by the hyperplane method is worse than that of the loops with a parallel outer loop, but we cannot parallelize all outer loops because of dependencies between iterations.

The blocking method improves data locality and speedup for loops with a parallel outer loop and loops parallelized with the hyperplane method.

The main goal of future research is creating an algorithm for the calculation of the iteration block size for different system architectures, data references and loop parallelization methods.

9. References

[1] Allen R., Kennedy K.: Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2001.
[2] Bacon D., Graham S., Sharp O.: Compiler transformations for high-performance computing. Computing Surveys, 26(4):345-420, December 1994.
[3] Banerjee U.: Loop Transformations for Restructuring Compilers. Kluwer Academic, 1993.
[4] Beletskyy V., Poliwoda M.: Parallelizing perfectly nested loops with non-uniform dependences. In Proceedings of the Advanced computer systems, pages 83-98, October 2002.
[5] Cameron H., Tracey H.: Parallel and Distributed Programming Using C++. Prentice Hall Professional , 2003, 720 pp.
[6] Darte A., Robert Y., Vivien F.: Scheduling and Automatic Parallelization. Birkhäuser Boston, 2000.
[7] Lamport L.: The Parallel Execution of DO Loops. Communications of the ACM, Vol. 17, No.2, Feb. 1974, pp. 83-93.
[8] Schrijver. Theory of Linear and Integer Programming. Wiley, Chichester, 1986.
[9] Watkins D., Hammond M., Abrams B.: Programming in the .NET Environment. Addison-Wesley, 2003.
[10] Wolfe M.: High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, 1995.
[11] Wolf M. E., Lam M. S.: A data locality optimizing algorithm. In Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, pages 30–44, June 1991.
[12] Zima H., Chapman B.: Supercompilers for Parallel and Vector Computers. ACM Press, 1990.
[13] http://www.openmp.org/drupal/
[14] Bielecki W., Poliwoda M.: Hyperplane method for loops parallelization in the .Net environment, Advanced Computer Systems - mat. konf., Szczecin, 2006.
[15] Wolfe M.: High Performance Compilers for Parallel Computing, Addison Wesley, 1996.
[16] Griebl M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures, Habilitation, Universitat Passau, 2004.
[17] Omni OpenMP Compiler Project; http://phase.hpcc.jp/Omni/
[18] OMPi OpenMP C Compiler; http://www.cs.uoi.gr/~ompi/
[19] Intel ® C++ Compiler for Windows* - Evaluation; http://www.intel.com/cd/software/products
[20] Microsoft Visual Studio 2005; http://www.microsoft.com

Artykuł recenzowany