Compilation and Parallelization Techniques with Tool Support to Realize Sequence Alignment Algorithm on FPGA and Multicore

Sunita Chandrasekaran[1], Oscar Hernandez[2], Douglas L. Maskell[1], Barbara Chapman[2], and Van Bui[2]

[1] School of Computer Engineering
Nanyang Technological University
{suni0003, asdouglas}@ntu.edu.sg
[2] Department of Computer Science,
University of Houston
{vtbui, oscar, chapman}@cs.uh.edu

Abstract. Reconfigurable computing (RC), such as computing using field programmable gate array (FPGA) technology has been shown as the field to accelerate a large variety of applications. RC fills the gap between hardware and software, achieving high performance on the hardware than the software and at the same time maintaining a remarkable amount of flexibility. Though there are bottlenecks associated with using the FPGA accelerators with legacy application code, it is becoming difficult to decide which parts of the code should be implemented in hardware (versus software), to provide an efficient mapping from code to the highly parallel FPGA fabric and to evaluate the costs associated with running in a hybrid (hardware/software) mode. In this paper we present a methodology to tune an application with the help of a tool environment consisting of an open source parallelizing compiler, and static and performance analysis tools. This serves as a high performance tuning strategy for identifying the bottlenecks in the application code, followed by preprocessing, before mapping the algorithm to the FPGA in order to take advantage of the intrinsic speed. Using this performance toolset and our tuning methodology, we were able to parallelize and tune a bioinformatics application to produce better load balances and higher performance, yielding almost linear speedup, of up to 80% on dynamic scheduling with 128 threads on a 1000 sequence data set.

1 Introduction

Reconfigurable computing (RC) is intended to fill the gap between the high speed, low flexibility of hardware and the low speed, high flexibility of software, achieving potentially a much higher performance than software while maintaining a higher level of flexibility than hardware. FPGAs are capable of achieving speedups of up to 500 times and energy savings of up to 70% [1] over microprocessor implementations of a specific application. Moving just the critical software loops to a reconfigurable fabric results in average energy savings of between 35% to 70% with an average speed-up of 3 to 7 times, depending on the particular device used [1]. One of the prime factors of using

FPGAs is to fully access the significant speed improvement automatically but unfortunately there is the lack of supporting toolsets. The existing toolsets are not yet mature enough to exploit the advantages of using the reconfigurable platform. This productivity gap between design complexity and design capacity [2] has put the benefits of FPGA performance improvements beyond the reach of the majority of software developers, who do not have the ability or inclination to go into the extensive detail required for designing efficient hardware. Unfortunately, there is still an order-of-magnitude gap between what can be achieved by a good design engineer [3] [4] and that which is able to be achieved by state-of-art electronic design automation (EDA) tools [5].

In addition to tools, a good programming model and compilation system needs to be explored and this is highly dependent on the hardware being used. For example if the FPGA is attached to an SMT or multicore processor and memory (e.g. via high bandwidth low latency buses), we can use a shared memory programming model within the cores. In most of the cases, an interface via libraries (using Impulse-C), buffering of shared data, or DMA accesses, might be needed to send the data to the FPGAs. This approach can combine the shared memory model of OpenMP mixed with the stream programming capabilities of the FPGAs. In this paper we explore the use of OpenMP to exploit thread level parallelism and means to find the portions to be mapped to the accelerators via iterative tuning. We believe that this approach works because we can leverage time critical computations to FPGAs acceleration while multicore processors can do thread, memory and I/O management or focus on non-compute intensive tasks. With the intention of closing this productivity gap as the basis for our motivation, we use the multiple sequence alignment (MSA) problem as a benchmark. The design methodology for mapping this algorithm to FPGA is based on hardware descriptive languages such as VHDL and Verilog HDL. An attractive alternative is to use a high level language such as 'C' to describe the algorithm and generate equivalent hardware descriptions for implementation in FPGA. We use the portable open source Open64/OpenUH compilers and the OpenMP application programming interface (API) for multi-platform shared-memory parallel programming. We use performance analysis tools to identify the bottlenecks in the software code, optimize and parallelize the application code to achieve the desired result.

2 The Multiple Sequence Alignment Problem

Molecular biologists frequently compute MSAs, comparing protein sequences with unknown functionality to a set of known sequences to detect functional similarities [6]. Among the heuristic methods available the progressive alignment method [7] is widely used. ClustalW [8] makes use of such a method and consists of three stages: distance matrix, guided tree and progressive alignment along the tree. Profiling of the ClustalW program on a single processor showed that almost 90% of the time is spent in the first stage [9]. The Smith-Waterman algorithm can be used to compute the optimal local alignment of two sequences [7]. Full details of the MSA stages and the SW algorithm can be found in [1] [2] [5] [6] [8] [10].

3 Tool Support

3.1 Open64/OpenUH High Level Optimizations

The OpenUH compiler [11], a branch of Open64 [12],is an optimizing and portable open-source OpenMP compiler for C/C++ and Fortran 90 programs. It is based on SGI's open source Pro64 compiler and pathscale EkoPath compiler, which targets the IA-64, IA-32 and Opteron Linux-based platform. OpenUH additionally supports OpenMP, provides static analysis to the user and supports libraries that facilitate the integration of performance tools. OpenUH's interprocedural analysis supports array region analysis that is a constraint-based technique proposed by Triolet [13] to describe array accesses/patterns [14]. Such analysis is critical to extract the parallelism of the code via an accurate inter-procedural dependence analyzer. OpenUH also performs loopnest dependence analysis that is based on the Omega test [13]. Fig. 1. gives an overview of OpenUH.

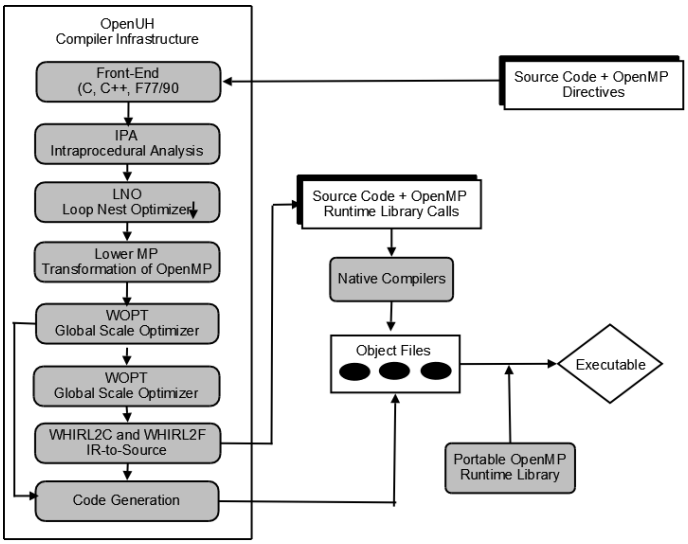


Fig. 1. An overview of OpenUH

3.2 The Dragon Analysis Tool

We use Dragon [15], a program analysis tool built on the top of the OpenUH compiler to support, analyze and optimize the OpenMP application. Dragon is helpful in gathering information about the threads accessing the elements of shared arrays at run time. The input languages to Dragon are Fortran77/95, C and OpenMP/MPI. The tool facilitates the task of writing OpenMP programs by analyzing the code for data dependencies and

understanding procedure side effects which could obstruct the functions or loop nests from being parallelized.

3.3 Tools Analysis and Utilities - TAU

TAU [16] [17] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java or Python. The results are displayed in aggregate and single node/context/thread forms, thus identifying the performance bottleneck. A number of profiling modes exists including: statistical sampling of the program counter or call stack; hardware counter sampling; instruction counting; and timer based instrumentation. The time and hardware counter data are analyzed by ParaProf and PerfExplore that are both parts of the TAU toolkit. The functions that require considerable execution time can be identified and function entry/exit points can be traced along with the messages between different nodes. Secondary cache miss behavior can also be tracked.

3.4 Perfsuite

PerfSuite [18] is a collection of tools, utilities, and libraries that together provide the user with several options for performance monitoring including support for hardware performance counting and profiling of application programs. PerfSuite consists of a set of command line utilities that includes support for statistical profiling, hardware performance counting, deriving high-level performance metrics, access to machine level characteristics, and customizing performance experiments. These utilities together enable both coarse and fine grain performance experiments for parallel applications with low overheads.

4 Execution and Tuning Model

Programs using OpenMP compiler can insert directives either manually or automatically with the help of a compiler or stand-alone utility inducing parallelism. We run the OpenMP threads on the multicore processor and propose to map the compute intensive inner loops to the FPGA accelerators as shown in Fig. 2. The serial portions that are not compute intensive are run in one core. The OpenMP parallel region threads running on the multiple cores perform software synchronizations (locks and barriers), I/O operations, and dynamic memory management. In our model, the accelerator streams reside inside a parallel region. More precisely, they reside inside a compute intensive **parallel do**. A chunk of work for a **parallel do** is assigned to a thread that invokes the accelerator to compute the inner loops via streaming. Our compilation model translates OpenMP to object code via a native compiler and a native OpenMP runtime library. In order to decide the portions that should run on the multicore processor as opposed to the FPGA, we first profile the code to recognize the most time consuming and computationally expensive hotspots in the program via compiler instrumentation and bottleneck analysis using hardware counters. The compiler logs of loop nest optimization and interprocedural analysis are closely examined for details pertaining to

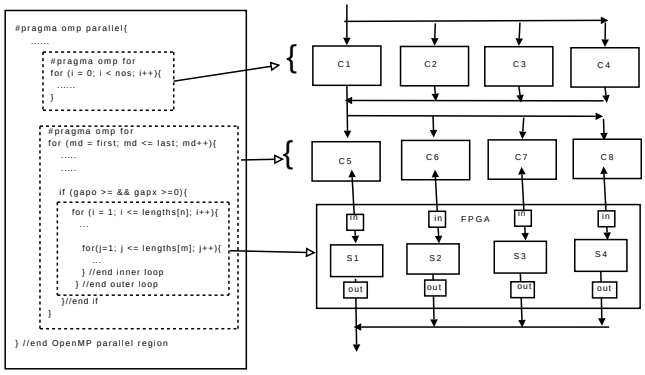


Fig. 2. OpenMP and FPGA Execution Model.

the optimizations applied by the compiler. We use feedback directed optimizations to hoist loops out of the compute intensive loopnest and apply short circuit optimizations to reduce branch misspredictions. The goals of our optimizations are to reduce the total execution cycles, increase the useful instruction per total instruction ratio and reduce total stall cycles. After achieving a low stall ratio on the sequential code, we proceed with

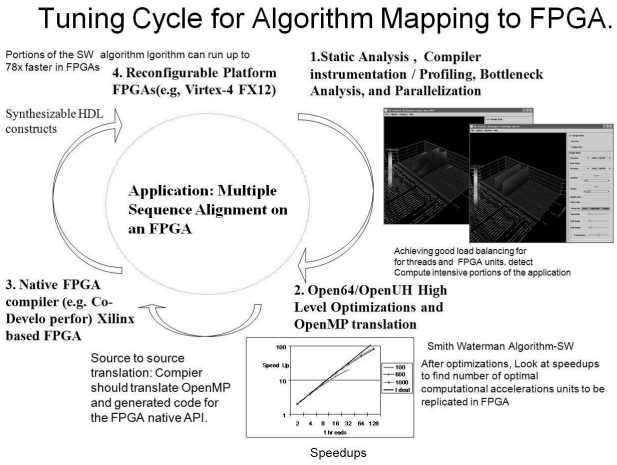


Fig. 3. An overview of the tuning cycle.

parallelization by instrumentation of the code with OpenMP. We perform privatization analysis to achieve good memory locality and check for load imbalances and find the

right scheduling discussed in one of the next sections. We perform experiments to see if the algorithm scales as shown in Fig. 3 on an SGI Altix 3700 distributed shared memory multi-processor machine, from NCSA, which consists of two SMP systems running on Linux. Each system has 512 Intel Itanium 2 processors. Our experimental platform has similar memory characteristics as some of the multicore/manycore processors (e.g. Opteron with HyperTransport and local memory access/core). Our analysis will help in mapping the critical portions of the loop on the FPGA using Impulse-C.

5 Tuning Methodology

To begin with, we use the Itanium 2 hardware performance counters to identify bottlenecks in the program and obtain a comprehensive performance-related characterization of the application [19]. We first calculate the parameters with respect to the unoptimized 'C' code and discovered that there are more than 40% of NOPs to be reduced for better performance. Similarly the branch misprediction and CPI could also be reduced. To analyze the program using the graphical analysis tool, Dragon, we compile the program with the OpenUH compiler. We use the software parallel programming paradigm, OpenMP, and add 'pragmas' to a sequential application code with the aim of parallelizing the code. OpenMP exploits shared memory by defining various types of parallel regions performing aggressive privatizations for data locality.

- **#pragma omp parallel:** Defines a parallel region to be run by multiple threads in parallel. All the directives work within the parallel regions defined by this directive
- **#pragma omp for:** Work-sharing construct identifying the iterative for-loop whose iterations should be run in parallel.
- **#pragma omp no wait:** Overrides the barrier implicit in a directive.

The following Fig. 4. is an example of the code generated by OpenMP.

```
msap{
  #pragma parallel region private(...) firstprivate(...)
  {
    #pragma omp for
      for(...)
        initialize array of locks

    #pragma omp for no wait
      for(...){
        for(...){
    #pragma omp for schedule(dynamic, chunk) nowait
      for(...){
        for(...){

    if condition then
       sequence 1
    else
       sequence 2

    #pragma omp critical

     //update to shared data
     omp_set_lock()
       //make updates
     omp_set_unlock()

  }
} /*end msap*/
```

Fig. 4. Example of pseudo code with OpenMP.

Dragon is invoked to instrument a program, to gather and display dynamic execution details, to find the areas where the loop is concentrated, to highlight the true, anti, output and procedure dependencies and to understand the control flow of the procedures in the source code. The flow graph and call graph are used to distinguish between the regions, branches, loops and the structure of the program respectively. The application is run several times. Feedback from the different runs are collected and merged into the call graph to show the frequencies with which the procedures are invoked. This information, plus the cycle count for each procedure,help in detecting the hot spots of the application. We use them to perform a manual optimization.

Table 1 shows the results of the bottleneck analysis for the SW local alignment 'C' code with an Itanium 2 target. In the unoptimized code, NOP operations account for 44% of the code. Realizing this portion of the code on the FPGA would therefore be inefficient. 75% of branches are incorrectly predicted, stalling the pipeline and causing wastage of resources. We found that the compiler could not apply high level loop optimizations due to loop dependencies, branches and lack of alias information. So we used the feedback capabilities of OpenUH to look at the branch frequencies and dependency graphs of the code. We manually hoisted the branches out of the loops to allow the compiler to perform more optimizations such as pipelining and eliminating dependencies. After optimizing the sequential code, we were able to parallelize the code using the OpenMP directives and performed aggressive optimizations for data locality through data privatization. The optimized version gives an improvement in CPI of 11.89%. The branch mispredictions are reduced by 21.33% thus once again resolving the dependency factors and helping in issuing multiple instructions per cycle. The nops_instruction are reduced by 45.45% and the count of useful instruction is increased by 10.09%.

Parameters	Unoptimized	Optimized
Useful Instruction Count	7.63E+9	8.40E+9
NOP Instructions	44%	24%
Branch Mispredictions	75%	59%
Cycles/instruction (CPI)	0.3178	0.28

Table 1. Bottleneck analysis of the SW 'C' for an Itanium 2 target.

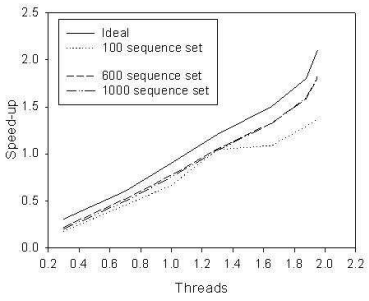
6 Scheduling

To improve the performance, we use the schedule clause to specify how the iterations of the loop should be allocated to the threads.

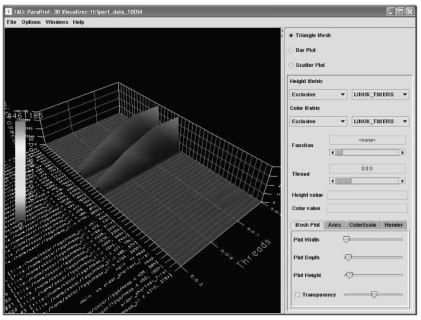
6.1 Static Scheduling

OpenMP can divide the main loop into a number of chunks using static scheduling. All the iterations are allocated to the threads. We initially perform a static scheduling

with 50 to 1000 data sequences and 1 to 128 threads. Each thread is given an equal number of iterations. If there are n iterations and T threads, each thread will get n/T iterations. The number of iterations each thread performs is decided at the start of the par-



(a) Static scheduling on different problem sequences.



(b) Static schedule is unbalanced with respect to the timings.

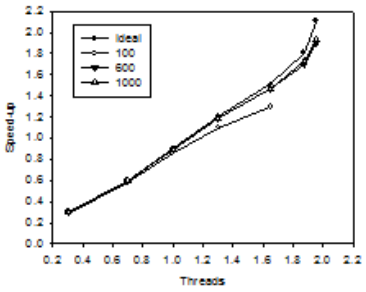
Fig. 5. Static Scheduling

allel loop. Static scheduling has advantages of reduced synchronization/communication overheads but there are drawbacks too. Due to uneven sized tasks distributed to the processing units, there are load-imbalances and idle processors. These cause wastage of resources. The speed-up is not ideal as shown in Fig. 5(a). The graph constructed on a logarithmic scale shows the different problem sequence set with threads ranging from 1 to 128. The speed-ups of the sequence sets do not reach the ideal speed-up giving rise to load imbalances. It is known, in the field of bio-informatics that the resultant matrix framed while achieving a protein database search is a triangular matrix. But due to the disadvantages discussed above, the expected triangular matrix is not framed and this is shown in Fig. 5(b).

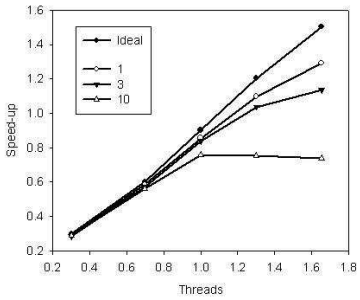
6.2 Dynamic Scheduling

Dynamic Scheduling gives the option of flexibility. Here, the number of iterations each thread performs is determined dynamically as the parallel loop is executed. We have to make sure that each processor spends an equal amount of working time on the tasks and the computational loads on the processor are well balanced avoiding any waiting time. Fig. 6(a) is the result of performing a dynamic scheduling on the protein sequence sets of 100, 600 and 1000. On comparing the results of Fig. 5(a), static scheduling, with Fig. 6(a) dynamic scheduling, the latter shows a clear and significant improvement in the speedup of 78x. But merely dividing the iterations equally among the processors does not result in a balanced load. Some processors should do more/fewer of the iterations that require less/more time per iteration. So the loop can be divided into chunks of h iterations or chunk size equaling to 1 or x% of the hth iterations. This would be

a better operation to perform compared to static scheduling since the chunks are taking a variable amount of time. Parallel applications have to be executed efficiently so that their workload is distributed among the various parallel processors. Choosing the correct chunk size would be a trade-off between running fewer chunks and achieving better thread balance. The larger the problem size, the less significant the synchronization overhead becomes. Fig. 6(b) shows that OpenMP is able to divide the loop into any number of chunks and launch any number of threads to process these chunks. The graph shows the results on the 600 sequence data set with 1, 3 and 10% of iterations.



(a) Dynamic Scheduling on different protein sequence set



(b) Speed-up with chunk sizes of 1, 3 and 10% of iterations.

Fig. 6. Dynamic Scheduling

As shown in the figure, chunk size 1 appears to be the ideal choice to do most of the computations. Since the data set is huge, we use upto 128 threads to perform the scheduling. We identify the tasks, design an algorithm to execute the tasks in parallel and manage the load balance among the processing units efficiently to achieve the effective speed-up using OpenMP. The efficient usage of chunks helps in balancing the processor loads as shown in Fig. 7. Though there may be increased overheads but this is outweighed by a good load balance. So the triangular matrix of the protein database search appears as desired.

To summarize the scheduling portion, we were not able to minimize the load imbalances statically. We detected this issue by using performance tools and the compiler instrumentation. Since the parallel loop were at a very coarse grain level (containing four nested loops), we tested different dynamic scheduling techniques by introducing different chunk sizes until we found a scheduling strategy that scaled. This gave us a significant speedup of upto 80% using 128 threads achieving excellent load imbalance as indicated in the screenshot.

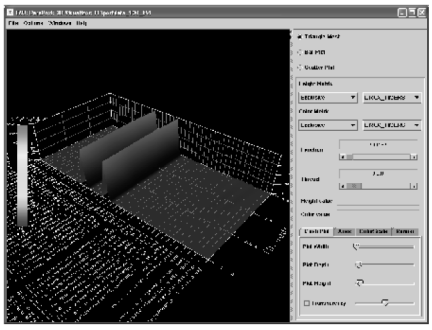


Fig. 7. Triangular Matrix Solver when using Dynamic Scheduling

7 OpenMP to FPGA Translation

After the optimizations, it will still be challenging to map either the entire OpenMP parallelized region of the code or portion of it on the FPGA. The elements on the hardware such as programmable gates, LUTs and flipflops need to be assembled into a programming model that high level programming languages can comprehend. To generate FPGA hardware from OpenMP 'C' programs via synthesizable hardware constructs, we use ImpulseC [20].

To allow the compilation and simulation of these hybrid applications, consisting of independently synchronized processes, the ImpulseC libraries include functions that define process interconnections (typically streams and/or signals) and emulate the behavior of multiple processes using threads. Streams represent a one-way communication channel between concurrent processes [21]. Related work has been done in [22] using Handel-C. We propose to overcome the difficulties in specifying the clock boundaries, clock timings and explicit statement-level parallelism used in Handel-C by introducing ImpulseC and synthesizable HDL constructs.

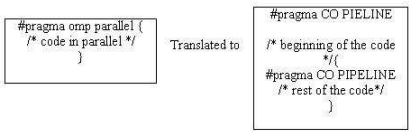


Fig. 8. Example of the translation of a OpenMP parallel construct.

The translation of an OpenMP parallel construct is as shown in Fig. 8. Thus the ImpulseC tools include a software-to-hardware compiler that can convert individual ImpulseC processes to functionally equivalent hardware descriptions, including the necessary process-to-process interface logic.

8 Conclusions and Future Work

In this paper, we have used the OpenUH compiler infrastructure and the associated graphical analysis tools for profiling and performing bottleneck analysis in an application. We have optimized and parallelized the code using efficient scheduling techniques. By running the different data sequence sets with varied chunk sizes and with up to 128 threads significant speed-ups close to the ideal have been achieved. These compiler techniques have helped us to reduce the complexity of the source code by removing the compute intensive mathematical operations. We have achieved CPI improvements of 11.89% with a reduction in branch mispredictions of 21.33%. NOP instructions have been reduced by 45.45%. Our multithreaded application achieves an almost linear speedup of upto 80% on dynamic scheduling with 128 threads on a 1000 sequence data set. Moving this optimized and parallelized code to a reconfigurable platform will give even better speed-up and performance. We are currently looking at translating OpenMP to Impulse-C, a tool for mainstream embedded system programmers seeking high performance through FPGA co-processing. We plan to address the lack of tools and techniques for turn-key mapping of algorithms to the hybrid CPU-FPGA systems by developing an OpenUH add-on module to perform this mapping automatically.

References

1. Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W., Cheung, P.: Reconfigurable computing: architectures and design methods. Computers and Digital Techniques, IEEE Proceedings- **152**(2) (2005) 193–207
2. Sullivan, C., Wilson, A., Chappell, S.: Using c based logic synthesis to bridge the productivity gap. In: ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation, Piscataway, NJ, USA, IEEE Press (2004) 349–354
3. Oliver, T., Schmidt, B., Maskell, D., Nathan, D., Clemens, R.: High-speed multiple sequence alignment on a reconfigurable platform. Int. J. Bioinformatics Research and Applications **2**(4) (2006) 394–406
4. Oliver, T., Schmidt, B., Nathan, D., Clemens, R., Maskell, D.: Using reconfigurable hardware to accelerate multiple sequence alignment with clustalw. Bioinformatics **21**(16) (2005) 3431–3432
5. Aung, Y., Maskell, D., Oliver, T., Schmidt, B., Bong, W.: C-based design methodology for fpga implementation of clustalw msa. In: Pattern Recognition in Bioinformatics: Second IAPR International Workshop, PRIB, Berlin, Germany, Springer-Verlag (2007) 11–18
6. Zomaya, A., ed.: Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies. 1 edn. Wiley Series on Parallel and Distributed Computing. Wiley Interscience (2006)
7. Feng, D.F., Doolittle, R.: Progressive sequence alignment as a prerequisite to a correct phylogenetic trees. Journal of Molecular Evolution **25** (1987) 351–360
8. Thompson, J., Higgins, D., Gibson, T.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucl. Acids Res. **22** (1994) 4673–4680
9. Ebedes, J., Datta, A.: Multiple sequence alignment in parallel on a workstation cluster. Bioinformatics **20**(7) (2004) 1193–1195
10. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Molec. Biol. **147** (1981) 195–197

11. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: Openuh: An optimizing, portable openmp compiler. In: 12th Workshop on Compilers for Parallel Computers. (2006)
12. Open64: http://open64.sourceforge.net (2005)
13. Triolet, R., Irigoin, F., Feautrier, P.: Direct parallelization of call statements. In: SIGPLAN Symp. on Compiler Construction. (1986) 176–185
14. Hernandez, O., Liao, C., Chapman, B.: A tool to display array access patterns in OpenMP programs. In: PARA'04 Workshop On State-Of-The-Art In Scientific Computing, Springer (2004)
15. Chapman, B., Hernandez, O., Huang, L., Weng, T., Liu, Z., Adhianto, L., Wen, Y.: Dragon: An open64-based interactive program analysis tool for large applications. In: 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). (2003)
16. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid mpi/openm p applications. Journal of Systems Architecture: the EUROMICRO Journal **49**(10-11) (2003) 421–439
17. Shende, S., Malony, A., Cuny, J., Beckman, P., Karmesin, S., Lindlan, K.: Portable profiling and tracing for parallel, scientific applications using c++. In: SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, New York, NY, USA, ACM Press (1998) 134–145
18. Kufrin, R.: Perfsuite: An accessible, open source performance analysis environment for linux. In: 6th International Conference on Linux Clusters: The HPC Revolution 2005. (2005)
19. Jarp, S.: A methodology for using the itanium-2 performance counters for bottl eneck analysis. Technical report, HP Labs (2002)
20. ImpulseC: http://www.impulsec.com/ (2007)
21. Pellerin, D., Thibault, S.: Practical FPGA Programming in C. Prentice Hall Professional Technical Reference (2005)
22. Leow, Y., Ng, C., Wong, W.: Generating hardware from openmp programs. In: IEEE Int. Conf on Field Programmable Technology, FPT. (2006) 73–80