Parallelizing program code automatically through HPCC

[1]Mariyamath Rifaina, [2]Mustafa Basthikodi, Ahmed Rimaz Faizabadi
[1]PG student, [2]Professor, BIT, Mangalore [3]Associate Professor, BIT, Mangalore

ABSTRACT

Parallel computing, a form of computation which allows many instructions in a program to run concurrently, in parallel. In fact to accomplish this, a program has to be split into independent parts so that each processor can execute its part of program simultaneously with the other processor. Algorithmic Species, an algorithm classification which captures low-level algorithmic details and represents them with the use of five easy way to understand array access patterns. To inscribe the challenge of parallel programming, we propose Bones. Bones is one of a source-to-source compiler that deploy on algorithmic skeletons and, a new algorithm classification. To evaluate the applicability of algorithmic species and to validate A-Darwin and Bones, we test against the HPCC benchmark. In this work, the existing species for all algorithmic classification are analyzed and some of the computationally intensive programs from HPCC benchmark have been tested using existing tools such as Bones compiler and the A-Darwin, an automatic species extraction tool. The access patterns are created for various algorithmic kernels by running against A-Darwin and the analysis is done for various code segments.

Keywords- bones compiler, *algorithmic species, A-Darwin tool, hpcc benchmark.*

1. INTRODUCTION

Advantage of multi-core processor over single core is that former can either use both its cores to accomplish a single task or it can

span the threads which divided tasks between both of its cores, so that it take twice the time it would take to execute the task than in the latter. An example would be watching movie on windows media player while your dual-core processor is already running a background virus check. Multi-core is a shared memory processor. All cores share the same memory. In multi-core environment the sequential computing paradigm is not good and inefficient, while the usual parallel computing may be suitable. The factor lead the design of parallel algorithm for the multi-core system is the performance.

2. LITERATURE SURVEY

Parallel computing is type of computation where numerous calculations are being carried out continously, operating on principle that even a large problems can be often divided into smaller ones, which then solved at same time. Parallel computing can be attained on a single computer with multiple processors, number of individual computers connected by network or combination of two. Parallel performance measurement parameters and parallel benchmarks are used to measure the performance of a parallel computing system[10]. There are several forms of parallel computing: bit, instruction, data, and task–level parallelism. As power utilization and inevitably heat generation by computers has become concern in recent years, parallel computing has become a dominant paradigm in the computer architecture, mainly in form of multi-core processors. In parallel computing, a computational task is typically broken down in several, often many, very similar subtasks that can be processed independently and whose results are combined afterwards, upon completion. . Data parallelism is a form of parallelization of computing over multiple processors in the parallel computing environments. Data parallelism aims on distributing data across different parallel computing nodes. Task parallelism focuses on distributing tasks apparently performed by processes or the threads across different parallel computing nodes.

OpenMP is a standard API for writing shared-memory parallel applications in C++, C and Fortran. OpenMP employ on a Fork and Join model of the parallel execution. All the OpenMP programs begin as single process which is called master thread. At this point, master thread ‘forks’ into number of parallel worker threads. Instructions in parallel region are then executed by this team of worker threads. At the end of parallel region, threads synchronize and join to befall the single master thread. Most OpenMP implementations have translated “OpenMP threads” to “kernel threads”, which are internally mapped to system-scope POSIX threads or native OS threads. The OpenMP has the support of multithreading. Programmers and compilers can greatly benefit from a structured classification of program code[6]. Similarly, parallelizing compilers and source-to-source compilers can take threading and optimization decisions based on the same classification. Algorithmic species, classification of affine loop nests based on polyhedral model and are targeted for both the automatic and manual use. Discrete classes capture information such as structure of parallelism and data reuse. To demonstrate the use of algorithmic species, 115 classes in a benchmark set are identified.

Algorithmic species, an algorithm classification deploy on access patterns of arrays in the loop nests. Algorithmic species are used for skeleton-based source-to-source compiler in order to achieve the performance across different architectures. Algorithmic species, an algorithm classification based on access patterns of the arrays in loop nests. The classification is designed to fulfill the following goals:

1)The programmers can reason relevant to their program code by the means of algorithm classes,

2)The performance models can also use class information to estimate performance.

3) compilers can be designed based on the classification .
To achieve this, the following requirements for classes are set: automatically extracted, intuitive, formally defined, complete and fine-grained. We use an algorithm classification to drive a source-to-source compiler build on algorithmic-skeletons. Since the algorithmic species theory is based on the polyhedral model, completeness and automatic extraction can at best be achieved only for code that is represented as static affine loop nests. Algorithmic species, a classification which process low-level algorithm details from individual loops or loop nests and their bodies. Key to the algorithmic species approach is that every array, accessed in the classified loop nest, is assigned with one of the five access patterns[3]. These five access patterns are element, chunk, neighborhood, full, and shared. The combination of access patterns, of the input and output arrays of the loop nest, then form the species. This modular approach validates us to form unlimited amount of species with the use of only five access patterns. The five different patterns which are part of the classification's vocabulary are defined as follows:
element: The element access pattern represents a single access of a data structure's contents at each coordinate in the structure. Accesses to individual elements of the data structure are assumed to be independent of each other. The amount of the parallelism for this pattern is hence equal to the size of data structure.
Tile(T): When accessing data in a tile pattern, a structure of dimensions T is accessed simultaneously, but independent of other tiles in the same data structure. There is no data re-use, all contents are accessed once. The parallelism present inside this pattern is equal to the size of data structure divided by tile size.
Neighborhood(N): The neighborhood access pattern is similar to the element pattern, but enables overlap. The pattern represents the re-use of a data structure's contents for a neighborhood of dimensions N centered around each coordinate of a structure. This data access pattern is to be only used as input pattern.
Shared: Shared data access pattern is an output pattern. It is applied when multiple accesses prevail to contents at a single coordinate in a data structure. The pattern even enables reading from the output at the same coordinate.
Full: The full access pattern approaches the complete data structure. This implies that no parallelism is available. This pattern is equal to the use of the tile pattern with a tile size equal to the data structure's size[s7].

A-Darwin is a tool that automatically extract algorithmic species. The new tool is largely equal to ASET in terms of functionality, but is different internally. This tool is based on CAST, a C99 parser which allows analysis on an abstract syntax tree (AST). From the AST, the tool extracts the array references and constructs a 5 or 6-tuple for each loop nest, then merging is applied and the species are extracted. At last, the species are inserted as pragmas in the original source code. To perform the dependence tests in A-Darwin, combination of the GCD and Banerjee tests was made. Together, these tests are reserved, so it might not find all the species.

The automatic deployment of algorithmic species is done using the BONES source-to-source compiler Skeletons which added as new classes are recognized, creating a flexible compiler. Anyhow, in the case of BONES, algorithmic species information is used to automatically select a skeleton for a given algorithm. This makes BONES, combined with an automatic species extraction tool such as ASET, a fully automatic source-to source compiler[1].

Recent advances in multi-core and many-core processors require programmers to exploit an increasing amount of parallelism from their applications. To address these challenge of parallel programming, we come up with Bones which is a source-to-source compiler based on algorithmic skeletons and a new algorithm classification. Bones is open source and is written using Ruby language. The compiler is depending on the C-parser CAST, which is used to parse the input code into an abstract syntax. Then the compiler holds C-code annotated with class information as input and it produces parallelized target-code. A high-level overview of the Bones source-to-source compiler is found in figure-1. The compiler performs 4 steps, which are explained below. The step numbers correspond to numbers given in the blue boxes in figure 2.1.

- STEP1: The pre-processor extracts the user algorithmic species information from the source code. Code in between the #pragma species kernel and #pragma species endkernel statements along with the class name is stored as an 'algorithm'. Multiple such algorithms can exist within one piece of source code.
- STEP 2: The AST of the kernel code and the AST of the original code are analyzed to obtain information on the used variables, e.g. are they dynamically or statically allocated, are they private to the kernel, are they used as input or output. The information on the variables is used later on to instantiate and populate the skeletons.
- STEP 3: The kernel AST is transformed according to a per skeleton unique transformation list. These transformations are fairly basic, e.g. renaming the variables, removal of the outer loop. The transformation list contains integer codes referring to the different available transformations.
- STEP 4. The skeletons are instantiated and populated. A common library supplies parameterized code which is equal independent of the algorithm class. This includes memory allocation and memory transfer mechanisms for example. The skeleton library supplies both parameterized host code and kernel code, unique to an algorithm class.

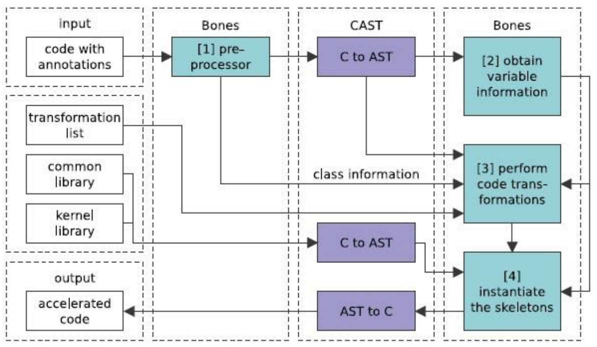


Figure 2.1: High-level overview of the Bones source-to-source compiler.

Benchmarking is the technique of using crafted programs in order to attach quantifiable performance metrics to targeted computer subsystems.It is also an act of measuring and evaluating computational performance, networking protocols, devices and networks, under reference conditions, relative to a reference evaluation. Benchmarks can be classified as Micro benchmarks and Macro benchmarks. Micro benchmarks tend to be synthetic kernels. Micro benchmarks measure a specific aspect of computer system like CPU speed, Memory speed, I/O speed, Operating system performance and Networking. Macro benchmark measures the performance of a computer system as a whole. Anyhow, macro benchmarks do not disclose why a system performs well or badly. Examples: Rodinia, NAS, PolyBench, HPCC. Software benchmarks run against database management systems or compilers. Benchmarks are incredibly important part of computer architecture research. As computer systems modified over time, older benchmarks become less useful, influencing the creation of new benchmarks applicable to the current systems and applications of interest. Comparability should be a basic property of any benchmark; comparability means that two independently executed benchmarks can be meaningfully compared to each other. One of the factors influencing the comparability is repeatability: running an identical benchmark on an identical solution at different moments in time should result in a (close to) identical result[9]. Benchmarks can be full-fledged applications or just kernels. Common benchmarks fall into one of the category:

- Synthetic: These code segments are small programs especially constructed for benchmarking purposes and do not represent any real computation but exercise various basic machine functions.
- Kernel: These code segments are typically extracted from larger program and represent portions of execution times in real application.
- Application: These code segments represents complete application programs solving well-defined scientific problems and evaluates machines in a way that kernels and code-fragments cannot.
- Hybrid: As it is very difficult to describe the exact performance of a system with just one type of benchmark, hybrid benchmarks has been proposed using some or all of the code fragments in their benchmarks[5].

HPC Challenge is a suite of tests that investigate the performance of HPC architectures using kernels with memory access patterns. The first aspect of HPCC is benchmarking the system with different combinations of high and low temporal and spatial locality of the memory access. Other aspects are measuring basic parameters like achievable computational performance the bandwidth of the memory access and latency and bandwidth of the inter-process communication based on ping-pong benchmarks and on parallel effective bandwidth benchmarks. The collection of tests includes tests on a single processor (local) and tests over the complete system (global). In specific, to characterize the architecture of the system we consider three testing scenarios:

- Local –were only a single processor is performing computations.
- Embarrassingly Parallel –were each processor in the entire system is performing computations but they do not communicate with each other explicitly.
- Global – were all processors in the system are performing computations and they explicitly communicate with each other.

HPC Challenge is a benchmark suite that evaluates a range memory access patterns. The HPC Challenge Benchmark brings togther several benchmarks to test a number of disinct attributes of the performance of high-performance computer (HPC) systems..The performance of complex applications on the HPC systems can rely on a variety of independent performance attributes of the hardware. The benchmark presently consists of 7 tests with the modes of operation indicated for each:

- HPL (High Performance LINPACK) – It measures performance of a solver for a dense system of linear equations(global). This is the widely used implementation of the Linpack TPP benchmark. It measures the sustained floating point rate of execution for solving a linear systemof equations.
- DGEMM – It measures performance for matrix-matrix multiplication (single, star). It measures the floating point rate of execution of double precision real matrix-matrix multiplication.
- STREAM - It measures sustained memory bandwidth to/from memory (single, star).
- PTRANS(Parallel matrix transpose) – It measures the rate at which the system can transpose a large array (global).
- RandomAccess - It measures the rate of 64-bit updates to randomly selected elements of a large table (single, star, global). It Measures the rate of integer updates to random locations in large global memory array.
- FFT(Fast Fourier Transformations)– It performs a Fast Fourier Transform on a large one-dimensional vector using the generalized Cooley-Turkey algorithm (single, star, global).
- Communication Bandwidth and Latency – It is MPI-centric performance measurements based on the b_eff bandwidth/latency benchmark. It is a set of tests to measure latency and bandwidth of a number of simultaneous communication patterns[4].

3. DESIGN

3.1 Overview of Auto Parallelization approach

Algorithmic species includes a tool to automatically extract species from static affine loops (A-Darwin) and a source-to- source compiler based on skeletons (Bones). Both A-Darwin and Bones are open-source and are available freely. The compiler Bones is programmed in Ruby language and uses the C-to-AST module CAST. Algorithm classifications exist in different forms, with different names and for different purposes. Many variations of algorithm classifications have been introduced as part of the work on the algorithmic skeletons technique, which is used for code generation purposes Algorithmic skeletons is a technique that uses parameterizable program code, called as skeletons. A single skeleton can be viewed as template code for a particular class of computations on a particular processor. If no skeleton implementation was present for the specific class or processor, it could manually be added.

In this work we represent a model to automatically process readable parallel code for parallel architectures .We base this technique on 'algorithmic species' an algorithm classification of program code based on the polyhedral model .Algorithmic species encapsulate information such as data re-use and memory access patterns Algorithmic species forms the backbone (see Fig. 3.1), which contains a tool that automatically extract species from static affine loops (A-Darwin) and a source-to-source compiler based on skeletons (Bones)[7]. In this two-step approach, the first one is to extract related information from source code. Then the compiler takes C code annotated with species information as an input. The algorithmic-species that extracted by a pre-processor are used straightly to decide which skeleton to use. Furthermore, they are used to enable/disable additional transformations and optimizations. These skeletons must be constructed in such a way that they are proper for all algorithms belonging to a given species. This information is encapsulated in the form of algorithmic species[2] .

Bones uses introduced algorithm classification as a basis for the skeleton library. The bones compiler improves upon existing C-to-CUDA tools by three main factors:

1. Bones generates readable and editable code.
2. Bones is based on a well-defined algorithm classification.
3. Similar to PGI Accelerator, only a minimum effort is required (two directives) to transform the original code into code suitable for the compiler.

Figure 3.1 Shows the overview of the complete auto-parallelization approach including algorithmic species, A-Darwin, bones compiler.

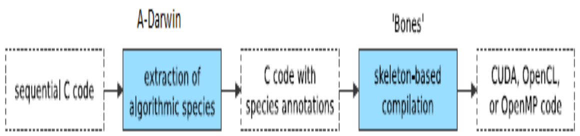

A-Darwin
'Bones'
sequential C code
extraction of algorithmic species
C code with species annotations
skeleton-based compilation
CUDA, OpenCL, or OpenMP code

Figure 3.1:Overview of complete auto-parallelization approach.

- The first step is to collect needed information from the source code. This information is encapsulated in the form of algorithmic Species, an algorithm classification based on polyhedral. A preprocessor extracts the user supplied class information from the source code.
- The second step is to, to avoid mistakes, and automatically process parallel code, the species are obtained automatically using A-Darwin, an algorithmic species extraction tool.
- The third step, the AST of the kernel code and the AST of the original code are analyzed to obtain information. The kernel AST is transformed according to a per skeleton unique transformation list. The transformation list contains integer codes referring to the different available transformations.
- The fourth step, the compiler takes C-code annotated with species information as an input.
- The last step, Algorithmic-species map one-to-one to skeletons. To select a suitable skeleton, algorithmic species information are used.

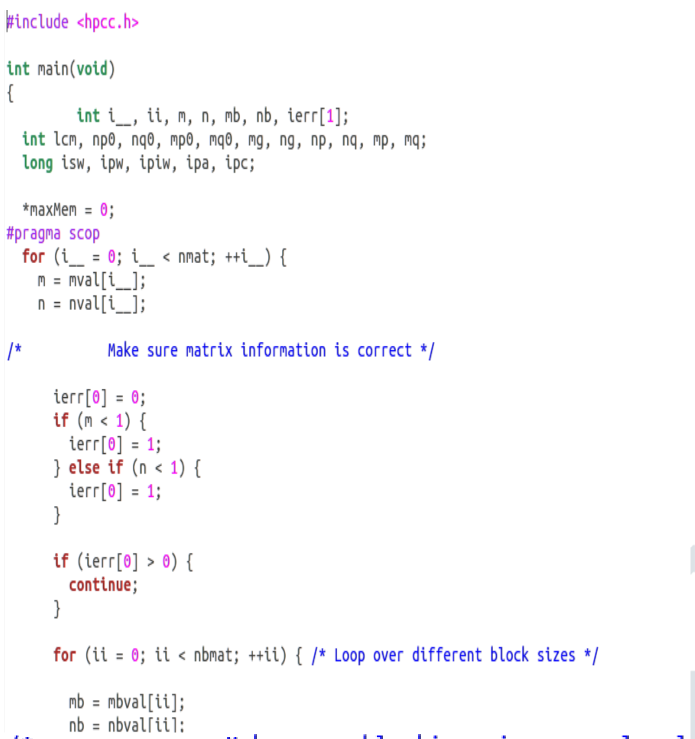
4. IMPLEMENTATION

To evaluate the how much the algorithmic species is applicable and to validate A-Darwin tool and Bones, we test against the HPCC benchmark suite. The HPCC suite contains 7 test, For our evaluation, we therefore include PTRANS, FFT, RandomAccess, DGEMM, HPL. By integrating algorithmic-species with the skeleton-based compiler, we have developed a distinct view to automatically creating code for parallel targets

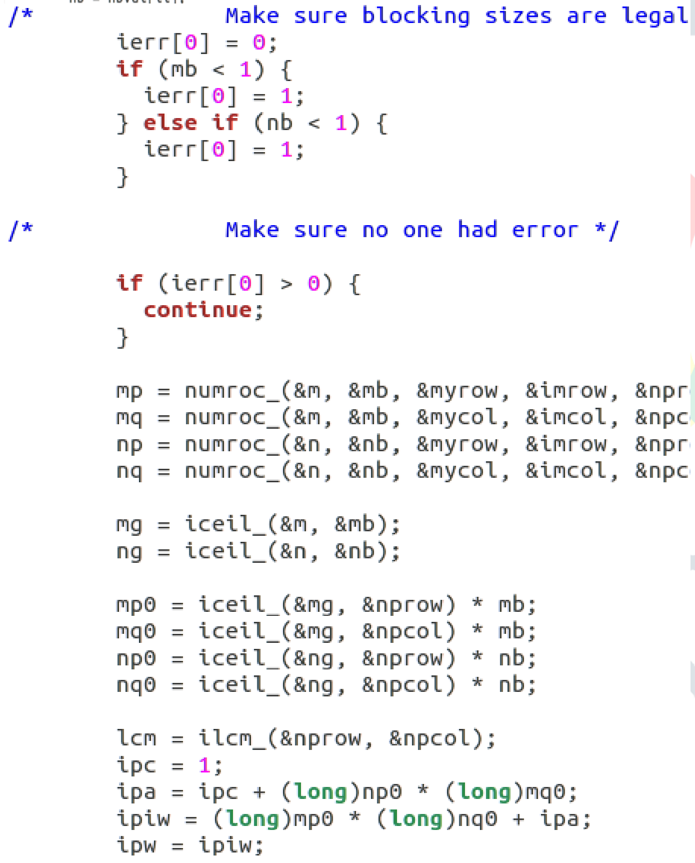
Skeleton-based compilation has several uses. Firstly, the compilation needs only basic transformations that can perform at the abstract syntax-tree level, eliminating the requirments for intermediate representations which continously lose code-structure and the variable naming. This allows the compiler to generate a code which is readable, allowing chances for further fine-tuning and manual optimization. Furthermore, skeletons by themselves can be formatted to contain structure and code comments. Secondly, skeleton-based compilation benefits from the flexibility of improving the compiler or extending to other targets: simply adjust or write the appropriate skeletons. Finally, several optimizations applied within skeletons in Bones can't be applied as code transformations on original-code. Because of the integration of algorithmic species, Bones is the first skeleton-based compiler that can be used in a fully-automatic tool-flow. This eliminates the requirements of existent skeleton-based approaches to manually find a skeleton and modify the code such that the skeleton can be used. Moreover, an algorithmic species gives a structured, clear and formally defined way of utilizing a skeletons, that can be useful in cases where manual-classification is inevitable.

Bones and A-Darwin ahead with required gems are to be installed in quad core system for the experimentation. Bones is the source-to source compiler that is designed based on algorithmic skeletons and the algorithmic species. The compiler takes as input C-code as shown in Fig 4.1 and generates parallel code as output. Target processors include NVIDIA GPUs for CUDA, AMD GPUs for OpenCL and CPUs for OpenCL and OpenMP. HPCC Benchmark consists of 7 tests and it is open source. HPCC Benchmark is given as the input to Bones compiler and patterns are being produced for each kernels. The produced patterns may be element, shared, chunk(tile), neighborhood or full.

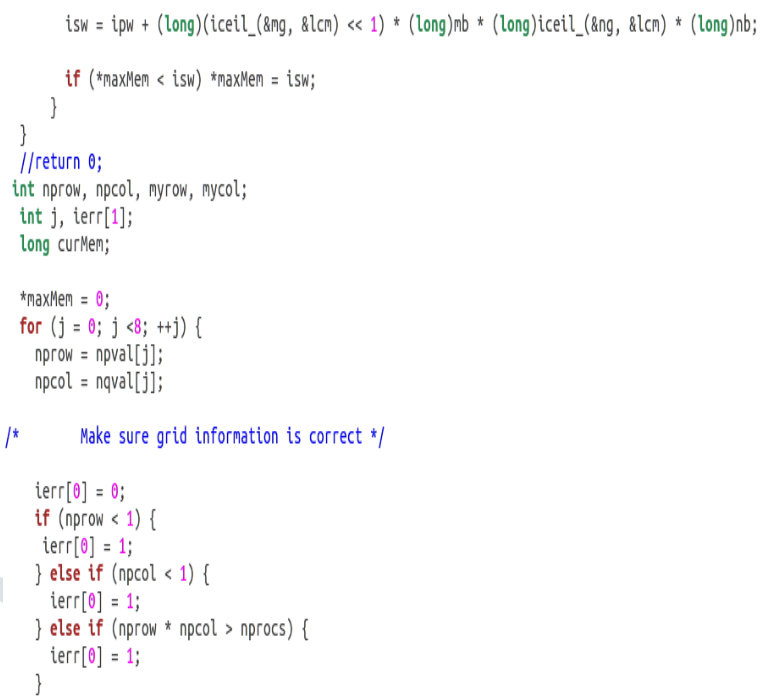
Let us take an example of HPCC,PTRANS benchmarks consisting of 3 kernels. The code is illustrated below:



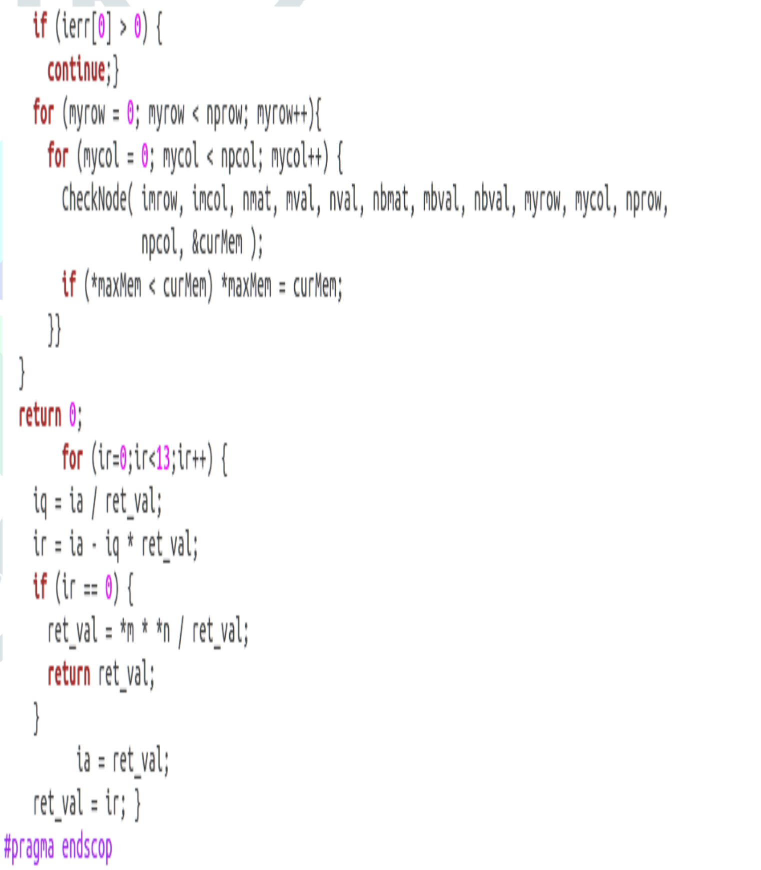
```
#include <hpcc.h>

int main(void)
{
        int i__, ii, m, n, mb, nb, ierr[1];
  int lcm, np0, nq0, mp0, mq0, mg, ng, np, nq, mp, mq;
  long isw, ipw, ipiw, ipa, ipc;

  *maxMem = 0;
#pragma scop
  for (i__ = 0; i__ < nmat; ++i__) {
    m = mval[i__];
    n = nval[i__];

/*          Make sure matrix information is correct */

      ierr[0] = 0;
      if (m < 1) {
        ierr[0] = 1;
      } else if (n < 1) {
        ierr[0] = 1;
      }

      if (ierr[0] > 0) {
        continue;
      }

      for (ii = 0; ii < nbmat; ++ii) { /* Loop over different block sizes */

        mb = mbval[ii];
        nb = nbval[ii];
```



```
/*              Make sure blocking sizes are legal
        ierr[0] = 0;
        if (mb < 1) {
          ierr[0] = 1;
        } else if (nb < 1) {
          ierr[0] = 1;
        }

/*              Make sure no one had error */

        if (ierr[0] > 0) {
          continue;
        }

        mp = numroc_(&m, &mb, &myrow, &imrow, &npr
        mq = numroc_(&m, &mb, &mycol, &imcol, &npc
        np = numroc_(&n, &nb, &myrow, &imrow, &npr
        nq = numroc_(&n, &nb, &mycol, &imcol, &npc

        mg = iceil_(&m, &mb);
        ng = iceil_(&n, &nb);

        mp0 = iceil_(&mg, &nprow) * mb;
        mq0 = iceil_(&mg, &npcol) * mb;
        np0 = iceil_(&ng, &nprow) * nb;
        nq0 = iceil_(&ng, &npcol) * nb;

        lcm = ilcm_(&nprow, &npcol);
        ipc = 1;
        ipa = ipc + (long)np0 * (long)mq0;
        ipiw = (long)mp0 * (long)nq0 + ipa;
        ipw = ipiw;
```



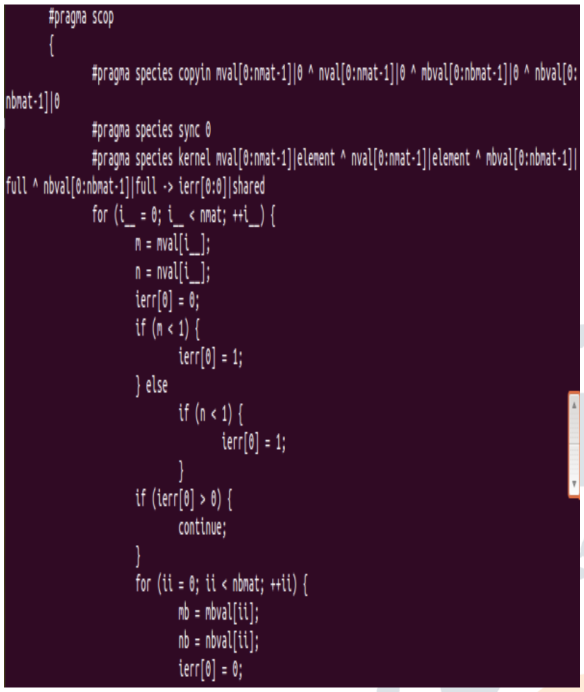
```
        isw = ipw + (long)(iceil_(&mg, &lcm) << 1) * (long)mb * (long)iceil_(&ng, &lcm) * (long)nb;

        if (*maxMem < isw) *maxMem = isw;
      }
  }
  //return 0;
 int nprow, npcol, myrow, mycol;
  int j, ierr[1];
  long curMem;

  *maxMem = 0;
  for (j = 0; j <8; ++j) {
    nprow = npval[j];
    npcol = nqval[j];

/*        Make sure grid information is correct */

    ierr[0] = 0;
    if (nprow < 1) {
     ierr[0] = 1;
    } else if (npcol < 1) {
      ierr[0] = 1;
    } else if (nprow * npcol > nprocs) {
      ierr[0] = 1;
    }
```



```
    if (ierr[0] > 0) {
      continue;}
    for (myrow = 0; myrow < nprow; myrow++){
      for (mycol = 0; mycol < npcol; mycol++) {
        CheckNode( imrow, imcol, nmat, mval, nval, nbmat, mbval, nbval, myrow, mycol, nprow,
                   npcol, &curMem );
        if (*maxMem < curMem) *maxMem = curMem;
      }}
  }
  return 0;
        for (ir=0;ir<13;ir++) {
    iq = ia / ret_val;
    ir = ia - iq * ret_val;
    if (ir == 0) {
      ret_val = *m * *n / ret_val;
      return ret_val;
    }
        ia = ret_val;
    ret_val = ir; }
#pragma endscop
```

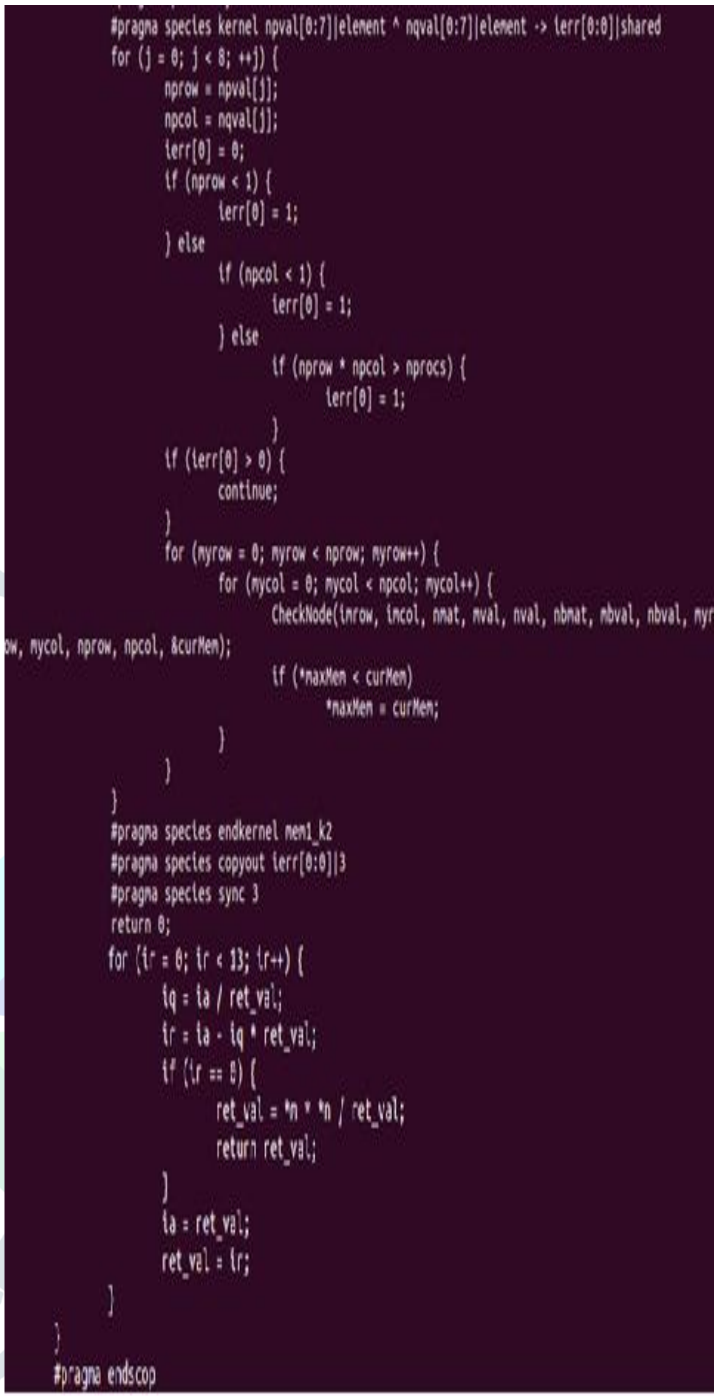
The Bones compiler is based on CAST C-parser , which is used to parse input source code into the AST (Abstract Syntax Tree) and to generate desired code from the transformed abstract syntax tree. A-Darwin (short for `automatic Darwin`) is automatic extraction tool, which is based on CAST, a C99 parser which allows analysis on AST. From the AST, the tool extracts the array references and constructs a 5 or 6-tuple for each loop nest. Following, merging is applied and species are extracted. Finally, the

species are inserted as pragmas in the original program code. The code segments of various algorithm classes are been executed to analyze the output of A-Darwin. The patterns generated by the Bones compiler on executing one of the HPCC benchmarks is as shown in Fig 4.1 and 4.2.The syntax of A-Darwin tool is given below:
Adarwin-
a../examples/benchmarks/hpcctst/PTRANS/pmem.c



```
    #pragma scop
    {
        #pragma species copyin mval[0:nmat-1]|0 ^ nval[0:nmat-1]|0 ^ mbval[0:nbmat-1]|0 ^ nbval[0:
nbmat-1]|0
        #pragma species sync 0
        #pragma species kernel mval[0:nmat-1]|element ^ nval[0:nmat-1]|element ^ mbval[0:nbmat-1]|
full ^ nbval[0:nbmat-1]|full -> ierr[0:0]|shared
        for (i__ = 0; i__ < nmat; ++i__) {
            m = mval[i__];
            n = nval[i__];
            ierr[0] = 0;
            if (m < 1) {
                ierr[0] = 1;
            } else
                if (n < 1) {
                    ierr[0] = 1;
                }
            if (ierr[0] > 0) {
                continue;
            }
            for (ii = 0; ii < nbmat; ++ii) {
                mb = mbval[ii];
                nb = nbval[ii];
                ierr[0] = 0;
```

Figure 4.1: Shows the pattern generated for kernel 1 after parallelizing the code.



```
            #pragma species kernel npval[0:7]|element ^ nqval[0:7]|element -> ierr[0:0]|shared
            for (j = 0; j < 8; ++j) {
                nprow = npval[j];
                npcol = nqval[j];
                ierr[0] = 0;
                if (nprow < 1) {
                    ierr[0] = 1;
                } else
                    if (npcol < 1) {
                        ierr[0] = 1;
                    } else
                        if (nprow * npcol > nprocs) {
                            ierr[0] = 1;
                        }
                if (ierr[0] > 0) {
                    continue;
                }
                for (myrow = 0; myrow < nprow; myrow++) {
                    for (mycol = 0; mycol < npcol; mycol++) {
                        CheckNode(imrow, imcol, nmat, mval, nval, nbmat, mbval, nbval, myr
ow, mycol, nprow, npcol, &curMem);
                        if (*maxMem < curMem)
                            *maxMem = curMem;
                    }
                }
            }
            #pragma species endkernel mem1_k2
            #pragma species copyout ierr[0:0]|3
            #pragma species sync 3
            return 0;
            for (ir = 0; ir < 13; ir++) {
                iq = ia / ret_val;
                ir = ia - iq * ret_val;
                if (ir == 0) {
                    ret_val = *n * *n / ret_val;
                    return ret_val;
                }
                ia = ret_val;
                ret_val = ir;
            }
        }
        #pragma endscop
```

Figure 4.2: Shows the pattern generated for kernel 2 after parallelizing the code.

5. EXPERIMENTATION AND RESULT ANALYSIS

HPPCC BENCHMARK				
Modules	**Number of Kernels**	**Executed**	**Failed**	**Build-in function/function calls/operators**
donecpu.c	1	1	0	rand()

dtstdgemm.c	2	2	0	rand()
				fabs(a0[k])
				Mmin(nx,m-i)

sonecpu.c	1	1	0	rand()

dstream.c	9	9	0	fabs(a[j]/aj-1.0)
				fabs(b[j]/bj-1.0)
				fabs(c[j]/cj-1.0)
				MPI_Wtime()
				Mmin(minDelta,Mmax(Delta,0))
				fabs(a[]-aj)

fbcrand.c	5	5	0	expm2(a-p3i,p3i)
				ddmuldd(d2,d3,dd1)
				dddiv(dd1,p3i,dd2,&d1)

ftstfft.c	3	2	1	Sqrt(tmp1*tmp1+tmp2*tmp2)
				HPCC_factor235(n,f)
				c_re(in[i])
				c_im(out[i])

fwrapfftw.c	1	1	0	c_assign(out[i],in[i])

pmem.c	3	3	0	ilcm_(&nprow,&npcol)
				numroc_(&m,&b,&myrow,&mycol)
				iceil_(&m,&mb)

psclapack.c	3	3	0	dcputime00()
				dwalltime00()

rbuckets.c	2	2	0	HPCC_PoolReturnObj(Update_Pool,ptr2);
rutility.c	4	3	1	Left shift << Right shift >>
hmatgen.c	1	1	0	HPL_rand()

hpdmatgen.c	1	1	0	HPL_rand()
				HPL_jumpit(ia4,ic4,ib3,iran4)

Table 5.1: Shows the number of kernels in each algorithmic class and its state of execution.

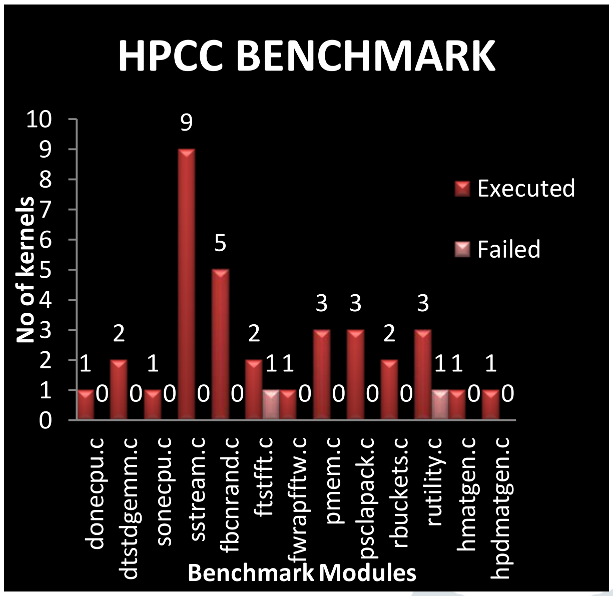


Figure 5.1: Graphical representation of HPCC Benchmark over Bones compiler.

The Table 5.1 gives the no of kernels executed and failed during the execution of each module of the benchmark and its function calls, build-in functions and operators. The figure 5.1 gives the graphical representation of the number of kernels executed and failed. Hence this table gives the performance of bones compiler over HPCC benchmark. The executed kernels produce different access patterns according to the variable, function call, build-in function and the calculation used in a particular kernel.

6. CONCLUSION AND FUTURE WORK

Parallel computing has played a vital role in improving the performance of applications. With A-Darwin, we are automatically be to extract species from affine loop nests. The species which are used within the skeleton-based compiler Bones to select a skeleton. BONES is able to produce a readable code, provides flexibility for new targets and optimizations, and also generates efficient-code. In this work we presented 'Algorithmic Species', an algorithm classification which captures low-level algorithmic details and represents them with the use of five easy to understand array access patterns. The HPCC benchmark is given as the input to bones compiler which then produced different access pattern. . In this work, we have analyzed the existing access patterns generated for the kernels and listed out many of the kernels. We have mentioned the number of kernels executed and number of kernels failed in HPCC benchmark suite. We have also listed the constants, built in functions and mathematical functions which come under the particular kernel. In this work, we have shown potential of Bones compiler for alimited number of examples. Future work on algorithmic species can expand the classification to more types of algorithms with a particular focus on irregular algorithms and remove limitations to the theory, and implementing a debug mode to check for correctness.

REFERENCES

[1] "*The Bones Source-to-Source Compiler Manual*", Cedric Nugteren, August 7, 2012.

[2] "*Algorithmic Species Revisited: A Program Code Classification Based on Array References*", Cedric Nugteren, Rosilde Corvino, and Henk Corporaal Eindhoven University of Technology, The Netherlands.

[3] "*Algorithmic Species: Classifying Program Code for Parallel Computing*", P.J.J.M. Custers, Electronic Systems group, Eindhoven University of Technology.

[4] "*Introduction to the HPC Challenge Benchmark Suite*", Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi11.

[5] "*Benchmarking Parallel Processing Systems A Survey*", Hannes Pfneiszl, Gabriele Kotsis, Technical Report No. TR-96103,Institute of Applied Computer Science and Information Systems, University of Vienna.

[6] "*An Introduction to Parallel Programming with OpenMP*", Alina Kiessling, A Pedagogical Seminar April 2009.

[7] "*A Modular and Parameterisable Classification of Algorithms*", C. Nugteren and H. Corporaal.

[8] "*Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification*", Cedric Nugteren, Pieter Custers, and Henk Corporaal.

[9] "*Benchmarking Computers and Computer Networks*", Stefan Bouckaert, Jono Vanhie-Van Gerwen, Ingrid Moerman.

[10] "*Current Trends in Parallel Computing*", Rafiqul Zaman Khan, Md Firoj Ali, Department of Computer Science, Aligarh Muslim University, Aligarh, UP, INDIA.