The Technological Roadmap of Parallware and Its Alignment with the OpenPOWER Ecosystem

Manuel Arenaz[1(✉)], Oscar Hernandez[2], and Dirk Pleiter[3]

[1] University of A Coruna and Appentra Solutions, A Coruña, Spain
manuel.arenaz@appentra.com
[2] Oak Ridge National Laboratory, Oak Ridge, TN, USA
oscar@ornl.gov
[3] Julich Supercomputing Center, Jülich, Germany
d.pleiter@fz-juelich.de

Abstract. Accelerated, heterogeneous systems are becoming the norm in High Performance Computing (HPC). The challenge is choosing the right parallel programming framework to maximize performance, efficiency and productivity. The design and implementation of benchmark codes is important in many activities carried out at HPC facilities. Well known examples are fair comparison of R+D results, acceptance tests for the procurement of HPC systems, and the creation of miniapps to better understand how to port real applications to current and future supercomputers. As a result of these efforts there is a variety of public benchmark suites available to the HPC community, e.g., Linpack, NAS Parallel Benchmarks (NPB), CORAL benchmarks, and Unified European Application Benchmark Suite. The upcoming next generation of supercomputers is now leading to create new miniapps to evaluate the potential performance of different programming models on mission critical applications, such as the XRayTrace miniapp under development at the Oak Ridge National Laboratory. This paper presents the technological roadmap of Parallware, a new suite of tools for high-productivity HPC education and training, that also facilitates the porting of HPC applications. This roadmap is driven by best practices used by HPC expert developers in the parallel scientific C/C++ codes found in CORAL, NPB, and XRayTrace. The paper reports preliminary results about the parallel design patterns used in such benchmark suites, which define features that need to be supported in upcoming realeases of Parallware tools. The paper also presents performance results using standards OpenMP 4.5 and OpenACC 2.5, compilers GNU and PGI, and devices CPU and GPU from IBM, Intel and NVIDIA.

Keywords: Hybrid heterogeneous programming models · OpenACC · OpenMP4 · Static analysis tools · LLVM · Performance portability · Use of parallware on minsky

J.M. Kunkel et al. (Eds.): ISC High Performance Workshops 2017, LNCS 10524, pp. 237–253, 2017.
https://doi.org/10.1007/978-3-319-67630-2_19

1 Introduction

Science, Technology, Engineering and Mathematics (STEM) plays a key role in the sustained growth and stability of the economy world-wide. There is a huge and urgent need in training STEM people in parallel programming, as well as in providing STEM people with better programming environments that help in porting scientific applications to modern supercomputers. It is key for code modernization, specially to exploit the computational power of new devices such as NVIDIA GPUs and Intel Xeon Phi.

The new specifications of directive-based parallel programming standards OpenMP 4.5 [24] and OpenACC 2.5 [23] are increasingly complex in order to support heterogeneous computing systems. There is some debate regarding the prescriptive nature of OpenMP 4.5 compared to the descriptive capabilities available in OpenACC. In addition, performance portability is not guaranteed by OpenMP 4.5 specification, and currently there is divergence in features supported by different vendors (e.g., PGI, Cray, IBM, Intel, GNU) in different devices (e.g., CPU, GPU, KNC/KNL). Overall, the responsibility falls on the developer to choose best practices that facilitate performance portability [18,20].

The new Parallware tools [2] aim at going one step forward towards supporting best practices for performance portability. It is useful to track the progress of new features in OpenMP 4.5 and OpenACC 2.5, as well as the availability and performance of their implementation in each available compiler and target device. We expect Parallware tools to be of interest for HPC facilities in the following use cases:

- Improvement of current HPC education and training environments in order to provide experiential learning to STEM people.
- Design and implementation of new miniapps to help porting HPC applications to next generation supercomputers.
- Creation of acceptance tests for HPC systems procurement.
- Port of HPC applications to upcoming (pre-)exascale supercomputers.

The rest of the paper is organized as follows. Section 2 discusses best practices from HPC expert developers in the parallel programming of CORAL Benchmarks [9], NAS Parallel Benchmarks [6] and XRayTrace miniapp [19]. The nomenclature of the *parallel design patterns* used internally in Parallware technology is introduced in order to compare the parallelization of the benchmark codes. Section 3 presents the new tool Parallware Trainer using as a guide the CORAL microbenchmark HACCmk. The technological challenges to be addressed in Parallware core technology in order to support modern HPC applications are described. It also introduces the foreseen tool Parallware Assistant oriented to development of HPC codes. Section 4 presents the current technological roadmap of Parallware tools. Finally, Sect. 6 presents conclusions and future work.

2 Analysis of Benchmarks: CORAL, NPB and XRayTrace

Benchmark suites are designed to test the performance of supercomputers at a hardware/software level, ranging from processor architecture, memory, interconnection network, I/O, file system, operating system, up to user applications that are mission critical for HPC facilities. In this study we analyze OpenMP and OpenACC parallel implementations of the compute-intensive C/C++ codes found in XRayTrace [19], CORAL [9] and NPB [6].

2.1 Parallel Design Patterns of Parallware

Several approaches try to divide programs into (parallel) algorithmic patterns and follow a pattern-based analysis of the code (see McCool et al. [22], Mattson et al. [21], Berkeley's dwarfs (or motifs) [5]). However, such patterns seem to be too difficult to apply in practice [25]. In contrast, we use the parallel design patterns detected by the Parallware technology [4], which have been successfully applied to real programs from the NAS Parallel Benchmarks [16], and from the fields computational electromagnetics [10], oil & gas [3] and computational astrophysics [12].

The pseudocodes presented in Fig. 1 describe three parallel design patterns detected by Parallware technology. The *Parallel Forall* of Listing 1.1 represents parallel computations without race conditions at run-time. In each iteration of `for_j`, a new value `A[j]` is computed. The value `T` is a loop temporary computed in each iteration of `for_j`, where `B[j]` denotes read-only values. The *Parallel Scalar Reduction* of Listing 1.2 represent a reduction operation whose result is a single value `A`, where `+` is a commutative, associative operator. The *Parallel Sparse Reduction* of Listing 1.3 represent a reduction operation whose result is a set of values. Each iteration of `for_j` updates a single value `A[B[j]]`, where the access pattern can only be determined at run-time and thus there may appear race conditions during the parallel execution.

Examples of parallel design patterns are presented in source code snippets of XRayTrace and HACCmk. In Listing 1.5, the loop `for_i` contains a parallel forall where the output is `vx1`, where `dx1` is a loop temporary and `fcoeff` is a read-only value. In Listing 1.6, the loop `for_j` contains a parallel scalar reduction whose output is `xi`, where `dxc` is a loop temporary and `fcxx1oeff` is a set of read-only values. Finally, Listing 1.4, the loop `for_it` contains parallel sparse reductions where the outputs `image` and `I_ang` have access patterns that are known at run-time only.

Listing 1.1. Fully Parallel Loop.

```
for (j=0; j<n; j++)
{
    T = B[j];
    A[j] = T;
}
```

Listing 1.2. Parallel Scalar Reduction.

```
for (j=0; j<n; j++)
{
    T = B[j];
    A += T;
}
```

Listing 1.3. Parallel Sparse Reduction.

```
for (j=0; j<n; j++)
{
    T = B[j];
    A[B[j]] += T;
}
```

Fig. 1. Pseudocodes of the *parallel design patterns* used in Parallware.

Listing 1.4. In XRayTrace version of routine *RayTraceImageLoop*

```
#define K_MAX 100 // Maximum number of frequencies
void RayTraceImageLoop(
     int N, int nx, int ny, int na, int nb, int nv,
     const double *x, const double *y, const double *a,
     const double *b, double dx, double dy, double dz,
     double da, double db, const double *dv,
     const RayTrace::ray_gain_struct *gain_in,
     const RayTrace::ray_seed_struct *seed_in,
     int method, const std::vector<ray_struct> &rays,
     double scale, double *image, double *I_ang,
     unsigned int &failure_code,
     std::vector<ray_struct> &failed_rays) {
     [...]
#pragma acc data copyin( x[0 : nx], y[0 : ny], a[0 : na],
     b[0 : nb], dv[0 : nv], rays2[0 :N_rays])
     deviceptr(gain, seed) copyout(image [0:nx * ny * nv],
     I_ang [0:na * nb]) {
// Initialize device images
#pragma acc parallel loop
          for (int i = 0; i < nx * ny * nv; ++i)
               image[i] = 0;
#pragma acc parallel loop
          for (int i = 0; i < na * nb; ++i)
               I_ang[i] = 0;
// Loop through y, x, b, a
#pragma acc parallel loop gang vector vector_length(32)
          for (int it = 0; it < N_rays; ++it) {
               const ray_struct ray = rays2[it];
               double Iv[K_MAX];
               ray_struct ray2;
               int error = RayTrace_calc_ray(
                    ray, N, dz, gain, seed, nv, method, Iv,
                    ray2);
               // Get the indices to the cells in image
               // and I_ang
               int i1, i2, i3, i4;
               int i1 = static_cast<int>(findfirstsingle(
                   x, nx, ray2.x - 0.5 * dx));
if (ray2.x < x[0] - 0.5 * dx ||
                    ray2.x > x[nx - 1] + 0.5 * dx)
                    i1 = -1; // The ray's z position is out
                             // of the range of image
               [...]
               // Copy I_out into image
               if (i1 >= 0 && i2 >= 0) {
                    double *Iv2 =
                        &image[nv * (i1 + i2 * nx)];
                    for (int iv = 0; iv < nv; iv++) {
#pragma acc atomic update
                         Iv2[iv] += Iv[iv] * scale;
                    }
               }
               // Copy I_out into I_ang
               if (i3 >= 0 && i4 >= 0) {
                    double tmp = 0.0;
                    for (int iv = 0; iv < nv; iv++)
                         tmp += 2.0 * dv[iv] * Iv[iv];
#pragma acc atomic update
                    I_ang[i3 + i4 * na] += tmp;
               }
          }
     } // pragma acc data region scope
}
```

Listing 1.5. CORAL microbenchmark HACCmk (file *main.c*).

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 15000
int main( int argc, char *argv[] )
{
  static float xx[N], yy[N], zz[N], mass[N], vx1[N], vy1[
      N], vz1[N];
  float fsrrmax2, mp_rsm2, fcoeff, dx1, dy1, dz1;
  int count = 327;
  ...
  for ( n = 400; n < N; n = n + 20 )
  {
    ...
    #pragma omp parallel for private( dx1, dy1, dz1 )
    for ( i = 0; i < count; ++i)
    {
      Step10_orig( n, xx[i], yy[i], zz[i], fsrrmax2,
          mp_rsm2, xx, yy, zz, mass, &dx1, &dy1, &dz1 );
      vx1[i] = vx1[i] + dx1 * fcoeff;
      vy1[i] = vy1[i] + dy1 * fcoeff;
      vz1[i] = vz1[i] + dz1 * fcoeff;
    }
    ...
  }
}
```

Listing 1.6. CORAL microbenchmark HACCmk (file *Step10_orig.c*).

```
#include <math.h>

void Step10_orig( int count1, float xxi, float yyi, float
     zzi, float fsrrmax2, float mp_rsm2, float *xx1,
    float *yy1, float *zz1, float *mass1, float *dxi,
    float *dyi, float *dzi )
{
    const float ma0 = 0.269327, ma1 = -0.0750978, ma2 =
        0.0114808, ma3 = -0.00109313, ma4 = 0.0000605491,
         ma5 = -0.00000147177;
    float dxc, dyc, dzc, m, r2, f, xi, yi, zi;
    int j;
    xi = 0.; yi = 0.; zi = 0.;
    for ( j = 0; j < count1; j++ )
    {
        dxc = xx1[j] - xxi;
        dyc = yy1[j] - yyi;
        dzc = zz1[j] - zzi;
        r2 = dxc * dxc + dyc * dyc + dzc * dzc;
        m = ( r2 < fsrrmax2 ) ? mass1[j] : 0.0f;
        f = pow( r2 + mp_rsm2, -1.5 ) - ( ma0 + r2*(ma1
            + r2*(ma2 + r2*(ma3 + r2*(ma4 + r2*ma5)))));
        f = ( r2 > 0.0f ) ? m * f : 0.0f;
        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
*dxi = xi;
    *dyi = yi;
    *dzi = zi;
}
```

The results in Table 1 reveal the usage of Parallware's parallel design patterns by HPC expert developers. The pattern *Parallel Forall* implementation *ParallelForLoopImpl* corresponds to a fully parallel loop implemented with pragmas *parallel for* in OpenMP and pragmas *parallel loop* in OpenACC. In contrast, the design patterns *Parallel Scalar Reduction* and *Parallel Sparse Reduction* add synchronization to guarantee correctness during parallel execution. Only three different implementations are used in the benchmarks: *AtomicImpl*, which prevents race conditions by adding pragmas *atomic* that guarantee atomic memory accesses to the reduction variable; *ReductionImpl*, which uses clause *reduction* to compute thread-local temporary results that are later reduced into the output reduction value; and *PrivateImpl*, which is a hand-made implementation of the clause *reduction* using clauses *private*/*shared* and pragma *critical*. The numbers show that *atomic* is not used for scalar reductions. In addition, *reduction* is not used for sparse reductions because the clause does not support arrays in OpenMP 3.1.

Table 1. Parallel design patterns used by HPC developers in the C/C++ implementation of NPB, CORAL and XRayTrace. (*) XRayTrace provides one OpenACC implementation; the remaining bechmark codes are OpenMP implementations only.

Benchmark	Parallel Design Pattern						
	Parallel Forall	Parallel Scalar Reduction			Parallel Sparse Reduction		
	Parallel ForLoop Impl	Atomic Impl	Reduction Impl	Private Impl	Atomic Impl	Reduction Impl	Private Impl
ORNL miniapp	0	0	0	0	2	0	0
XRayTrace					(*)2		
CORAL	24	0	1	1	0	0	4
lulesh	22						4
MILCmk	1		1	1			
HACCmk	1						
NPB	168	0	17	2	5	0	24
BT	25						14
CG	12		6				4
EP			2		1		
FT	7			1			
IS	5		1				
LU	28		3				2
MG	9		1	1			2
SP	32						2
UA	50		4		4		

2.2 Case Study: ORNL's Miniapp XRayTrace

Work in progress at ORNL is focused on creating the miniapp XRayTrace, a new benchmark that will be used to evaluate the performance of pre-exascale *Summit* supercomputer. The Listing 1.4 shows an excerpt of the GPU code implemented by the HPC expert using OpenACC 2.5. The miniapp also provides an OpenMP 3.1 version to run on multicore CPUs.

The routine `RayTraceImageLoop()` (see Listing 1.4, line 2) basically consists of a loop `for_it` that computes a parallel design pattern *Parallel Sparse Reduction* on variables `I_ang` and `image` using an *AtomicImpl* implementation (see `#pragma acc atomic`, lines 48 and 57). It is the best choice from the point of view of maintainability, as it provides a compact, easy-to-understand implementation. It is also applicable across standards and devices, all of which provide performant support for atomic operations. As shown in Table 1, it is noticeable that *AtomicImpl* was not the preferred implementation in NPB and CORAL, where *PrivateImpl* was largely the option of choice by HPC expert developers.

The OpenACC 2.5 implementation is optimized to reduce CPU-to/from-GPU data transfers, as this is a critical performance factor according to best practices for GPUs today [17]. The key issue here is to handle data scoping for scalar and array variables. The HPC expert developer has specified the array ranges in clauses `copyin` and `copyout` for arrays `x, y, a, b, dv, rays2, image, I_ang` (see Listing 1.4, line 14). In order to avoid unnecessary CPU-to/from-GPU data transfers, temporary array variables `gain` and `seed` have been allocated only in the device using the clause `deviceptr` and the API calls `copy_device()` and `free_device()`.

Finally, the C/C++ features used in the code[1] also pose technological challenges on the Parallware core technology. There are calls to the auxiliary functions `RayTrace_calc_ray()` and `findfirstsingle()` (see Listing 1.4, lines 29 and 35), aliases that temporarily point to the output array (see `double *Iv2` pointing to `double *image`, lines 45–46), as well as user-defined datatypes `ray_gain_struct`, `ray_seed_struct` and `ray_struct` (e.g. see lines 7, 8 and 9).

The execution times and speedups of Table 2 were measured on the *Juron* system at the Julich Supercomputing Centre (JSC). The hardware setup is a IBM S822LC with CPU 2x POWER8NVL, 10 cores each (8xSMT) and NVIDIA P100 GPUs (only 1 used for these runs). The tested setups are CPU-based sequential execution (*CPU Serial*), CPU-based parallel execution with OpenMP 3.1 (*CPU OMP3.1*) and GPU-based parallel execution with OpenACC 2.5 (*GPU ACC2.5*). The compiler flags for GCC 6.3 are `-fopenmp -O2` (thus, *CPU Serial* is measured as *CPU OMP3.1* with 1 thread). The flags for PGI 16.10 are `-mp -O2` for *CPU Serial* and *CPU OMP3.1*, and `-acc -O2 -ta=tesla` for *GPU ACC2.5*. Four increasing test sizes were considered: *Small ASE* (399000 rays, 3 lengths), *Medium ASE* (399000 rays, 8 lengths), *Small Seed* (7803000 rays, 3 lengths) and *Medium Seed* (7803000 rays, 8 lengths). The numbers show that GCC is

[1] The code also uses the C++ STL (`std::vector &failed_rays`). However, we do not consider it a key challenge because it has been commented out in the OpenACC code (the same may stand for OpenMP as well).

Table 2. Execution times (in seconds) and speedups of XRayTrace in Finisterrae (CPU Intel Xeon and NVIDIA GPU P100).

Test size	Original							
	Small ASE		Medium ASE		Small seed		Medium seed	
Compiler GCC 6.3								
CPU serial	4.42	–	10.69	–	46.21	–	94.33	–
CPU OMP3.1	0.14	31×	0.294	36×	1.579	29×	2.99	31×
Compiler PGI 16.10								
CPU serial	3.19	–	8.10	–	50.27	–	120.55	–
CPU OMP3.1	17.80	0.18×	49.07	0.08×	1126.21	0.04×	907.95	0.13×
GPU ACC2.5	0.04	79×	0.11	73×	0.73	68×	1.99	60×

the best choice for multi-threaded execution on the CPU (minimum speedup is 29× using 160 threads, with respect to *CPU Serial*). In contrast, PGI enables efficient execution on the GPU, which is $1.5\times - 3.5\times$ faster than *CPU OMP3.1* using GCC (minimum speedup is 60×, with respect to *CPU Serial*).

Finally, the execution times and speedups of Table 3 were measured on the *Finisterrae* system at the Supercomputing Centre of Galicia (CESGA). The test platform is a dual Intel Xeon E5-2680 v3 CPU, 12 cores each, running at 2.5 GHz (hyperthreading is disabled). The GPU accelerator for OpenACC computing is a Tesla K80. It is remarkable that, using the GCC compiler, *CPU OMP3.1* is significantly faster on Minsky nodes than on Intel-based nodes (minimum speedup is 29× on Juron, compared to 5.7× on Finisterrae). Regarding the PGI compiler, *CPU OMP3.1* does not perform well on Minsky nodes because the support of POWER ISA is very recent in PGI compilers, and more investigations on the correct usage of the PGI compiler are needed.

3 Parallware Trainer

Parallware Trainer [2] is a new interactive tool for high-productivity HPC education and training using OpenMP 4.5 and OpenACC 2.5. It allows experiential learning by providing an interactive, real-time GUI with editor capabilities to interact with the Parallware technology for source-to-source automatic parallelization of sequential codes.

Hereafter, the current strengths and weaknesses of Parallware Trainer binary release 0.4 (May 2017) are discussed using as a guide the CORAL microbenchmark HACCmk. Next, the suite of Parallware tools under development is presented, describing the key technological differences with respect to other tools available to the HPC community.

3.1 Case Study: CORAL Microbenchmark HACCmk

The Listings 1.5 and 1.6 show an excerpt of the source code of the CORAL microbenchmark HACCmk, written in C and parallelized using OpenMP 3.1 by an HPC expert developer. The main program consists of a loop that defines increasing tests sizes *n*, ranging from *400* up to *15000*. The HPC expert developer has used a parallel design pattern *Fully Parallel Loop*. For each test size, the pragma `#pragma omp parallel for` enables the conflict-free multi-threaded computation of the output arrays `vx1`, `vy1` and `vz1`.

Parallware Trainer 0.4 does not discover parallelism across calls to user-defined functions. It fails to find the fully parallel loop `for_i` in `main()` because of the call to `Step10_orig()` (see Listing 1.5, line 19). In contrast, Parallware succeeds to parallelize the loop `for_j` inside this routine (Listing 1.6, lines 10–22), and reports the following user messages:

```
Step10_orig.c:3:1: info: Analyzed function 'Step10_orig'
Step10_orig.c:10:5: info:
   Offloading to coprocessor device (use of target pragma)
   Parallel loop
   Dependencies due to temporary variables do not prevent parallelization:
      'dxc', 'm', 'r2', 'dzc', 'f', 'dyc'
   Parallel reduction on variable 'zi' with associative, commutative operator '+'
   Parallel reduction on variable 'yi' with associative, commutative operator '+'
   Parallel reduction on variable 'xi' with associative, commutative operator '+'
   Ranking of available parallelization strategies:
      #1 Use of the clause <reduction>   (*) selected
      #2 Use of pragma <atomic> (memory optimized)
   TODO list:
      * Complete access range for variables: xx1, yy1, zz1, mass1}
```

The code contains a parallel design pattern *Parallel Scalar Reduction* involving three scalar variables `xi`, `yi` and `zi`. It also displays the ranking of available implementations: `#1` being *ReductionImpl* and `#2` being *AtomicImpl* (see Table 1). Following best practices observed in CORAL and NPB, Parallware selects *ReductionImpl* as the option that minimizes synchronization during the execution of the parallel scalar reductions `xi`, `yi` and `zi`. The output source code produced by Parallware contains OpenMP 3.1, OpenACC 2.5 and OpenMP 4.5 pragmas annotated on loop `for_j` (see codes in Listings 1.7, 1.8 and 1.9).

Listing 1.7. In CORAL microbenchmark HACCmk, version of routine *Step10_orig* generated by Parallware Trainer using OpenMP 3.1.

```
    #pragma omp parallel default(none) shared(count1,
        fsrrmax2, mass1, mp_rsm2, xi, xx1, xxi, yi, yy1,
        yyi, zi, zz1, zzi)
    {
    #pragma omp for private(dxc, dyc, dzc, f, m, r2)
        reduction(+: zi) reduction(+: yi) reduction(+: xi
        ) schedule(auto)
    for ( j = 0; j < count1; j++ )
    {
...
        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
    } // end parallel
```

Listing 1.8. In CORAL microbenchmark HACCmk, version of routine *Step10_orig* generated by Parallware Trainer using OpenACC 2.5.

```
    #pragma acc data copy(xi, yi, zi) copyin(count1,
        fsrrmax2, mass1[], mp_rsm2, xx1[], xxi, yy1[],
        yyi, zz1[], zzi)
    {
    #pragma acc parallel
    {
    #pragma acc loop reduction(+: zi) reduction(+: yi)
        reduction(+: xi)
    for ( j = 0; j < count1; j++ )
    {
        ...
        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
    } // end parallel
    } // end data
```

Listing 1.9. In CORAL microbenchmark HACCmk, version of routine *Step10_orig* generated by Parallware Trainer using OpenMP 4.5.

```
    #pragma omp target map(to:xxi, fsrrmax2, mp_rsm2, xx1
        [], count1, yyi, zzi, yy1[], zz1[], mass1[]) map(
        tofrom:zi, yi, xi)
    {
    #pragma omp parallel default(none) shared(count1,
        fsrrmax2, mass1, mp_rsm2, xi, xx1, xxi, yi, yy1,
        yyi, zi, zz1, zzi)
    {
    #pragma omp for private(dxc, dyc, dzc, f, m, r2)
        reduction(+: zi) reduction(+: yi) reduction(+: xi
        ) schedule(auto)
    for ( j = 0; j < count1; j++ )
    {
        ...
        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
    } // end parallel
    } // end target
```

The OpenMP annotations manage data scoping explicitly both on the CPU version (Listing 1.7, lines 1–3, clauses `default`, `private`, `shared` and `reduction`) as well as on the accelerated version (Listing 1.9, line 1, clause `map`). Parallware also suggest a list of actions to be carried out by the user. For array variables `xx1`, `yy1`, `zz1` and `mass1`, the tool generates empty array ranges because it cannot determine the array elements to be transferred between the CPU and the accelerator (Listing 1.9, line 1, clause `map(to:...,xx1[])`). The development of more precise array range analysis is planned in Parallware's technological roadmap.

The execution times and speedups of Table 4 were measured on the *Juron* system at JSC (Table 5 shows similar numbers on the *Finisterrae* at CESGA). Running the OpenMP 3.1 version with 160 threads, we observe that HACCmk's original implementation runs faster than Parallware's automatically generated version (speedups 18× and 0.9×, respectively). The reason is that the HPC expert developer exploits coarser grain parallelism (number of parallel regions is 730 in Listing 1.5, line 13), while Parallware versions incur in high parallelization overhead because a parallel region is created/destroyed in each call to procedure `Step10_orig()` (number of parallel regions is $730 \times 327 = 238710$ in Listing 1.7). The PGI compiler provides worse performance for the original HACCmk (running time 9.14 versus 7.42 of GCC). However, the performance with Parallware version performs poorly, probably due to the fact that the PGI run-time incurs in higher overhead in creation/destruction of parallel regions. Work-in-progress aims at adding support for interprocedural detection of parallelism. By managing procedure calls that write only on scalar variables passed by reference (see Listing 1.5, line 19, parameters `&dx1`, `&dy1`, `&dz1`), Parallware will successfully detect and parallelize the fully parallel loop `for_i` (Listing 1.5, lines 17–23). By matching the HPC expert's parallel implementation, we expect Parallware to provide acceptable performance similar to the original version.

Finally, note that OpenACC-enabled version with PGI 16.10 does not accelerate CPU OMP3.1, probably because HACCmk requires many CPU-to/from-GPU data transfers of small size. However, it is remarkable that Parallware still achieves an speedup 4.8× with respect to the serial code. The numbers show that while thread creation/destruction is very expensive on the CPU, its overhead is not so critical on the GPU.

Overall, the experiments show that Parallware supports the parallel design pattern *Fully Parallel Loop*, but it needs improvements in inter-procedural analysis to exploit coarser-grain parallelism with OpenMP and OpenACC.

3.2 The Parallware Suite

Parallware technology [1,4] uses an approach for parallelism that does not rely on loop-level classical dependence analysis. The classical approach builds systems of mathematical equations whose solutions allow to identify pairs of memory references in the loop body that might lead to race conditions during the parallel execution of the loop. In contrast, Parallware uses a fast, extensible hierarchical classification scheme to address dependence analysis. It splits the code into

Table 3. Execution times (in seconds) and speedups of HACCmk in Finisterrae (CPU Intel Xeon and NVIDIA GPU Tesla K80).

Test size	Original							
	Small ASE		Medium ASE		Small seed		Medium seed	
Compiler GNU 4.8.2								
CPU Serial	3.57	–	8.82	–	41.23	–	90.42	–
CPU OMP3.1	0.47	7.6×	0.99	8.9×	7.27	5.7×	13.68	6.6×
Compiler PGI 16.10								
CPU Serial	3.24	–	8.09	–	41.36	–	96.87	–
CPU OMP3.1	1.17	2.8×	1.26	6.4×	33.74	1.2×	117.25	0.8×
GPU ACC2.5	0.24	13.5×	0.42	19.3×	2.91	14.2×	6.19	15.6×

a small domain-independent computational kernels (e.g. assignment, reduction, recurrence, etc.), combining multiple static analysis techniques including array access patterns, array access ranges, and alias analysis. Next, it checks contextual properties between the kernels in order to discover parallelism and to select the most appropriate paralleling strategy for the loop. Finally, Parallware adds the corresponding OpenMP/OpenACC directives and performs code transformations as needed (e.g. array privatization in parallel reductions, which is natively supported by OpenMP in Fortran but not in C). Parallware is also based on the production-grade LLVM compiler infrastructure.

Parallware Trainer [2] is a new interactive tool for high-productivity HPC education and training. It allows experiential learning by providing an interactive, real-time GUI with editor capabilities to assist in the design and implementation of parallel code. Powered by the hierarchical classification engine of Parallware technology, it discovers parallelism, provides a ranking of parallel design patterns, and implements those designs using standards OpenMP 4.5 and OpenACC 2.5 (see video tutorials *How to use Parallware Trainer* available at www.parallware.com). Overall, the main advantages of Parallware Trainer are high availability 24×7, reduction of costs, and broader audience of STEM people in far-away geographical locations.

Parallware Assistant, currently under development, will be the next tool of the suite. The Parallware Trainer is oriented to HPC education and training, so its GUI only shows the key information needed to understand why a code snippet can be parallelized (e.g., contains a scalar reduction with an associative, commutative sum operator), and how it can be executed in parallel safely (e.g., atomic update of the sum operator). In contrast, the Parallware Assistant will provide detailed information about every operator and every variable of the code, for instance, detailed data scoping and detailed array access ranges.

Table 4. Execution times (in seconds) and speedups of HACCmk in Juron (CPU IBM Power8 and NVIDIA GPU P100).

	Original		Parallware	
Compiler GCC 6.3				
CPU serial	137.99	–		
CPU OMP3.1	7.42	18×	157.68	0.9×
Compiler PGI 16.10				
CPU serial	92.91	–		
CPU OMP3.1	9.14	10.8×	268.07	0.35×
GPU ACC2.5	n/a	n/a	19.13	4.8×

Table 5. Execution times (in seconds) and speedups of HACCmk in Finisterrae (CPU Intel Xeon and NVIDIA GPU Tesla K80).

	Original		Parallware	
Compiler GCC 6.3				
CPU serial	126.26	–		
CPU OMP3.1	12.37	10.2×	708.44	0.18×
Compiler PGI 16.10				
CPU serial	104.21	–		
CPU OMP3.1	10.17	10.3×	236.64	0.44×
GPU ACC2.5	n/a	n/a	32.11	3.2×

4 The Technological Roadmap of Parallware

Following current startup business practices, we seek to attain the minimum viable product (MVP) as quickly as possible. At the same time we are doing the MVP work, we are testing the market from the business side. We are discussing the sales cycle, price sensitiveness and business value to the target customers. We are conducting an early access program for Parallware Trainer, our first product for high-productivity STEM education and training in parallel programming for undergraduate and PhD levels.

Parallware's technological roadmap is driven by the best practices observed in CORAL, NPB and XRayTrace. Our go-to-market strategy is based on engaging with world-class HPC facilities, working together to better understand how to help them with their mission-critical activities (e.g., technology scouting, creation of benchmark codes, porting of HPC applications). Thus, we are participating in strategic partnership programs (BSC, ORNL and TACC) and deploying Parallware Trainer in real production environments (BSC, ORNL, NERSC, LRZ).

As of writing, our priorities for the technological development of Parallware Trainer in the short and medium term are (in order of priority):

1. Improve the usability of the GUI to facilitate the analysis, compilation and execution of scientific programs. Analyze programs across multiple source code files that use MPI, OpenMP and OpenACC. Integrate the GUI with compilers from different vendors (e.g., IBM, PGI) in production-level supercomputers (e.g. *modules*, job queuing systems).
2. Improve the support for parallel design patterns *parallel scalar reduction* and *parallel sparse reduction*. As of writing, Parallware already supports the *AtomicImpl* and *ReductionImpl* implementations for OpenMP 4.5 and OpenACC 2.5, both for CPU and GPU devices. Work-in-progress aims at adding support for the *PrivateImpl* implementation as well.
3. Provide a ranking of parallel implementations, suggesting the "best" option for a given parallel programming standard, compiler and device. Mechanisms for the user to select the preferred implementation will be added.
4. Provide a list of suggestions for the user to improve the parallel implementation generated by the Parallware tools. Current work aims at improving data scoping support though advanced techniques for array range analysis and detection of temporary arrays. This information is useful to minimize CPU-to/from-GPU data transfers, for example by allocating temporary arrays directly on the GPU device.
5. Improve the Parallware core technology to discover parallelism across procedure calls. Inter-procedural analysis (IPA)[2] usually requires handling user-defined data structures (e.g. *struct*), auxiliary pointer variables that alias with output variables (*aliasing*), and C++ STL classes (e.g. `std::vector`).

Finally, our go-to-market strategy is aligned with world-class exhibitions ISC High Performance 2017 (ISC'17) and Supercomputing 2017 (SC17). During 2017, we plan to commercially launch Parallware Trainer, an interactive, real-time editor with GUI features to facilitate the learning, usage, and implementation of parallel programming. We also plan to test a prototype of Parallware Assistant, a new software tool that will offer a high-productivity programming environment to help HPC experts to manage the complexity of parallel programming. Example of technical features are detailed data scoping at the loop and functions levels, and visual browsing of the parallelism found in the code.

5 Related Work

Parallelization tools have been built in the past that discover parallelism in loops via symbolic equations, where the user can input ranges of values given a set of inputs. Some of these tools include SUIF [15], Polaris [7], Cetus [8], iPAT/OMP [11] and ParaWise [13] (CAPTools/CAPO). However, these tools are extremely hard to use with real applications, even for advanced application developers because they rely on research compilers with complex user interfaces and they take significant amount of time to complete their analysis. Some of

[2] Requirement for porting HPC applications, not for HPC education and training.

them are restricted to the older Fortran 77 standard or focus on loop level parallelism for simple array operations. In contrast, Parallware uses a fast, extensible hierarchical classification scheme to address dependence analysis. Based on the production-grade LLVM compiler infrastructure, Parallware is beginning to show success on the parallelization of C codes that defeat the other tools.

6 Conclusions and Future Work

Preliminary results suggest that the parallel design patterns used by HPC expert developers have not changed significantly across NPB, CORAL and new bechmarks such as XRayTrace. The latest updates in OpenMP 4.5 and OpenACC 2.5 improve support for reductions and atomic operations. This is expected to simplify implementations, leading to better productivity and maintainability.

Writing performance portable code is a challenge and a responsability for the programmer. Parallware tools are a step forward to help in this regard by supporting best practices for OpenMP 4.5 and OpenACC 2.5 across different compilers and devices. The parallel design patterns used in Parallware technology have been shown to be an effective approach to discover the parallelism available in the benchmark NPB, CORAL and XRayTrace.

As future work, we plan to finish this study with NPB, CORAL, XRayTrace and other well-known benchmark suites such as SPECaccel. We are aware of the importance of Fortran, and we are working to support it as soon as the new Fortran front-end is available for LLVM [14].

Acknowledgements. The authors gratefully acknowledge the access to the HPB PCP Pilot Systems at Julich Supercomputing Centre, which have been partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 604102 (HPB). Also thanks to the Supercomputing Centre of Galicia (CESGA) for providing access to the FinisTerrae supercomputer.

References

1. Andión, J., Arenaz, M., Rodríguez, G., Touriño, J.: A novel compiler support for automatic parallelization on multicore systems. Parallel Comput. **39**(9), 442–460 (2013)
2. Appentra: Parallware Trainer, April 2017. http://www.parallware.com/
3. Arenaz, M., Domínguez, J., Crespo, A.: Democratization of HPC in the oil & gas industry through automatic parallelization with parallware. In: 2015 Rice Oil and Gas HPC Workshop, March 2015
4. Arenaz, M., Touriño, J., Doallo, R.: XARK: an extensible framework for automatic recognition of computational kernels. ACM Trans. Program. Lang. Syst. (TOPLAS) **30**(6), 32:1–32:56 (2008)
5. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. Technical report, UC Berkeley (2006)

6. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS parallel benchmarks - summary and preliminary results. In Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991, pp. 158–165. ACM (1991)
7. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with Polaris. Computer **29**(12), 78–82 (1996)
8. Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. IEEE Micro **42**(12), 36–42 (2009)
9. Department of Energy (DoE): CORAL Benchmark Codes (2014). https://asc.llnl.gov/CORAL-benchmarks/
10. Gómez-Sousa, H., Arenaz, M., Rubiños-López, O., Martínez-Lorenzo, J.: Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments. In: Proceedings of the 9th European Conference on Antenas and Propagation, EuCap 2015, April 2015
11. Ishihara, M., Honda, H., Sato, M.: Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP. IEICE Trans. Inf. Syst. **89**–**D**(2), 399–407 (2006)
12. Jiang, Q., Lee, Y.C., Zomaya, A., Arenaz, M., Leslie, L.: Optimizing scientific workflows in the cloud: a montage example. In: Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC), pp. 517–522. IEEE, December 2014
13. Johnson, S., Evans, E., Jin, H., Ierotheou, C.: The ParaWise expert assistant – widening accessibility to efficient and scalable tool generated OpenMP code. In: Chapman, B.M. (ed.) WOMPAT 2004. LNCS, vol. 3349, pp. 67–82. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31832-3_7
14. Lawrence Livermore National Laboratory: Open-Source Fortran Compiler Technology for LLVM (2015). https://www.llnl.gov/news/nnsa-national-labs-team-nvidia-develop-open-source-fortran-compiler-technology
15. Liao, S.-W., Diwan, A., Bosch Jr., R.P., Ghuloum, A., Lam, M.S.: SUIF explorer: an interactive and interprocedural parallelizer. In: Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPopp 1999, pp. 37–48. ACM Press, New York (1999)
16. Lobeiras, J., Arenaz, M.: a success case using parallware: the NAS parallel benchmark EP. In: Proceedings of the OpenMPCon Developers Conference (2015)
17. Lobeiras, J., Arenaz, M., Hernández, O.: Experiences in extending parallware to support OpenACC. In: Chandrasekaran, S., Foertter, F. (eds.) Proceedings of the Second Workshop on Accelerator Programming using Directives, WACCPD 2015, Austin, Texas, USA, 15 November 2015, pp. 4:1–4:12. ACM (2015)
18. Lopez, M.G., Larrea, V.V., Joubert, W., Hernandez, O., Haidar, A., Tomov, S., Dongarra, J.: Towards achieving performance portability using directives for accelerators. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD 2016, pp. 13–24. IEEE Press, Piscataway (2016)
19. Berril, M.: XRayTrace miniapp (2017). https://code.ornl.gov/mbt/RayTrace-miniapp
20. Martineau, M., Price, J., McIntosh-Smith, S., Gaudin, W.: Pragmatic performance portability with OpenMP 4.x. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 253–267. Springer, Cham (2016). doi:10.1007/978-3-319-45550-1_18

21. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional (2004)
22. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
23. OpenACC Architecture Review Board: The OpenACC Application Programming Interface, Version 2.5, October 2015. http://www.openacc.org
24. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.5, November 2015. http://www.openmp.org
25. Wienke, S., Miller, J., Schulz, M., Müller, M.S.: Development effort estimation in HPC. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 10:1–10:12. IEEE Press, Piscataway (2016)