Yet Another Approach to Small and Medium Scale Parallelization of Adaptive Monte Carlo Integration

Thorsten Ohl[*][†]

Darmstadt University of Technology
Schloßgartenstr. 9
D-64289 Darmstadt
Germany

IKDA 98/??
hep-ph/yymmnnn
July 1998

Abstract

. . .

1 Introduction

The problem of the parallelization of adaptive Monte Carlo integration algorithms has gained some attention recently [1, 2]. Both authors present parallel versions of the Vegas algorithm [3].

The implementations start from the classic implementation of Vegas and add synchronization barriers, either mutexes for threads accessing shared memory or explicit message passing. This approach results in compact code and achieves high performance, but the implementations of threads bases parallelism on one hand and message passing on the other are very different.

[*] e-mail: `ohl@hep.tu-darmstadt.de`

[†] Supported by Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie, Germany.

Therefore, a close coupling of parallelization and of the integration algorithm sacrifices flexibility. Even the move from one message passing library to another is a non trivial exercise with many subtle failure modes. The same is true for any improvement of the integration algorithm.

Instead, we suggest a *mathematical* model of parallelism for adaptive Monte Carlo integration that is independend both of a concrete paradigm for parallelism and of the programming language used for an implementation. We decompose the algorithm and prove that certain parts can be executed in *any* order without changing the result. As a corollary, we know that they can be executed in parallel.

The algorithms presented below have been implemented successfully in the library VAMP [4], along with other, independent, improvements of Vegas [5].

In section 2 we discuss the features of Vegas, that are important for our model.

2 Vegas

In this section we discuss the features of Vegas, that are important for building a model of parallelism, but are not discussed in [3].

Vegas uses two *grids*: an adaptive grid G^A, which is used to adapt the distribution of the sampling points and a stratification grid G^S for stratified sampling. The latter is static and depends only on the number of dimensions and on the number of sampling points. Both grids factorize into *divisions* $d^i_{A,S}$

$$G^A = d_1^A \otimes d_2^A \otimes \cdots \otimes d_n^A \tag{1a}$$

$$G^S = d_1^S \otimes d_2^S \otimes \cdots \otimes d_n^S \,. \tag{1b}$$

The divisions come in three kinds

$$d_i^S = \emptyset \qquad \text{(importance sampling)} \tag{2a}$$

$$d_i^A = d_i^S/m \qquad \text{(stratified sampling)} \tag{2b}$$

$$d_i^A \neq d_i^S/m \qquad \text{(pseudo-stratified sampling)}\,. \tag{2c}$$

In the classic implementation of Vegas [3], *all* divisions are of the same type. In a more general implementation [4], this is not required and it can be useful to use stratification only in a few dimensions.

Two-dimensional grids for the cases (2a) and (2b) are illustrated in figure 1. In case (2a), there is no stratification grid and the points are picked

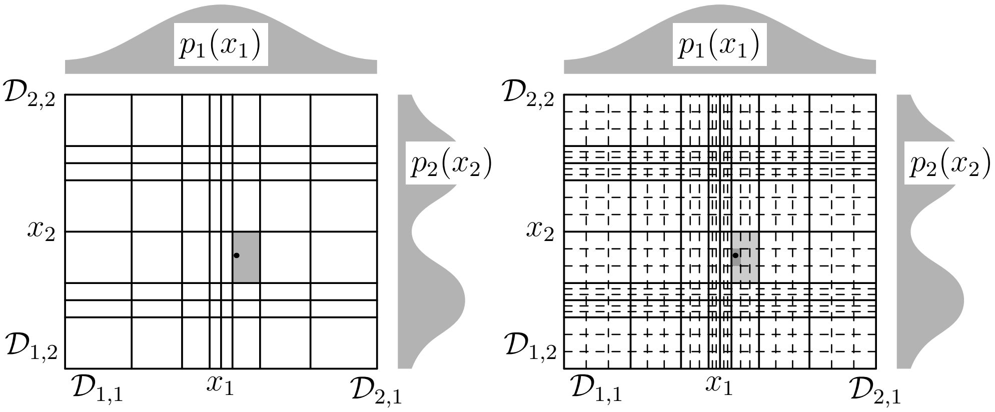

Figure 1: Vegas grid structure for importance sampling (2a) on the left and for genuinely stratified sampling (2b) on the right. The latter is used in low dimensions only.

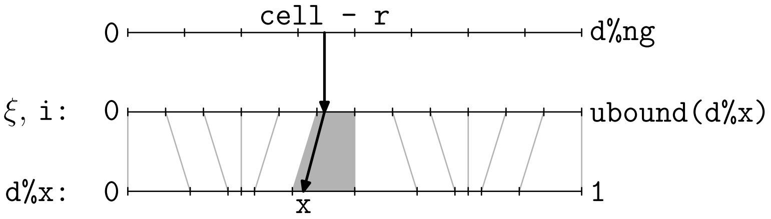

Figure 2: One-dimensional illustration of the `vegas` grid structure for pseudo stratified sampling, which is used in high dimensions.

at random in the whole region according to G_A. In case (2b), the adaptive grid G_A is a regular subgrid of the stratification grid G_S and an equal number of points are picked at random in each cell of G_S. Since $d_i^A = d_i^S/m$, the points will be distributed according to G_A as well.

A one-dimensional illustration of (2c) is shown in figure (2). The case (2c) is the most complicated.

3 Parallelization

3.1 Formalization of Adaptive Sampling

In order to discuss the problems with parallelizing adaptive integration algorithms and to present solutions, it helps to introduce some mathematical notation. A sampling S is a map from the space π of point sets and the space F of functions to the real (or complex) numbers

$$\begin{aligned} S : \pi \times F &\to \mathbf{R} \\ (p, f) &\mapsto I = S(p, f) \end{aligned}$$

For our purposes, we have to be more specific about the nature of the point set. In general, the point set will be characterized by a sequence of pseudo random numbers $\rho \in R$ and by one or more grids $G \in \Gamma$ used for importance or stratified sampling. A simple sampling

$$\begin{aligned} S_0 : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\to R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (\rho', G, a', f, \mu'_1, \mu'_2) = S_0(\rho, G, a, f, \mu_1, \mu_2) \end{aligned} \tag{3}$$

estimates the n-th moments $\mu'_n \in \mathbf{R}$ of the function $f \in F$. The integral and its standard deviation can be derived easily from the moments

$$I = \mu_1 \tag{4a}$$

$$\sigma^2 = \frac{1}{N-1} \left(\mu_2 - \mu_1^2\right) \tag{4b}$$

while the latter are more convenient for the following discussion. In addition, S_0 collects auxiliary information to be used in the grid refinement, denoted by $a \in A$. The unchanged arguments G and f have been added to the result of S_0 in (3), so that S_0 has identical domain and codomain and can therefore be iterated. Previous estimates μ_n may be used in the estimation of μ'_n, but a particular S_0 is free to ignore them as well. Using a little notational freedom, we augment $\mathbf{R}$ and A with a special value $\perp$, which will always be discarded by S_0.

In an adaptive integration algorithm, there is also a refinement operation $r : \Gamma \times A \to \Gamma$ that can be extended naturally to the codomain of S_0

$$\begin{aligned} r : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\to R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (\rho, G', a, f, \mu_1, \mu_2) = r(\rho, G, a, f, \mu_1, \mu_2) \end{aligned} \tag{5}$$

so that $S = rS_0$ is well defined and we can specify n-step adaptive sampling as

$$S_n = S_0(rS_0)^n \tag{6}$$

Since, in a typical application, only the estimate of the integral and the standard deviation are used, a projection can be applied to the result of S_n:

$$\begin{aligned} P : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\to \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (I, \sigma) \end{aligned} \tag{7}$$

Then

$$(I, \sigma) = PS_0(rS_0)^n(\rho, G_0, \perp, f, \perp, \perp) \tag{8}$$

and a good refinement prescription r, such as Vegas, will minimize the σ.

For parallelization, it is crucial to find a division of S_n or any part of it into *independent* pieces that can be evaluated in parallel. In order to be effective, r has to be applied to *all* of a and therefore a sychronization of G before and after r is appropriately. Forthermore, r usually uses only a tiny fraction of the CPU time and it makes little sense to invest a lot of effort into parallelizing it beyond what the Fortran compiler can infer from array notation. On the other hand, S_0 can be parallelized naturally, because all operations are linear, including he computation of a. We only have to make sure that the cost of communicating the results of S_0 and r back and forth during the computation of S_n do not offset any performance gain from parallel processing.

When we construct a decomposition of S_0 and proof that it does not change the results, i.e.

$$S_0 = \iota S_0 \phi \tag{9}$$

where ϕ is a forking operation and ι is a joining operation, we are faced with the technical problem of a parallel random number source ρ.

$$\begin{tikzcd}
\bigoplus_{i=1}^N G_i \arrow[r, "\bigoplus_{i=1}^N S_0"] & \bigoplus_{i=1}^N G_i \arrow[d, "\iota"'] \\
G \arrow[u, "\phi"] \arrow[r, "S_0"] & G
\end{tikzcd} \tag{10}$$

3.2 Weakly Commutative Diagrams

As made explicit in (3, S_0 changes the state of the random number general ρ, demanding *identical* results therefore imposes a strict ordering on the operations and defeats parallelization. It is possible to devise implementations of S_0 and ρ that circumvent this problem by distributing subsequences of ρ in such a way among processes that results do not depend on the number of parallel processes.

However, a reordering of the random number sequence will only change the result by the statistical error, as long as the scale of the allowed reorderings is *bounded* and much smaller than the period of the random number generator [1] Below, we will therefore use the notation $x \approx y$ for "equal for an appropriate finite reordering of the ρ used in calculating x and y". For our

[1] Arbirtrary reorderings on the scale of the period of the random number generators could select constant sequences and have to be forbidden.

porposes, the relation $x \approx y$ is strong enough and allows simple and efficient implementations.

3.3 Multilinear Structure of the Sampling Algorithm

Since S_0 is essentially a summation, it is natural to expect a linear structure

$$\bigoplus_i S_0(\rho_i, G_i, a_i, f, \mu_{1,i}, \mu_{2,i}) \approx S_0(\rho, G, a, f, \mu_1, \mu_2) \tag{11a}$$

where

$$\rho = \bigoplus_i \rho_i \tag{11b}$$

$$G = \bigoplus_i G_i \tag{11c}$$

$$a = \bigoplus_i a_i \tag{11d}$$

$$\mu_n = \bigoplus_i \mu_{n,i} \tag{11e}$$

for appropriate definitions of "$\oplus$". For the moments, we have standard addition

$$\mu_{n,1} \oplus \mu_{n,2} = \mu_{n,1} + \mu_{n,2} \tag{12}$$

and since we only demand equality up to reordering, we only need that the ρ_i are statistically independent. This leaves us with G and a and we have to discuss importance sampling ans stratified sampling separately.

3.3.1 Importance Sampling

In the case of naive Monte Carlo and importance sampling the natural decomposition of G is to take j copies of the same grid G/j which is identical to G, each with one j-th of the total sampling points. As long as the a are linear themselves, we can add them up just like the moments

$$a_1 \oplus a_2 = a_1 + a_2 \tag{13}$$

and we have found a decomposition (11). In the case of Vegas, the a_i are sums of function values at the sampling points. Thus they are obviously linear and this approach is applicable to Vegas in the importance sampling mode.

3.3.2 Stratified Sampling

The situation is more complicated in the case of stratified sampling. The first complication is that in pure stratified sampling there are only two sampling points per cell. Splitting the grid in two pieces as above provide only a very limited amount of parallelization. The second complication is that the a are no longer linear, since they corrspond to a sampling of the variance per cell and no longer of function values themselves.

However, as long as the samplings contribute to disjoint bins only, we can still "add" the variances by combining bins. The solution is therefore to divide the grid into disjoint bins along the divisions of the stratification grid and to assign a set of bins to each processor.

Finer decompositions will incur higher communications costs and other resource utilization. An implementation based on PVM is described in [2], which miminizes the overhead by running identical copies of the grid G on each processor. Since most of the time is usually spent in function evaluations, it makes sense to run a full S_0 on each processor, skipping function evaluations everywhere but in the region assigned to the processor. This is a neat trick, which is unfortunately tied to the computational model of message passing systems such as PVM and MPI [11]. More general paradigms can not be supported since the separation of the state for the processors is not explicit (it is implicit in the separated address space of the PVM or MPI processes).

However, it is possible to implement (11) directly in an efficient manner. This is based on the observation that the grid G used by Vegas is factorized into divisions D^j for each dimension

$$G = \bigotimes_{j=1}^{n_{\text{dim}}} D^j \tag{14}$$

and decompositions of the D^j induce decompositions of G

$$\begin{aligned} G_1 \oplus G_2 &= \left(\bigotimes_{j=1}^{i-1} D^j \otimes D_1^i \otimes \bigotimes_{i=j+1}^{n_{\text{dim}}} D^j \right) \oplus \left(\bigotimes_{j=1}^{i-1} D^j \otimes D_2^i \otimes \bigotimes_{i=j+1}^{n_{\text{dim}}} D^j \right) \\ &= \bigotimes_{j=1}^{i-1} D^j \otimes \left(D_1^i \oplus D_2^i \right) \otimes \bigotimes_{j=i+1}^{n_{\text{dim}}} D^j \end{aligned} \tag{15}$$

We can translate (15) directly to code that performs the decomposition $D^i = D_1^i \oplus D_2^i$ discussed below and simply duplicates the other divisions $D^{j \neq i}$. A decomposition along multiple dimensions is implemented by a recursive application of (15).

In Vegas, the auxiliary information a inherits a factorization similar to the grid (14)

$$a = (d^1, \ldots, d^{n_{\text{dim}}}) \tag{16}$$

but not a multilinear structure. Instead, *as long as the decomposition respects the stratification grid*, we find the in place of (15)

$$a_1 \oplus a_2 = (d_1^1 + d_2^1, \ldots, d_1^i \oplus d_2^i, \ldots, d_1^{n_{\text{dim}}} + d_2^{n_{\text{dim}}}) \tag{17}$$

with "+" denoting the standard addition of the bin contents and "$\oplus$" denoting the aggregation of disjoint bins. If the decomposition of the division would break up cells of the stratification grid (17) would be incorrect, because, as discussed above, the variance is not linear.

Now it remains to find a decomposition

$$D^i = D_1^i \oplus D_2^i \tag{18}$$

for both the pure stratification mode and the pseudo stratification mode of vegas (cf. figure 1). In the pure stratification mode, the stratification grid is strictly finer than the adaptive grid and we can decompose along either of them immediately. Technically, a decomposition along the coarser of the two is straightforward. Since the adaptive grid already has more than 25 bins, a decomposition along the stratification grid makes no practical sense and the decomposition along the adaptive grid has been implemented. The sampling algorithm S_0 can be applied *unchanged* to the individual grids resulting from the decomposition.

For pseudo stratified sampling (cf. figure 2), the situation is more complicated, because the adaptive and the stratification grid do not share bin boundaries. Since Vegas does *not* use the variance in this mode, it would be theoretically possible to decompose along the adaptive grid and to mimic the incomplete bins of the stratification grid in the sampling algorithm. However, this would be a technical complication, destroying the universality of S_0. Therefore, the adaptive grid is subdivided in a first step in

$$\operatorname{lcm}\left(\frac{\operatorname{lcm}(n_f, n_g)}{n_f}, n_x\right) \tag{19}$$

bins,[2] such that the adaptive grid is strictly finer than the stratification grid. This procedure is shown in figure 3.

[2] The coarsest grid covering the division of n_g bins into n_f forks has $n_g/\gcd(n_f, n_g) = \operatorname{lcm}(n_f, n_g)/n_f$ bins per fork.

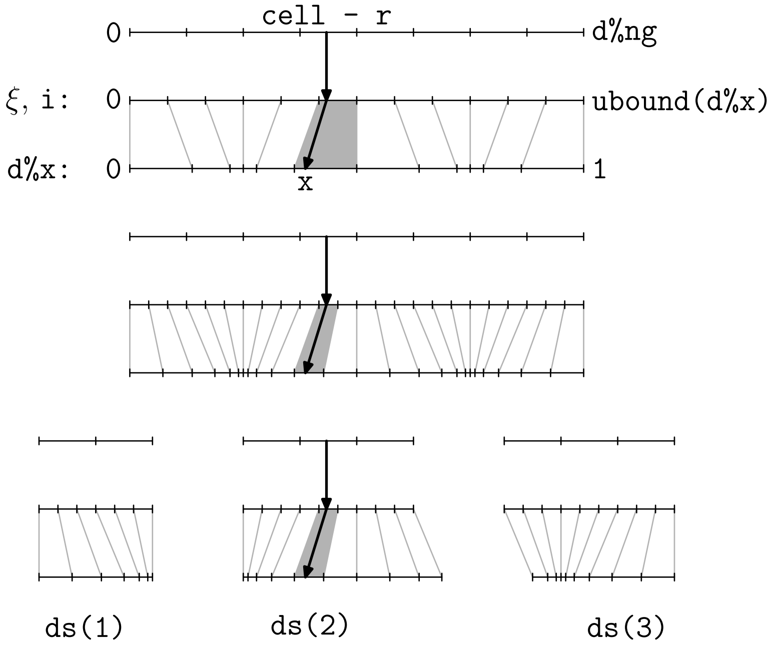

Figure 3: Forking one dimension `d` of a grid into three parts `ds(1)`, `ds(2)`, and `ds(3)`. The picture illustrates the most complex case of pseudo stratified sampling (cf. fig. 2).

3.4 State and Message Passing

3.5 Random Numbers

In the parallel example sitting on top of MPI [11] takes advantage of the ability of Knuth's generator [12] to generate statistically independent subsequences. However, since the state of the random number generator is explicit in all procedure calls, other means of obtaining subsequences can be implemented in a trivial wrapper.

The results of the parallel example will depend on the number of processors, because this effects the subsequences being used. Of course, the variation will be compatible with the statistical error. It must be stressed that the results are deterministic for a given number of processors and a given set of random number generator seeds. Since parallel computing environments allow to fix the number of processors, debugging of exceptional conditions is possible.

4 Practice

In this section we show three implementations of S_n: one serial, and two parallel, based on HPF [9, 10] and MPI [11], respectively. From these examples, it should be obvious how to adapt VAMP to other parallel computing paradigms.

4.1 Serial

Here is a bare bones serail version of S_n, for comparison with the parallel versions below. The real implementation of `vamp_sample_grid` in the module `vamp` includes some error handling, diagnostics and the projection P (cf. (7)):

```
subroutine vamp_sample_grid (rng, g, iterations, func)
  type(tao_random_state), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: iterations
  ⟨⟨ Interface declaration for func ⟩⟩
  integer :: iteration
  iterate: do iteration = 1, iterations
     call vamp_sample_grid0 (rng, g, func)
     call vamp_refine_grid (g)
  end do iterate
end subroutine vamp_sample_grid
```

4.2 HPF

The HPF version of S_n is based on decomposing the grid `g` as described in section 3.3 and lining up the components in an array `gs`. The elements of `gs` can then be processed im parallel. This version can be compiled with any Fortran compiler and a more complete version of this procedure (including error handling, diagnostics and the projection P) is included with VAMP as `vamp_sample_grid_parallel` in the module `vamp`. This way, the algorithm can be tested on a serial machine, but there will obviously be no performance gain.

Instead of one random number generator state `rng`, it takes an array consisting of one state per processor. These `rng(:)` are assumed to be initialized, such that the resulting sequences are statistically independent. For this purpose, Knuth's random number generator [12] is most convenient and is included with VAMP (see the example on page 12). Before each S_0, the procedure `vamp_distribute_work` determines a good decomposition of the grid `d` into `size(rng)` pieces. This decomposition is encoded in the array `d` where `d(1,:)` holds the dimensions along which to split the grid and `d(2,:)` holds the corrsponding number of divisions. Using this information, the grid is decomposed by `vamp_fork_grid`. The HPF compiler will then distribute the `!hpf$ independent` loop among the processors. Finally, `vamp_join_grid` gathers the results.

```
subroutine vamp_sample_grid_hpf (rng, g, iterations, func)
  type(tao_random_state), dimension(:), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: iterations
  ⟨⟨ Interface declaration for func ⟩⟩
  type(vamp_grid), dimension(:), allocatable :: gs, gx
  !hpf$ processors p(number_of_processors())
  !hpf$ distribute gs(cyclic(1)) onto p
  integer, dimension(:,:), pointer :: d
  integer :: iteration, num_workers
  iterate: do iteration = 1, iterations
     call vamp_distribute_work (size (rng), vamp_rigid_divisions (g), d)
     num_workers = max (1, product (d(2,:)))
     if (num_workers > 1) then
        allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
        call vamp_create_empty_grid (gs)
        call vamp_fork_grid (g, gs, gx, d)
        !hpf$ independent
        do i = 1, num_workers
           call vamp_sample_grid0 (rng(i), gs(i), func)
        end do
        call vamp_join_grid (g, gs, gx, d)
        call vamp_delete_grid (gs)
        deallocate (gs, gx)
     else
        call vamp_sample_grid0 (rng(1), g, func)
     end if
     call vamp_refine_grid (g)
  end do iterate
end subroutine vamp_sample_grid_hpf
```

Since `vamp_sample_grid0` performes the bulk of the computation, an almost linear speedup with the number of processors can be achieved, if `vamp_distribute_work` finds a good decomposition of the grid. The version of `vamp_distribute_work` distributed with VAMP does a good job in most cases, but will not be able to use all processors if their number is a prime number larger than the number of divisions in the stratification grid. Therefore it can be beneficial to tune `vamp_distribute_work` to specific hardware. Furthermore, using a finer stratification grid can improve performance.

For definiteness, here is an example of how to set up the array of random number generators for HPF. Note that this simple seeding procedure only guarantees statistically independent sequences with Knuth's random number generator [12] and will fail with other approaches.

```
type(tao_random_state), dimension(:), allocatable :: rngs
!hpf$ processors p(number_of_processors())
!hpf$ distribute gs(cyclic(1)) onto p
integer :: i, seed
! ...
allocate (rngs(number_of_processors()))
seed = 42 !: can be read from a file, of course ...
!hpf$ independent
do i = 1, size (rngs)
   call tao_random_create (rngs(i), seed + i)
end do
! ...
call vamp_sample_grid_hpf (rngs, g, 6, func)
! ...
```

4.3 MPI

The MPI version is more low level, because we have to keep track of message passing ourselves. Note that we have made this synchronization points explicit with three `if ... then ... else ... end if` blocks: forking, sampling, and joining. These blocks could be merged (without any performance gain) at the expense of readability. We assume that rng has been initialized in each process such that the sequences are again statistically independent.

```
subroutine vamp_sample_grid_mpi (rng, g, iterations, func)
  type(tao_random_state), dimension(:), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: iterations
  ⟨⟨ Interface declaration for func ⟩⟩
  type(vamp_grid), dimension(:), allocatable :: gs, gx
  integer, dimension(:,:), pointer :: d
  integer :: num_proc, proc_id, iteration, num_workers
  call mpi90_size (num_proc)
  call mpi90_rank (proc_id)
  iterate: do iteration = 1, iterations
     if (proc_id == 0) then
        call vamp_distribute_work (num_proc, vamp_rigid_divisions (g), d)
        num_workers = max (1, product (d(2,:)))
     end if
     call mpi90_broadcast (num_workers, 0)
     if (proc_id == 0) then
        allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
        call vamp_create_empty_grid (gs)
        call vamp_fork_grid (g, gs, gx, d)
        do i = 2, num_workers
           call vamp_send_grid (gs(i), i-1, 0)
        end do
     else if (proc_id < num_workers) then
        call vamp_receive_grid (g, 0, 0)
     end if
     if (proc_id == 0) then
        if (num_workers > 1) then
           call vamp_sample_grid0 (rng, gs(1), func)
        else
           call vamp_sample_grid0 (rng, g, func)
        end if
     else if (proc_id < num_workers) then
        call vamp_sample_grid0 (rng, g, func)
     end if
     if (proc_id == 0) then
        do i = 2, num_workers
           call vamp_receive_grid (gs(i), i-1, 0)
        end do
        call vamp_join_grid (g, gs, gx, d)
        call vamp_delete_grid (gs)
```

```
      deallocate (gs, gx)
      call vamp_refine_grid (g)
   else if (proc_id < num_workers) then
      call vamp_send_grid (g, 0, 0)
   end if
 end do iterate
end subroutine vamp_sample_grid_mpi
```

A more complete version of this procedure is included with VAMP as well, this time as `vamp_sample_grid` in the MPI support module `vampi`.

5 Performance

6 Conclusions

References

[1] R. Krecker, Comp. Phys. Comm. **106**, 258 (1997).

[2] S. Veseli, Comp. Phys. Comm. **108**, 9 (1998).

[3] G. P. Lepage, J. Comp. Phys. **27**, 192 (1978); G. P. Lepage, Cornell Preprint, CLNS-80/447, March 1980.

[4] T. Ohl, *VAMP, Version 1.0: Vegas AMPlified: Anisotropy, Multi-channel sampling and Parallelization*, Preprint, Darmstadt University of Technology, 1998 (in preparation).

[5] T. Ohl, *Vegas Revisited: Adaptive Monte Carlo Integration Beyond Factorization*, hep-ph/9806432, Preprint IKDA 98/15, Darmstadt University of Technology, 1998.

[6] American National Standards Institute, *American National Standard Programming Languages FORTRAN, ANSI X3.9-1978,* New York, 1978.

[7] International Standards Organization, *ISO/IEC 1539:1991, Information technology — Programming Languages — Fortran,* Geneva, 1991.

[8] International Standards Organization, *ISO/IEC 1539:1997, Information technology — Programming Languages — Fortran,* Geneva, 1997.

[9] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.1*, Rice University, Houston, Texas, 1994.

[10] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Rice University, Houston, Texas, 1997.

[11] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, Technical Report CS-94230, University of Tennessee, Knoxville, Tennessee, 1994.

[12] D. E. Knuth, *Seminumerical Algorithms* (third edition), Vol. 2 of *The Art of Computer Programming*, (Addison-Wesley, 1997).

[13] R. Kleiss, R. Pittau, *Weight Optimization in Multichannel Monte Carlo,* Comp. Phys. Comm. **83**, 141 (1994).

[14] George Marsaglia, *The Marsaglia Random Number CD-ROM*, FSU, Dept. of Statistics and SCRI, 1996.