PET: A Parallel Performance Estimation Tool

Kattamuri Ekanadham, Vijay K. Naik, and Mark S. Squillante

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598.

Abstract: In this paper, we present a functional description of the *Performance Estimation Tool* (PET), which estimates the performance of parallel scientific applications portrayed in a simple language called *Portrayal Specification Language* (PSL). With relatively small effort, one can write PSL programs to express the execution rate determining segments, the control flow, and data-flow of an application as well as the parameters of a parallel execution environment. From PSL programs, PET estimates performance of applications written for existing and future architectures. We present some preliminary results that compare the performance predicted by PET with the timing results observed on IBM SP1.

1 Introduction

Performance analysis and estimation play a central role in the design and development of parallel application software. In parallel environments, the parameter space that affects program performance is much larger than that in the sequential case and hence both performance analysis and performance estimation tend to be more complex. As a result, simple "paper and pencil" analysis is not always a viable option, especially when the effects of scalability on performance are to be incorporated in the analysis. The alternative of simulating various choices is a time consuming process and, for many large applications, it may very well be impossible to accomplish this in a reasonable amount of time. In this paper, we describe a technique for the rapid performance estimation of parallel applications meant for execution on distributed as well as shared-memory architectures, without resorting to complex analysis or time-consuming simulations. Based on this technique, we have developed the *Performance Estimation Tool* (PET) that assists users in estimating the performance of parallel applications on existing or hypothetical parallel architectures.

The techniques employed in PET are developed based on the following observations. The performance of an application is governed by three factors: (i) the total amount of data manipulated and moved; (ii) the total amount of data computed; and (iii) the manner in which these two are interlaced. An algorithm for an application specifies all the operations to be performed as well as their interdependences. A program casts this information by mapping the entities computed in the application onto program variables (and hence onto storage). Nevertheless, with some moderate effort, it is possible to capture the principal quantities being computed and their relationship. (If the programs were expressed in a single assignment language, the storage interference is automatically avoided.) A dataflow graph contains precisely this information [1]. Superimposed on this graph, we have the information about the placement of data and computations on processors. With this one can completely determine the three key components cited above: total data movement, computation and interlacing.

In its current form, PET requires that the dataflow information and the placement choices be specified in the *Portrayal Specification Language* (**PSL**), which was introduced in [2]. The costs of computation of individual sequential segments can be measured by running them separately on a processor and included as parameters in a PSL program. The costs of communication are modeled by a simple set of parameters as described later. PET combines all this information and drives the data computations and data movements as concurrent events and estimates the total time taken to execute the application in parallel. No numerical computations intended in the original application are performed.

The organization of the rest of the paper is as follows. Section 2 summarizes the features of the PSL language. This is done by presenting the example of a Poisson Solver. Section 3 describes the results we obtained by running the example. In Section 4, we briefly describe some of the performance estimation tools that are discussed in literature. Section 5 closes with some concluding remarks.

2 Specification in PSL

In this section, we show an example of a PSL specification for a sample application. For a complete description of PSL, we refer interested readers to [2]. For this discussion, we consider a simple example of computing a discrete solution to the Poisson equation over a 2-dimensional domain with Dirichlet boundary conditions. The general schema is illustrated in Figure 1. The array u0 is initialized in the interior as well as at its boundary elements. Then alternatively, the arrays u1 and u0 are recomputed using point-Jacobi relaxation method. The interior points are computed using a 5-point stencil while the boundary points remain unchanged. The relaxation operation in each iteration is completely data parallel. Then an error norm is computed by traversing all the interior points of the computed array and this requires a global reduction operation.

```
program main
real u0(n,n), u1(n,n)

call InitInterior(u0)
call InitBoundary(u0,u1)

do iter=1,num_iters
   if odd iter then
       call Relax(u1,u0)
       call Error(u1)
   if even iter then
       call Relax(u0,u1)
       call Error(u0)
end do
end main
```

```
subroutine Relax(v,u)
real v(n,n), u(n,n)

for i ∈ {2,n-1} and j ∈ {2,n-1} do:
   v(i,j) = average of 5-points
              around u(i,j)
end for
end Relax

function Error(u)
real err = 0.0
for i ∈ {2,n-1} and j ∈ {2,n-1} do:
      err = err + norm(u(i,j))
end for
return err
```

FIGURE 1
Schema for a Poisson Solver

The PSL program for this application is illustrated in Figure 2. We first observe that in the application, a series of distinct matrices are produced and these are denoted by u0[0], u0[1],... each of which is an $n \times n$ matrix. In estimating performance, we are not really concerned with the actual values of these matrices, and these are all of type **var**. This is a predefined type and the corresponding variables contain the respective time of their computation. Notice that the subroutine structure remains the same, except that the routines now explicitly return new values that are created each time and all quantities are named distinctly.

The relax routine constructs a new matrix. Notice that the distribution of the matrix is specified by HPF-style directives. They are distributed block-wise over a processor grid. The body of the relax routine specifies how each matrix element is computed, using a case statement. The binding variable = **compute** (expression) **using** variable-list specifies that the variable on the left depends upon the list of variables in the **using** list and the cost of computing its value is given by the expression. Thus, on the boundary the values are simply copied from u(i,j) and its cost is characterized by the parameter no-ops, which can be appropriately set. Similarly, the interior points are computed from the stencil points as defined by the macro. The cost parameter for their cost is relax-op.

The reduction is specified by a built-in **reduce** clause. It takes two cost parameters: the first is the unit cost applied to perform local reduction of elements within a processor. The second is the unit cost applied to reduce all the local sums. The list of values reduced is explicitly specified.

```
#define stencil4(v,i,j) (v[i+1,j], \
        v[i-1,j], v[i,j+1], v[i,j-1])

defchannel fixsend(C_fs),fixrecv(C_fr),
    varsend(C_vs),varrecv(C_vc),
    accum(B_sz),xmit(C_tr)::default;

def main ()= {
  var ::u00;
  var ::u0[num_iters+1], r[num_iters+1];

  u00 = InitInterior();
  u0[0] = InitBoundary(u00);

  (i=1:num_iters) {
    u0[i] = Relax(u0[i-1],u0[0]);
    r[i] = red(u0[i]);
    };
  return u0[num_iters];
  }

def Relax(v,u) = {
  var ::u,v;
  var dimension(n,n),processors(npx,npy),
      distribute(block,block),elemsize(8)::w;
  (i=1:n, j=1:n) {
    w[i,j] = case {
      (i ==1 || j ==1 || i==n || j ==n):
        compute (no_ops) using u[i,j];
      default:  compute (1*relax_op)
                using stencil4(v,i,j);
      }
    };
  return w;
  }

def red(u) = {
  var ::u;
  var dimension(1), distribute(block),
      processors(1)::r;
  r[1] = reduce (cmperr_op, gl_reduce_op)
        u[i=2:n-1,j=2:n-1];
  return r;
  }
```

FIGURE 2

Listing of a PSL program corresponding to the Poisson solver

The communication cost is modeled by the parameters specified in the **defchannel** declaration. The cost incurred in sending B bytes in a message is modeled by, $C_{fs} + C_{vs} \times B_{msg}$ and the cost incurred in receiving the same message (after any wait involved prior to the arrival of the message) is modeled by $C_{fr} + C_{vr} \times B_{msg}$, where C_{fs} and C_{fr} are, respectively, the fixed costs of sending and receiving a message and C_{vs} and C_{vr} are the per byte variable costs in sending and receiving data, respectively. These are specified as the expressions associated with the keywords **fixsend**, **fixrecv** etc. in the **defchannel** declaration. Furthermore, we assumed that the communication is done in packets whose size is specified by the parameter **accum**.

Thus, the structure of the PSL program resembles the application program and actual computation steps are replaced by the cost expressions. For brevity, we have omitted the InitInterior and InitBoundary routines in the listing shown in Figure 2.

3 Experimental Results

We have implemented a prototype version of PET that incorporates all the features described above. Currently, we are in the process of testing and validating this implementation by estimating performance of a variety of scientific applications on existing architectures. In this section, we present some preliminary results to indicate the level of accuracy we are able to achieve in the estimated performance.

We implemented a distributed memory version of the Poisson Solver described in the preceding

Number of Processors	1	2	4	8	16
Observed time per iteration (μsec)	16,423	9,126	5,578	3,940	2,880
Estimated time per iteration (μsec)	16,406	9,182	5,601	3,856	2,875

TABLE 1
Observed performance on SP1 and the performance estimated by PET for the Poisson solver

section, using simple block distribution of the solution vector arrays, on IBM SP1 and carried out performance measurements on 1 through 16 processors. We then used PET to estimate the performance by running the corresponding PSL program shown in Figure 2. The parameters input to this code are the problem size (`n`), number of iterations (`num_iters`), data partitioning parameters, and the communication and computation rate parameters. Using a sequential version of the Poisson solver, we parameterized the cost per iterate in applying the relaxation operator (i.e., cost of computing `v(i,j)`) as well as the cost per iterate in computing the error norm. We determined these parameters for SP1 over a range of message sizes that are relevant to the problem considered. Similarly we measured the communication parameters described earlier. Both the computation and communication parameters were supplied as input values in the PSL program. Using PET, we estimated the cost per iteration of the Poisson solver. The estimated performance figures for a 258×258 grid are tabulated in Table 1, for 1 through 16 processors. Also listed in that table are the execution time as observed on SP1 over the same range of processors. Note that, except for the 8 processor case, the estimated performance in each case is within 1% of the actual observed performance. For the 8 processor case, the estimate is within 3% of the observation.

4 Related Work

Numerous efforts have gone into developing tools to help users understand the performance of their parallel applications. Typically these tools tend to be used at run-time or as post-processors [4, 5, 6, 8, 9, 10]. In these approaches, performance data is gathered by monitoring the program execution and/or by collecting execution trace. These tools serve a useful purpose as performance tuning aids *after* an application has been parallelized. As such, these tools are not directly useful in designing and developing parallel applications.

In [3], the authors report a performance prediction tool that automatically derives performance estimates for SPMD parallel Fortran77 programs based on distributed memory systems. The performance prediction system reported in [7] has a similar approach. There, an application behavior is described in a language, called BDL, and an event-driven simulator predicts the performance by simulating the specified behavior. The computation is modeled by supplying execution times of sequential code segments and the communication architecture is modeled by supplying parameters of latency and bandwidth. Our approach resembles the above in spirit, but differs substantially in the level of abstraction and detail. The distinctive feature of our approach is that our specification bears close resemblance to a sequential program written for a shared memory model. Variables and dependences are accurately expressed as in the algorithm. Distribution of data is expressed in a manner very similar to HPF directives. We model the data dependences and flow of data very accurately. The cost of data movement is modeled in a variety of ways by specifying the channel characteristics. Furthermore, the specification is done from a sequential algorithm and no additional work is involved – that is one does not have to specify a parallel version with appropriate communications and synchronizations (which is necessary in an SPMD style programming). In contrast, we do not model the network routing or cache effects as in the above approaches. Also, we do some speculation on the local scheduling of tasks on a processor and we provide some minimal controls to regulate this. It is our belief that our approach is more flexible and is more amenable for automatic derivation from a sequential program in the near future.

5 Conclusions

In this paper, we have described the design of a tool called PET to meet two essential needs in performance estimation of parallel programs: (1) modeling the parallel implementation and the associated parallel environment should not be time consuming, and (2) the time required for the performance estimation should be relatively small as compared to the development and execution on the actual application on a real parallel system. PET essentially consists of a compiler and a run-time system that will execute programs written in a pseudo-programming language called PSL. Using first order approximations to model the architectural and execution costs, one can compare a large number of alternatives rapidly and with reasonable accuracy.

An advantage of our approach is the flexibility in choosing the level of detail – one can collapse uninteresting segments of computations or elaborate the details of computation and communication at a very fine granularity of individual elements. Thus, it renders an incremental approach to refine the modeling process and eventually it can even be used to derive the actual implementation from the ultimate PSL program. We see this approach to be most useful in developing scalable parallel implementations where more than one parallel algorithm may have to be considered to develop software that can run efficiently on more than one parallel system. Sensitivity analysis can be done by varying the data distributions, network and processor speed characteristics.

Furthermore, our approach renders a simple method to represent relatively complex applications. The representation bears strong resemblance to a sequential program and has a good chance for derivation from high-level mathematical description of the application. There is virtually no effort spent to parallelize the application in order to estimate performance. Instead, a few parameters enable quick estimates of the effects of various alternative choices.

References

[1] Arvind and K. Ekanadham. *Future Scientific Programming on Parallel Machines,* Journal of Parallel and Distributed Computing, vol. 5, pp. 460-493, 1988.

[2] K. Ekanadham and V. Naik, Portrayal of Parallel Applications for Performance Evaluation and Prediction. *To appear* in the Proceedings of the 28th Hawaii International Conference on System Sciences, 1995.

[3] T. Fahringer, R. Blasko, and H. Zima, Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. In *Proceedings of International Conference on Supercomputing*, 1992.

[4] D. Gannon, J. Lee Shei, B. Sarukkai, S. Narayana, et al. SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems. IFIP Trans. A, Computer Science Technology, Vol. A 11, 1992.

[5] J. Hollingsworth and B. Miller, Parallel program performance metrics: A comparison and validation. In *Supercomputing'92 Proceedings*, 1992.

[6] J. Hollingsworth and B. Miller, Dynamic Control of performance monitoring on larger scale parallel systems. In *Proceedings of International Conference on Supercomputing*, 1993.

[7] P. Mehra, C. Schulbach, and J. Yan, A Comparison of Two Model-Based Performance-Prediction Techniques for Message-Passing Parallel Programs. In *Proceedings of SIGMETRICS'94*, 1994.

[8] D. Reed, *et al*, The Pablo Performance Analysis Environment, Technical Report, University of Illinois, 1992.

[9] D. Rover, A. Waheed, and M. Doetsch, Advanced methods of performance data processing and analysis. Technical Report TR-MSU-EE-SCSL-012-92, Michigan State University, 1992.

[10] S. Sarukkai and D. Gannon. *SIEVE: A performance debugging environment for parallel programs.* J. Parallel and Distributed Computing, Vol. 18, pp. 147–168, 1993.