Using Predictive Adaptive Parallelism to Address Portability and Irregularity

David L. Wangerin and Isaac D. Scherson
{dwangeri,isaac}@uci.edu
School of Computer Science
University of California, Irvine
Irvine, CA, USA 92697-3425

Abstract

A semi-dynamic system is presented that is capable of predicting the performance of parallel programs at runtime. The functionality given by the system allows for efficient handling of portability and irregularity of parallel programs. Two forms of parallelism are addressed: loop level parallelism and task level parallelism.

1. Introduction

Parallel processing poses two distinct yet related problems for programmers and system developers: portability and irregularity. Portability is the ability of a program to execute quickly and efficiently on a multitude of different parallel platforms[1]. Irregularity is a quality of parallel programs where the resource requirements of the program are difficult to determine and thus efficient resource allocation is difficult. As will be demonstrated, both of these issues are related and can be addressed with a combination of compile-time and runtime components in an Adaptive Parallelism system.

The fundamental problem of creating portable parallel programs is that system resources, both in terms of numbers and capabilities, are unknown until execution time. Likewise, the fundamental problem of handling irregularity is that the resource requirements of the program are unknown until execution time. The symmetry of these problems lead to a common solution in the form of dynamic resource allocation.

Resource allocation issues from both portability and irregularity can be addressed through the use of an Adaptive Parallelism system. Adaptive parallelism is a field of study where programs perform runtime modifications in response to changes in the program or machine state [2][7][8]. This work extends the functionality of an adaptive parallelism system through the inclusion of a predictive execution time estimator, which guides the use of parallelism in a program. Since the adaptive parallelism system allocates resources dynamically at runtime, the problems of portability and irregularity are addressed simultaneously.

For this paper, the following terms are defined. A thread is any type of parallel workload and does not denote a specific implementation (e.g. task, process, lightweight thread) or invocation method (e.g. fork, spawn, FORALL). A processing element is a single processor in the parallel system and does not imply any specific architecture or set of capabilities for the processor.

The system is targeted for multi-programmed SPMD (Single Program Multiple Data) or MIMD (Multiple Instruction Multiple Data) homogeneous systems. No assumptions are made about the network topology, the system architecture, or the other programs running concurrently on the system. The methods will work equally well on adhoc clusters, server farms, and structured parallel systems. Although the proposed work does not cover heterogeneous systems, the methods could easily be extended to apply to grids, networks of workstations, and other mixed environments.

2 Solution Overview

The goal of parallel processing is to use as many resources as possible to minimize the execution time of a program. The computation time of a parallel program is inversely proportional to the number of processing elements assigned to the program. It is obvious that under-utilization of resources is not desirable. However, it is well established that using too many threads will slow down the execution of a parallel program. The use of parallel resources creates overhead from thread creation, workload migration, and inter-thread communications. Therefore, any portable system will have to find a balance between the computational

[1]There are several other aspects of portability that are not addressed in this paper, such as creating cross-platform executable file formats and defining cross-platform programming interfaces; these concerns are orthogonal to the issue of optimizing performance

speedup and the overhead of using parallel resources. This balance, called the optimal number of threads, will yield the minimal execution time of the program.

In order to determine the optimal number of threads, two pieces of data are needed: the capabilities of the system resources and the resource requirements of the program. The resource capability measurements must accurately reflect the delivered performance of processors and networks for a variety of different programs. This problem is known as the benchmarking problem for machine characterization and will addressed in Section 3. The resource requirement descriptions must be machine-independent and be able to adapt to data sets changing at runtime. The building of resource requirement descriptions and their on-line use is addressed in Section 4.

2.1 Adaptive Parallelism Overview

The problems of portability and irregularity can be addressed simultaneously in an on-line system. The system can be informally described as follows: When the program is compiled, the compiler inserts *cost functions* and *behavior models* that describe the resource requirements of each parallel section of code. When the program is submitted to the system, the *runtime system* translates the programs cost functions into estimated execution times. At runtime, when the program has the opportunity to create new threads, the behavior models are used to estimate the optimal number of threads under the current state of the program.

The program invokes the optimality process whenever it is possible to create new threads. This addresses both portability and irregularity, as a program can adapt to effectively use an unknown machine and can handle unpredictable and dynamically changing workloads.

The solution is a fully predictive system, meaning that no performance profiling, analyzing, or post-runtime modification is done. This is a distinguishing characteristic of the presented system, as all previous similar systems rely on executing a program at least once, analyzing the performance, and then optimizing the program.

3 Machine Characterization

The goal of machine characterization is to develop a set of instruction timings that describe the performance of a processor. The timings must reflect delivered performance, not peak performance, and should have a fairly coarse granularity. Since the adaptive parallelism system relies on accurate estimations of thread execution times, it is essential that the instruction timings be applicable to programs beyond the ones used to generate the timing information.

Performance Vectors [5][4] provide a method for quickly and accurately evaluating hardware performance. Performance vectors describe the average execution time of instruction classes. While a detailed understanding of the creation of performance vectors is not necessary for understanding the methods involved in this paper, it is useful to know their capabilities, limitations, and intended uses.

A performance vector is defined as a vector where each element represents the average execution time of an instruction class. An example performance vector characterizing microprocessor performance is:

$$\begin{bmatrix} \texttt{avg. time for ALU op.} \\ \texttt{avg. time for FP op.} \\ \texttt{avg. time for Mem op.} \\ \texttt{avg. time for other op.} \end{bmatrix} = \begin{bmatrix} x_{\texttt{ALU}} \\ x_{\texttt{FP}} \\ x_{\texttt{Mem}} \\ x_{\texttt{Other}} \end{bmatrix}$$

The data used to generate performance vectors is gathered by running a set of benchmarks that represent real-world applications. Counters and other non-intrusive techniques collect timing information and instruction counts. If the benchmark suite consists of *m* benchmarks with *n* instruction types being measured, then the execution times of the benchmarks form a vector $\mathbf{t} = [t_1, t_2, \cdots, t_m]^\mathsf{T}$ and the instruction counts form a matrix $\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$. When $\mathbf{A}$ and $\mathbf{t}$ are placed in a system $\mathbf{A}x \sim \mathbf{t}$, then the solution to the system, x, represents the performance vector for the benchmark set. Since the benchmarks are not synthetic programs and the system is measuring the execution times of instructions, the performance vector yields the delivered execution times of instructions under real-world conditions.

It is important to note that the system $\mathbf{A}x \sim \mathbf{t}$ is an inconsistent overdetermined system and does not have a trivial solution. The mathematics used to solve the system and produce the performance vector is beyond the scope of this paper. The reader is referred to the literature [4] for a full discussion of the mathematics used and their proofs.

While performance vectors were designed to measure the computational performance of a machine, they may also be adapted to reflect the communication times, or network performance, of a machine. Assuming that network performance is static, communication times may be measured as an instruction class of a performance vector.

Since performance vectors are generated from benchmark programs, it is necessary to ensure that the benchmark set is comprised of programs that are similar to the programs that they will be used for estimating execution times. For example, a benchmark set of parallel Navier-Stokes solvers would likely generate a performance vector that would be useful for predicting the performance of other parallel Navier-Stokes programs, but may not be useful for predicting the performance of graphics rendering or other categories of programs. Therefore, it is necessary to define a benchmark set that reflects and represents the expected mix of programs that will be executed on the adaptive parallelism system. For the purposes of this paper, assume that the benchmark set is compliant with the programs being analyzed.

Performance vectors are developed for a machine only once. The values are static and will not change unless the benchmark set is changed. Therefore, the benchmarks are only run a single time and the performance vectors are calculated only once. The performance vector values are passed to the runtime system (section 5) at system initialization.

4 Program Resource Requirements

Determining the resource requirements of a program is a complex problem that requires a two-phase approach. Since the exact control flow of a program cannot be determined without knowing the data set, and the data set is unknown until runtime, the presented solution is to have the compiler insert *cost functions* and *behavior models* into a program at compile time and then use them in conjunction with the data set at runtime.

A *cost function* is a description of the instructions that comprise a program segment. It is similar to a performance vector in that it is a vector corresponding to instruction classes, but instead of describing timing values it describes the number of instructions. The cost function is built at compile time and can be constructed from the compilers basic blocks. One cost function is generated per basic block, i.e. there is a one-to-one relation between cost functions and basic blocks. Cost functions do not contain any control-flow data; they are only used for describing the instruction contents of a program segment.

For example, the following program segment:

```
for i = 1 to N:
   A[i] = (A[i-1] + A[i+1]) / 2
```

contains two basic blocks: the loop header and the loop body. Assuming that the performance vectors have defined the following instruction classes, the loop header block has a cost function corresponding to a single index increment (arithmetic) operation and a single branch (other) operation. The body block has a cost function corresponding to two array index plus two math (arithmetic) operations and three array lookup (memory) operations.

$$\begin{array}{ccc} \textbf{definition} & \textbf{header} & \textbf{body} \\ \begin{bmatrix}\texttt{Arithmetic op.}\\ \texttt{Memory op.}\\ \texttt{Other op.}\end{bmatrix} & \begin{bmatrix}1\\0\\1\end{bmatrix} & \begin{bmatrix}4\\3\\0\end{bmatrix}\end{array}$$

Cost functions are constructed for all basic blocks in parallel segments of the program (sequential segments do not need cost functions because they play no role in determining how to use parallelism).

A *behavior model* is a compile-time description of the control flow of a thread. In particular, behavior models are used to determine the number of times each basic block will execute. If a control flow structure is based upon a variable of unknown value then the model uses a symbolic representation, called a *base metric*. The base metric is a value (usually data set size) that can be determined at runtime. It is permissible to nest loops with base metrics.

Two general categories of behavior models are defined for programs using loop-level parallelism and task-level parallelism.

4.1 Loop level Parallelism

One of the most common methods for expressing parallelism in programs is through parallel loop structures, such as Fortan's `FORALL` statement, called loop-level parallelism. Loop-level parallelism is typically found in scientific applications involving vector and matrix operations.

A useful property of loop-level parallelism is that the compiler can decide how the iterations of the loop body are divided. If the compiler divides the workload into large blocks (i.e. large computational load per iteration), the parallelism is classified as coarse grain parallelism. Conversely, if the compiler divides the workload into small block with a light workload, it is classified as fine grain parallelism.

The decision on whether to use coarse or fine grain parallelism is dependent on the target architecture, specifically the ratio of performance between the processing elements and the processor interconnection network. A relatively fast interconnection network will profit from using fine grain parallelism while a relatively slow interconnection network will need coarse grain parallelism to be effective. The reason for this is that the use of parallelism incurs an overhead from thread creation, migration, and interthread communication.

The execution time of a parallel thread can be described by the execution time of its component parts. For a program that contains a loop that can be parallelized, the time cost of executing the loop is equivalent to the number of iterations that the loop will execute times the time cost of executing a single iteration of the loop body. Parallelizing the loop effectively divides the number of iterations that the loop will execute by the number of threads. The overhead of using parallelism can be described as a function of the time cost of communications per loop iteration and the time cost of creating and distributing threads.

From these components, a general parallel loop can be described in terms of its runtime costs by the following equation:

$$F(t) = \frac{A}{t}(B + C) + Dt \tag{1}$$
$$= ABt^{-1} + ACt^{-1} + Dt \tag{2}$$

where:

t	data set size
A	total number of iterations
B	computation cost per data element
C	communication cost per iteration
D	distribution cost per thread

Since the objective is to minimize the execution time of the program, the derivative of the time function is calculated and equated to zero. The solution of the system represents the number of threads that will yield the minimal execution time.

$$F'(t) = -ABt^{-2} - ACt^{-2} + D \tag{3}$$
$$0 = -ABt^{-2} - ACt^{-2} + D \tag{4}$$
$$D = ABt^{-2} + ACt^{-2} \tag{5}$$
$$Dt^2 = AB + AC \tag{6}$$
$$t^2 = \frac{AB + AC}{D} \tag{7}$$
$$t = \sqrt{\frac{AB + AC}{D}} \tag{8}$$

The behavior model is equation 8.

For nested loops where the number of iterations of each loop is a function of the number of threads, more complex equations are needed to describe the execution time. Assume that the number of iterations of each loop is divided evenly in two dimensions. The time cost function for two nested loops then becomes:

$$F(t) = \frac{A_2}{\sqrt{t}}\left(B_2 + C_2 + \frac{A_1}{\sqrt{t}}(B_1 + C_1)\right) + Dt \tag{9}$$
$$= A_2 t^{\frac{-1}{2}}(B_2 + C_2) + A_2 A_1 t^{-1}(B_1 + C_1) + Dt \tag{10}$$

Taking the derivative of the equation, setting it equal to zero, and solving would work, but the intermediate equations are not as easy to work with as in the first system. By substituting x for $\sqrt{t}$, the equation can be simplified:

$$F(x) = A_2 x^{-1}(B_2 + C_2) + A_2 A_1 x^{-2}(B_1 + C_1) + Dx^2 \tag{11}$$
$$F'(x) = -A_2 x^{-2}(B_2 + C_2) - 2A_2 A_1 x^{-3}(B_1 + C_1) + 2Dx \tag{12}$$
$$0 = 2Dx^4 - A_2(B_2 + C_2)x - 2A_2 A_1(B_1 + C_1) \tag{13}$$

Since x is not easily factored out of the equation, a quartic equation is needed to find a solution, which results in four roots. Given that all of the parameters are non-negative real numbers, only one root will be non-negative real. While the full equation for solving quartic systems is large and complex (see [9] for the full quartic equation), it can be reduced to the following form to give the non-negative real root:

$$y = \sqrt[3]{\frac{d^2}{2} + \sqrt{\left(\frac{-4e}{3}\right)^3 + \frac{d^4}{4}}} + \sqrt[3]{\frac{d^2}{2} - \sqrt{\left(\frac{-4e}{3}\right)^3 + \frac{d^4}{4}}} \tag{14}$$
$$r = \frac{1}{2}\sqrt{y} + \frac{1}{2}\sqrt{\frac{-2d}{\sqrt{y}} - y} \tag{15}$$

where:

$$d = \frac{-A_2(B_2 + C_2)}{2D} \tag{16}$$
$$e = \frac{-A_2 A_1(B_1 + B_2)}{D} \tag{17}$$

4.2 Task-level Parallelism

Another common method of expressing parallelism is through explicitly parallel sections, such as fork and join statements or spawn and sync statements, called task-level parallelism. Task-level parallelism often does not have the nice structure of loop-level parallelism where the compiler can decide upon the division of workloads. Instead, the compiler can only decide if the data set is sufficiently large to warrant the creation of a thread. Task-level parallelism is commonly found in n-body problems and applications utilizing clustering methods, where data structure sizes can vary depending on the data values.

In general, parallelism will lower the execution time of a program only if the computation time saved by using parallelism is less than the additional communication and creation time incurred by using parallelism. Therefore, it is necessary to determine when parallel computational speedups are greater than parallel overhead. For task-level parallel threads where execution time is a function of its data set size, a minimum data set size, or cutoff point, can be defined where only threads over the cutoff point should be created.

The cutoff point can be established by equating the computation time cost to the thread creation and migration time cost. Note that the computation cost may have two sections: one governed by the data set size and one of a fixed size (such as initialization code). The equality is given:

$$D = Bn + F \tag{18}$$

$$n = \frac{D - F}{B} \tag{19}$$

where:

n	data set size
B	computation cost per data element
D	thread creation and migration cost
E	fixed communication cost per thread

The behavior model is equation 19.

Note that cost functions and behavior models are only built for a programs threads or parallel sections. Since they are used for predicting the execution time of threads, they are unnecessary for the programs serial sections.

5 Adaptive Parallelism Runtime System

The runtime system is responsible for tying together the information from the program and the system to calculate the optimal number of threads for the program. The program invokes the runtime system whenever it has the possibility of using parallelism. The cost function is passed from the program and the performance vectors are known by the system. The runtime system solves the cost function and behavior model of the thread and returns the optimal number of threads.

Recall that a performance vector P describes the average execution times of instructions. A cost function C describes the number of instructions in a section of a program. Assume that the instruction classes used in both the performance vector and the cost function are the same (e.g. they are of the same granularity and ordering). To estimate the execution time:

$$P \cdot C^{\mathsf{T}} = \begin{bmatrix} \text{avg. time for op } O_1 \\ \text{avg. time for op } O_2 \\ \vdots \\ \text{avg. time for op } O_n \end{bmatrix} \cdot \begin{bmatrix} \text{op } O_1 \text{ instances} \\ \text{op } O_2 \text{ instances} \\ \vdots \\ \text{op } O_n \text{ instances} \end{bmatrix}^{\mathsf{T}}$$
$$= \begin{bmatrix} p_1 & p_2 & \dots & p_n \end{bmatrix}^{\mathsf{T}} \cdot \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}$$

Using the standard dot product of vectors:

$$P \cdot C^{\mathsf{T}} = p_1c_1 + p_2c_2 + \cdots + p_nc_n = \mathbf{t} \tag{20}$$

Thus the expected execution time for a section of a program on a given machine can be predicted. Since performance vectors for a given system are static, this method can be applied to all of the cost functions of a program before program execution (e.g. at installation or load time).

At runtime, when the base metrics for behavior models are known, the equations from section 4 can be solved to yield the optimal number of threads for that section of the program. The calculation can be repeated whenever there is a change in the base metric values or in the underlying system (e.g. processing elements being added or removed).

The workings of the adaptive system are best shown through an example. Consider the following Fortran code for performing a Jacobi relaxation in parallel:

```
REAL A (50, 50)
DO k = 1, 3
   FORALL (i=2:49, j=2:49)
      A[i] = (A[i,j-1] + A[i,j+1] &
         + A[i-1,j] + A[i+1,j]) / 4
   END FORALL
END DO
```

The code is comprised of three loops, two of which are base metric loops. For simplicity, assume that the number of iterations for the i and j loops is equivalent to the size of the matrix dimension. Further assume that sharing of boundary values occurs once per iteration of the outermost loop, rather than once per iteration of the innermost loop. This gives the following cost functions, which are created at compile time:

Loop	1	2	3
arithmetic	8	0	0
memory	5	0	0
communication	0	0	4
iterations	0	0	3
base metric	50	50	0

The target machine has the characteristics of a fast modern processor (equivalent to a 1GHz processor) with an extremely fast interconnection network that has static performance. All performance vector values are in units of nanoseconds. The following performance vector describes the machine:

arithmetic	1
memory	10
communication	500
other	2

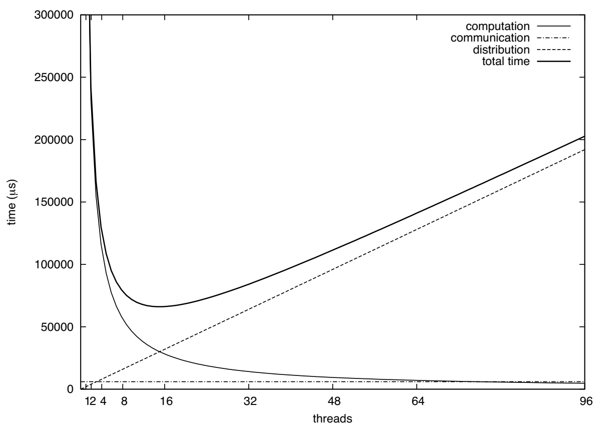

Figure 1. Timing components of the Jacobi Relaxation program segment

When the program is loaded onto the machine, the cost function values can be converted into timing values:

Loop	1	2	3
computation	60	2	2
communication	0	0	2000
iterations	0	0	3
base metric	50	50	0

By using the cost function values into the basic time equation from section 4.1, Figure 1 displays the execution time of the program under different numbers of threads. Solving the derivative function from section 4.1 gives the optimal number of matrix divisions as 3.87, which yields 14.99 as the optimal number of threads.

6 Prior Work

Much work has been done on creating parallel systems that adapt to runtime conditions. Piranha [2] uses a work-stealing load balancer to distribute workloads over a dynamically changing system. Adaptive Multiblock PARTI [3] provides support for repartitioning data on machines where the number of processors may change at runtime. SUIF-Adapt [6] addresses issues of optimizing parallelism by recording the execution times of threads and guiding future choices on data distribution by the history of past performance. Cilk [1] attempts to guarantee that newly created threads will improve the execution time of a program. The system relies on profiling the program to identify the critical path (and its execution time) of a thread.

7 Conclusions

The methods laid out in this paper form the start of a framework for automatically configuring programs to efficiently run on diverse architectures. The cost functions and behavior models are machine independent and applicable to a range of different architectures, allowing for portability of parallel programs. Since the methods are invoked at runtime and adapt to changing data sets, irregularity can be handled efficiently.

However, the methods detailed in this paper operate under certain requirements for the form and format of parallelism. Future work includes relaxing the requirements to allow for the handling of conditionals within parallel code segments, applying the methods to pipelined parallelism, and uses of predictive adaptive parallelism in systems where workload distribution and thread creation are not large cost factors.

References

[1] R. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Network of Workstations. In *USENIX 1997 Annual Technical Symposium*, pages 133–147, Jan. 1997.

[2] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, Jan. 1995.

[3] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and Runtime Support for Programming in Adaptive Parallel Environments. *Scientific Programming*, pages 215–227, Jan. 1997.

[4] U. Krishnaswamy. *Computer Evaluation Using Performance Vectors*. PhD thesis, University of California, Irvine, 1995.

[5] U. Krishnaswamy and I. D. Scherson. A Framework for Computer Performance Evaluation Using Benchmark Sets. *IEEE Transactions on Computers*, 49(12):1325–1338, Dec. 2000.

[6] D. K. Lowenthal, G. S. Howard, D. G. Morris, D. B. Weatherly, and F. Lowenthal. SUIF-Adapt: An Integrated Compiler/Run-Time System for Global and Dynamic Data Distributions. Technical report, University of Georgia, 2004.

[7] M. Martonosi and M. W. Hall. Adaptive Parallelism in Compiler-Parallelized Code. *Concurrency: Practice and Experience*, 10(14), 1998.

[8] A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent Adaptive Parallelism on NOWs using OpenMP. In *7th Conference on Principles and Practice of Parallel Programming*, pages 96–106, May 1999.

[9] E. W. Weisstein. Quartic Equation. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/QuarticEquation.html.