Resource Estimation for Parallel Architectures with Distributed Processor/Memory Nodes*

M. A. Thornton and D. L. Andrews

Department of Computer Systems Engineering, University of Arkansas, Fayetteville, USA

Determining the resources needed to run a specific program is an important task for static task schedulers for existing multiprocessors. It can also be a valuable computer-aided engineering tool for the design and implementation of application specific parallel processors. An approach for determining the required number of processors and the amount of memory needed per processor is described. The estimates are calculated using information available in a data-flow graph generated by a high-level language compiler. Metrics based on the notions of thread spawning and maximum length thread probability density functions are presented. The measures obtained from the parallelism profiles are used as input to a queuing system model to predict the number of processing elements that can be exploited. Memory resource estimates are predicted through a simple graph traversal technique. Finally, experimental results are given to evaluate the methods.

1.0. Introduction

Processor execution speeds are increasing dramatically due to advancements in integrated circuit manufacturing technology and engineering design methods and tools. However, the rate of speed increase for memory is much smaller. Currently, memory access versus processor latency is at a ratio of approximately 10:1. It is predicted that this ratio will increase to 100:1 in 20 years, a problem known as the processor-memory performance gap [1]. This implies that future architectures must efficiently deal with memory latencies in order to continue to provide machines with performance increases that have been common in the past.

A result of the processor-memory performance gap is that new architectural approaches for designing and implementing multiprocessor computer systems must be utilized. A common theme among several of the various approaches is to distribute and integrate memory with each processor to reduce bus contention, thus allowing concurrent local memory accesses. This leads to the question of determining the amount of memory needed for each processor. Too much memory results in a waste of resources in terms of available chip area and power consumption, while too little memory could impact program runtime [1]. For these reasons an analytical tool for estimating the number of processor/memory resources for given application programs has been developed.

Histograms of the theoretical maximum number of instructions that may execute versus cumulative runtime (referred to as 'parallelism profiles') have been used in the past to evaluate the available parallelism contained within an application program [2] [3]. The profiles indicate the number of parallel operations available at each step of program execution. We present a model derived from parallelism profiles for the analysis of the overhead time associated with the creation and completion of parallel operations. The model presents a method for determining the overhead associated with executing the program. The model results may then be used to determine the granularity of parallel operations within a program, partitioning, and load balancing, or determining optimal thread sizes for multithreaded architectures.

* This research was supported in part by a grant from the Arkansas Science and Technology Authority under contract 97–B–12.

Unfortunately, parallelism profiles only give the upper bound in achievable parallelism for any given architecture. Multiprocessor designers typically specify a system using a behavioral or structural description of the hardware and simulate the execution of benchmark programs to determine the behavior of the architecture. The hardware simulation results can then be compared against the ideal case given in the parallelism plot for performance analysis of the design. In this work, we define several parameters that are directly measurable from the parallelism plots and develop a statistical queuing model for execution of the program independent of architectural details. The statistical model results can then be used to guide the designer in determining the hardware organization. Since the statistical model can also be used to generate ideal cases, model validation is accomplished by comparing predicted results to the original ideal data.

Most stochastic models for multiprocessor systems only predict steady state responses by ignoring start-up and shut-down transients since they are typically too hard to model. In this work, we include the transients in the model through the notions of *maximal thread length* and *thread spawning* probability density functions (pdf). The pdfs are generated directly from the available parallelism curves and are used to pseudo-randomly generate random variables that represent the initiation and duration of *maximal length* computation threads.

The criterion for determining when to halt the statistical models' execution proved to be a crucial parameter with regard to the accuracy of the results. In the work described here, the halting criterion is based on equating the amount of work present in the available parallelism curve to that predicted by the simulation. When the maximum work criterion is used, estimates of total runtime and maximum required processing elements are predicted and compared to the ideal case. This validates the approach for estimating the results due to non-zero processor communication latencies or restricting the number of available processing elements in other experiments using the model.

Traditionally, memory requirements have been obtained by: 1) assuming an architectural organization, 2) building or simulating the architecture, and 3) executing an application on the architecture and monitoring memory usage. This approach is disadvantageous for the designer, since the design must be in place before the memory requirements can be found. Other methods for modeling and predicting memory include non-deterministic statistical models [4] [5] which unfortunately still assume some type of architecture. The work described in [6] is similar to that described here, except that register estimation in a high-level synthesis framework is accomplished, not memory estimation for application specific programs. The method discussed here has novelty in that the only requirement is a representation of the application program itself.

The organization of the paper consists of the following section that briefly reviews the notions of data-flow graphs and available parallelism curves. Next, some metrics and definitions are provided that are used in the formulation of the model. Sections 4 and 5 contain the descriptions of the number of processors and memory usage models respectively, including experimental results. Finally, conclusions and future work section is included.

2.0. Data-Flow Graphs and Associated Metrics

Any given application program can be viewed as a collection of sequential instruction threads to be executed as soon as their input data is available. An abstract representation of a program is then a directed acyclic graph where vertices correspond to computational instructions to be executed and edges represent data dependencies. The edges of the data dependency graph correspond to the transfer of data from the output of a producer instruction node to a consumer instruction node. In a multiprocessor system, each thread can be executed on a single processor; therefore, threads can execute concurrently as long as the required input data are available. With this viewpoint, representing an application program as a data dependency graph allows us to exploit the available parallelism.

As an example, Figure 1 shows a data dependency graph which computes a value from the formula $c = (1/a^2) + b^2 - (b + a) - 1$. The graph shows the data dependencies inherent in the computation. For example, node 2 cannot

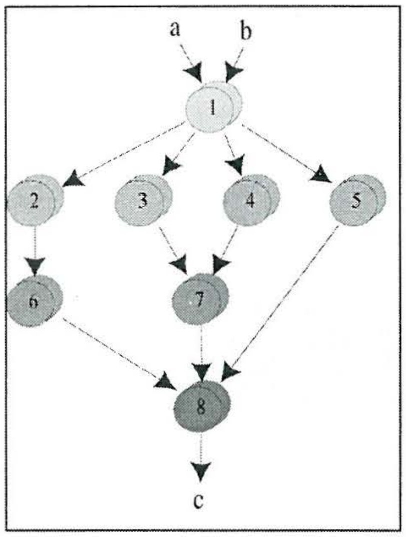


Fig. 1. Data dependency graph of $c = (1/a^2) + b^2 - (b + a) - 1$.

execute until it receives the value of a from node 1. When node 2 executes, it will produce the value of a^2. Node 2 then passes this value to node 6, and so on. The graph also shows the available parallelism in the computation. For example, given sufficient resources, nodes 2, 3, 4, and 5 can be executed in parallel depending only on the results of node 1. Table 1 summarizes the operations performed by each data dependency graph node.

Thread	Operation	Result
1	Retrieve	a, b
2	Square	a^2
3	Subtract	$-1-b$
4	Square	b^2
5	Negate	$-a$
6	Inverse	$1/a^2$
7	Add	$b^2+(-1-b)$
8	Add	$(1/a^2)+b2-(b+a)-1$

Table 1. Instruction Thread Operations for Each Vertex in Graph of Figure 1

Parallelism profiles present a graphical representation of the parallel operations available for execution at each time step in a program. A typical parallelism profile is shown in Figure 2. This parallelism profile taken from Loop 11 of the Livermore Loops [2,7] shows the number of parallel operations available for execution at each time step in the program. The parallelism profile also indicates that a variable number of operations are available for execution in parallel throughout the lifetime of the program. The shape of the parallelism curve is characteristic of the form of the source code.

The overall envelope or shape of the parallel profile in Figure 2 is determined by an inner parallel *For* construct, and the number of local maximum values is determined by an outer *while* loop. For a detailed discussion of the Livermore Loops, see [7]. The parallelism profile provides insight into the architectural characteristics of the machine type best suited for executing the program. The maximum number of processors required in order to exploit the parallelism can easily be determined by analysis of the profile graph. The maximum number of parallel operations and hence, the maximum number of processors required for this profile is 250. The area under the parallelism profile represents the total work required to execute this program.

One factor that will affect the execution time of the program is the overhead associated with task creation, completion, data copying, synchronization, etc., as well as resource contentions associated with the initialization and termination of the parallel operations. The effect of this overhead is not apparent by the data displayed in the parallelism profile. The parallelism profile is based on a data dependency ordering of operations by level. The parallelism profile shows the cumulative number of operations at each level, but does not show actual execution times.

3.0. Model Based on Data-Flow Graph

A model can be developed based on the parallelism profile to aid in the understanding of the overhead of creating and executing these parallel operations. The model is based on the following definitions:

Maximal Source Overhead: The overhead associated with starting execution of a new maximal thread.

Maximal Sink Overhead: The overhead associated with terminating a maximal thread.

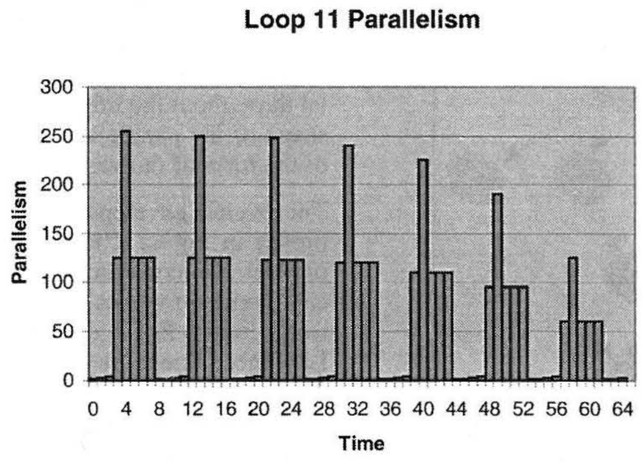


Fig. 2. Parallelism Profile for Livermore Loop 11.

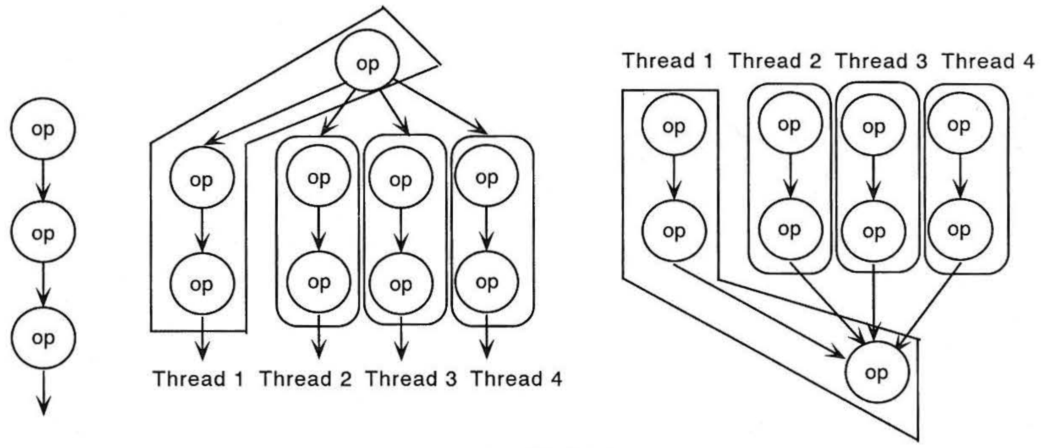

op
Thread 1 Thread 2 Thread 3 Thread 4
Thread 1 Thread 2 Thread 3 Thread 4

Fig. 3. Thread Definitions.

Average Maximal Thread Length: The average number of operations executed in all maximal threads.

The first graph in Figure 3 shows a data dependency graphical representation of a single thread. The data graph provides a strict ordering of operations represented by the data dependencies between the operations in the graph. The single thread shown in Figure 3 contains a single sequential ordering of operations. The second graph in Figure 3 shows three new threads sourced from the top-most node. Overhead will be introduced when these three threads are sourced. This overhead can be attributed to operating system scheduling, resource deallocation and contention, or transfer of data. The third graph in Figure 3 shows three threads that will be terminated by transferring data into the thread that contains the bottom node. Overhead will also be introduced when these three threads are sinked.

It is apparent from the second and third graphs shown in Figure 3 that exactly three threads are sourced, and three are sinked. This level of detail cannot be accurately obtained from a parallelism profile. Each time step shown in a profile shows the net number of parallel operations existing at that level. Consider the parallelism profile shown in Figure 2. The profile shows that approximately 120 parallel operations exist at time steps three, and 250 parallel operations exist at time step four. A net amount of 130 new

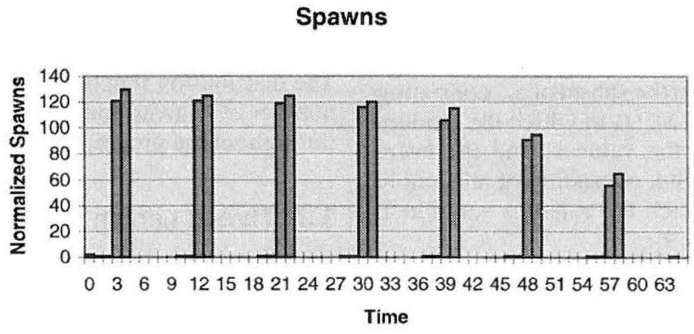


Fig. 4. $S_{rc}(t)$ Graph for Loop 11.

parallel operations were sourced between time steps three and four. However, the profile does not provide enough information to determine if 130 new operations were sourced, or, if the 120 operations in time step three were sinked while 250 new operations were sourced. This information is available from the source program, or the data dependency graph, but not the parallelism profile. The following definitions are required to continue describing the specification of the model based on parallelism profiles.

Let $N(t) \equiv$ number of processors required at time t in the parallelism profile. Enumerate $PE_i = i\ \forall i \in [1, n]$ where n is the maximum number of processors required throughout the program execution. For the parallelism profile illustrated in Figure 1, the value of $n = 250$.

If $N(t) = k$, then $PE_1, PE_2, \dots PE_k$ are executing, and $PE_{k+1} \dots PE_n$ are idle. If $N(t+1) > N(t)$, then we assume processors $PE_{N(t+1)} \dots PE_{N(t)+1}$ initiate execution. Processors $PE_1 \dots PE_{N(t)}$ continue execution, assuming $N(t) \le n$. If $N(t+1) < N(t)$, then we assume processors $PE_{N(t+1)+1} \dots PE_{N(t)}$ terminate execution. Processors $PE_1 \dots PE_{N(t+1)}$ continue execution. A *maximal* process thread is defined to operate on processor PE_k over the time interval $[\alpha, \beta]$, with length $\beta - \alpha$, such that $k \ge N(t)$ $\forall t \in [\alpha, \beta]$.

3.1. Source/Sink Definitions

We define $\Delta N(t)$ as:

$$\Delta_f N(t) = N(t+1) - N(t)$$
$$\Delta_b N(t) = N(t) - N(t-1)$$

where $\Delta_f N(t)$ is a first order forward difference equation, and $\Delta_b N(t)$ is a first order backward difference equation [8]. $\Delta_f N(t)$ represents the net number of maximal threads spawned, and $\Delta_b N(t)$ represents the net number of maximal threads sinked at time t. Define $S_{rc}(t)$ and $S_{nk}(t)$ as:

$$S_{rc}(t) = \frac{1}{2}\left[\left|\Delta_b N(t)\right| + \Delta_b N(t)\right]$$
$$S_{nk}(t) = \frac{1}{2}\left[\left|\Delta_f N(t)\right| - \Delta_f N(t)\right]$$

$S_{rc}(t)$ represents the number of maximal threads sourced, and $S_{nk}(t)$ represents the number of maximal threads sinked at time t. For any program

$$\sum_{t=1}^{\infty} S_{rc}(t) = \#\text{maximallengththreads}$$
$$= \sum_{t=1}^{\infty} S_{nk}(t)$$

The number of maximal length threads sourced must equal the number sinked, otherwise, the program would not terminate. Based on the definitions for $S_{rc}(t)$ and $S_{nk}(t)$, several observations can be made regarding the programs' overhead behavior.

$$0 \le S_{rc}(t) \le \max|\Delta_f N(t)|$$
$$0 \le S_{nk}(t) \le \max|\Delta_b N(t)|$$
$$\max|\Delta_f N(t)|, \max|\Delta_b N(t)| \le \max|N(t)|.$$

The graph shown in Figure 4 illustrates the $S)_{rc}(t)$ curve for the parallelism profile shown in Figure 2.

3.2. Probability Density and Distributions

Define two random variables X and Y. We can define the event A_x to the subset of S_{rc} consisting of all sample points $S_{rc}(t)$ to which the random variable X assigns the value x, and the event B_y to the subset of Snk consisting of all sample points $S_{nk}(t)$ to which the random variable Y assigns the value y [9]:

$$A_x = \{S_{rc}(t) \in S_{rc} \mid X(S_{rc}(t)) = x\}$$
$$B_y = \{S_{nk}(t) \in S_{nk} \mid Y(S_{nk}(t)) = y\}$$

Using these definitions,

$$\begin{aligned} P(A_x) &= P([X=x]) = P(\{S_{rc}(t) | X(S_{rc}(t)) = x\}) \\ &= \sum_{X(S_{rc}(t))=x} P(S_{rc}(t)) \\ P(B_Y) &= P([Y=y]) = P(\{S_{nk}(t) | Y(S_{nk}(t)) = y\}) \\ &= \sum_{Y(S_{nk}(t))=y} P(S_{nk}(t)) \end{aligned}$$

We define these functions as the spawning and sinking probability density functions (pdf), respectively. The following properties hold:

$$0 \leq p(S_{rc}(t)) \leq 1 \quad 0 \leq p(S_{nk}(t)) \leq 1$$
$$\sum_{x \in S_{rc}} P(S_{rc}(t)) = 1 \quad \sum_{y \in S_{nk}} P(S_{nk}(t)) = 1.$$

The cumulative spawning and sinking distribution functions $F_X(x)$ and $F_Y(y)$ as

$$F_X(x) = P\{X < x\} = \sum f_X(x_i)$$
$$F_Y(y) = P\{Y < y\} = \sum f_Y(y_i)$$

The spawning mass function represents the probability of spawning $S_{rc}(t)$ new maximal threads during the execution of the program. The sinking mass function represents the probability of sinking $S_{nk}(t)$ maximal threads during the execution of the program. The spawning and sinking probability mass functions for the parallelism profile shown in Figure 2 are illustrated in Figure 5. Likewise, the distribution functions are depicted in Figure 6.

The cumulative normalized spawning density function shows that threads are spawned fairly uniformly throughout the life of the program. The cumulative normalized sinks density function shows that threads are terminated fairly uniformly throughout the life of the program. The distribution functions show the normalized number of spawns and sinks during execution. The distribution functions in Figure 6 show the number of spawns and sinks are fairly constant throughout the program.

3.3. Thread Length Density / Distributions

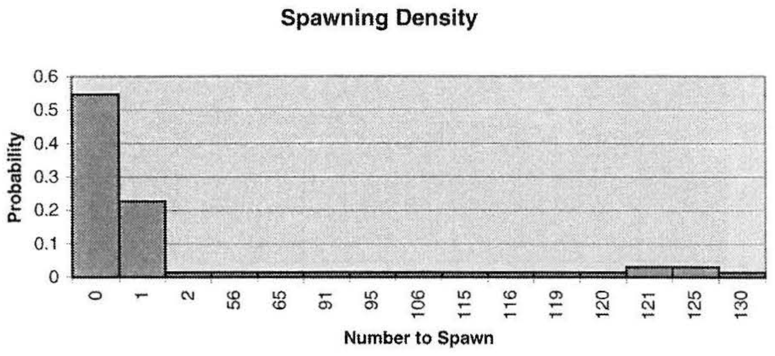
The spawning and sinking density functions provide a technique to model the frequency of maximal thread creation and completion. This provides a measure of how active the program is during execution, and how the overhead of creation and completion is distributed throughout the program. This cost of the overhead can be modeled by density and distribution functions of the length of the maximal threads. For long threads, the overhead cost is easily amortized over the length of the thread. This is typical of MIMD operation, where the length of the thread is long. For short threads, the overhead cost is not readily available, and can represent a significant delay in the thread execution. Short thread lengths are characteristic of SIMD operations.

3.4. Overhead Granularity

The parallelism profile in Figure 2 shows that a large degree of parallelism is available periodically throughout the program. A total of 9000 threads are spawned during the program execution. The average thread length is an important characteristic of the program, and can be determined by:

$$\text{thread length} = \frac{1}{\#\text{threads}} \sum_i t_i$$

where t_i is the length of thread I. The thread length can also be computed by dividing the total area under the parallelism profile curve by the total number of threads. The average thread length for Figure 2 is 3.6. This implies that only 3.6 instructions are executed on average in each thread.



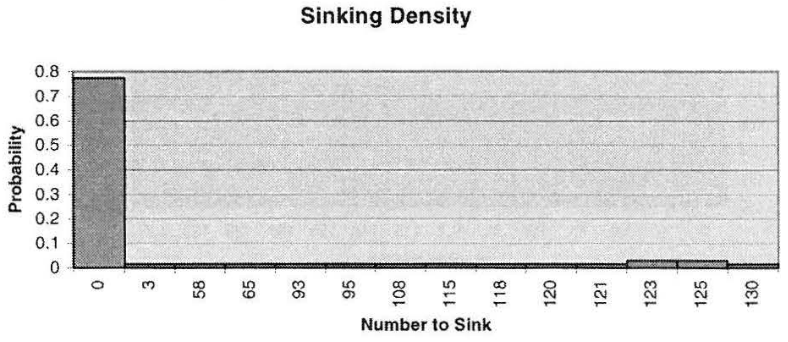


Fig. 5. Density Functions

4.0. Estimation of Processing Elements

To investigate the validity of using spawning and thread length distribution density functions for characterizing exploitable parallelism, a statistical model was developed and run using the *SIMSCRIPT* simulation language [10]. The model consisted of a set of resources representing maximal length threads and two main processes; a GENERATOR and PE process. The GENERATOR process is responsible for randomly determining if and how many maximal length threads are spawned at each CPU clock cycle. The PE process represents a single processing element upon which a maximal length thread will execute. The PE process pseudo-randomly generates the length of a maximal thread by using a user-defined probability density function. Likewise, the GENERATOR process determines the number of maximal length threads to spawn at a given time based upon another user-defined probability density function. The model also has the capability to add additional parameters such as latencies due to processing element overhead, and to limit the number of available processing elements to some finite number.

4.1. Model Validation

The accuracy of the model was tested by performing a series of runs using maximal thread source and length pdfs derived from the Livermore Loops as input. The model results were then compared to the original deterministic parallelism profiles. In order to have a fair comparison, the statistical model assumed that an unlimited number of processing elements with zero communication latency were present. These parameters match the deterministic parallelism curves given in [3,7].

During model validation, it was noted that a crucial parameter for model results is the number of initial threads executing. Upon examination of the deterministic data in [3,7], code segments such as the one represented by loop 10 begin with a small number of threads (less than 10) and at time 1 spawn several thousand threads (over 5500). For the highly parallel scientific

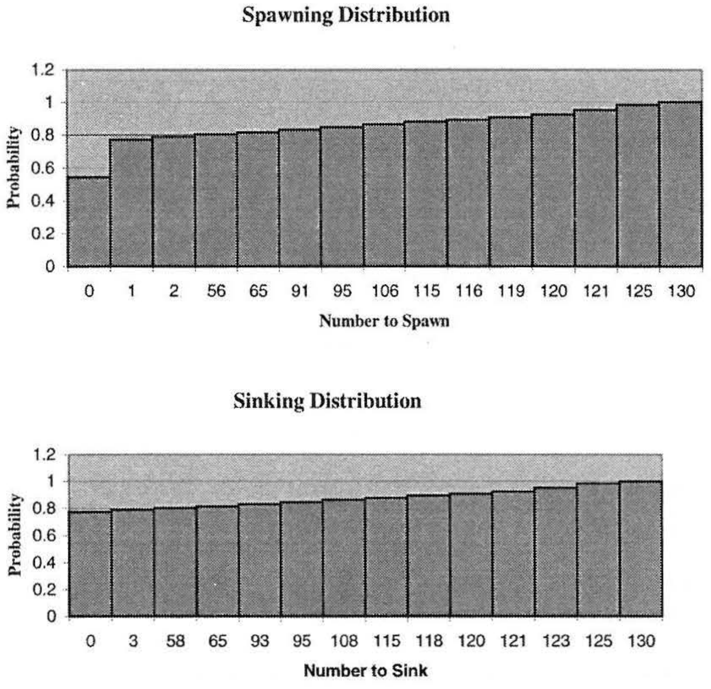


Fig. 6. Distribution Functions

codes used in this study, the initial instruction is typically a "scatter" type command which exploits the concurrency of the code as soon as possible.

4.2. Experimental Results

Table 2 contains the results when the halting criterion is set to equivalent amounts of work in the stochastic simulation and the available parallelism profiles. In roughly half of the benchmark cases the percent error is less than or equal to 5% in terms of resource estimation (required number of processing elements) and is greater than 15% in only 3 of the 15 cases. Since the pdfs may be derived from data dependency graphs as well as available parallelism curves, the model may be used to estimate the required number of processing elements for a given data dependency graph.

5.0. Estimation of Required Memory

Consider the case where all threads in a data dependency graph have the same execution time of one clock cycle and execute on a multithreaded multiprocessor with no delay due to interprocessor communications or synchronization. If this machine also has unlimited resources (i.e. the ideal parallel machine), then all available parallelism in the program can be exploited. Furthermore, the number of threads executing in parallel at each clock cycle will represent the maximum available parallelism in the program. In this simplified case, the data dependency graph may be viewed as having levels of execution, where a level is the collection of nodes executing concurrently during a given clock cycle.

As an example, in Figure 1 the first level would contain node 1 and would execute in one clock cycle. The second level would contain nodes 2 through 5 and would execute in the second clock cycle. Likewise, the third level would contain the nodes 6 and 7 and would execute in the third clock cycle. The fourth and final level would contain only node 8 and would complete execution in the fourth clock cycle. This is summarized in Table 3.

Another useful metric which can be obtained directly from the data dependency graph is the number of graph edges which enter and leave particular nodes on a per level basis. This in-

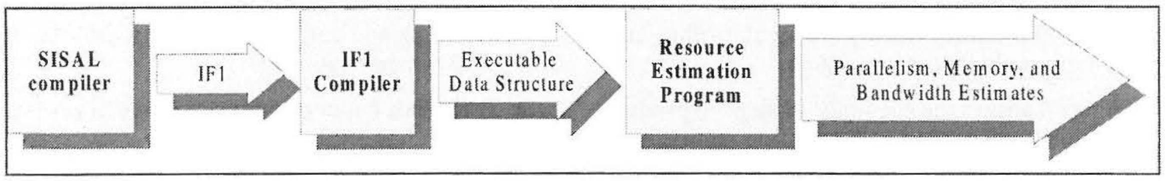

SISAL compiler
IF1
IF1 Compiler
Executable Data Structure
Resource Estimation Program
Parallelism, Memory, and Bandwidth Estimates

Fig. 7. The steps to produce resource estimates from SISAL source code.

Loop	Execution Time			Average Number of PEs			Maximum Number of PEs		
	Model	Actual	% Error	Model	Actual	% Error	Model	Actual	% Error
1	11	8	38%	1740.2	1493.8	16%	3158	2950	7%
2	110	109	1%	22.4	15.0	49%	165	200	18%
3	6	4	50%	1467.1	1250.0	17%	2080	2000	4%
4	22	19	16%	10.1	7.2	40%	24	30	20%
5	107	78	37%	136.4	117.3	16%	486	500	3%
6	967	635	52%	27.0	26.2	31%	168	190	12%
8	30	18	67%	837.8	822.0	2%	2584	2975	13%
9	26	14	86%	255.2	189.5	35%	1103	1000	10%
10	18	12	50%	2718.7	2228.3	22%	6716	5500	22%
11	94	65	45%	78.2	71.5	9%	254	254	0%
12	8	6	33%	1380.6	1000.2	38%	2091	2000	5%
15	35	25	40%	1206.8	1126.0	7%	3173	3290	4%
16	74	45	64%	149.1	156.3	5%	883	900	2%
22	21	12	75%	110.4	116.9	6%	210	200	5%
23	1969	2030	3%	28.1	23.5	19%	776	700	11%

Table 2. Model Validation Results Using the Total Work Halting Criteria

Level	Parallelism	Nodes	Incoming Arcs	Outgoing Arcs
1	1	1	2	4
2	4	2, 3, 4, and 5	4	3
3	2	6 and 7	3	2
4	1	8	3	1

Table 3. The parallelism and incoming and outgoing arc counts for each level for the example data dependency graph

formation can be used to estimate the memory and bandwidth a system requires to efficiently execute a program.

5.1. Memory Requirements Estimation

The amount of required local memory can be estimated for a given processor by noting the maximum amount of intermediate storage used during the execution of a program. However, it is important to note that the actual code does not need to be executed to perform this estimation. The required intermediate storage can be obtained by traversing the data dependency graph structure by application of a "graph walk" algorithm.

Consider the case when a data producing instruction thread completes execution but a corresponding consumer instruction thread requires data from the finished thread as well as another independent producer thread that has not yet completed execution. In this case, the data from the finished producer thread must be stored until the consumer thread has all available data and is scheduled for execution. Based on this premise, we have begun developing a tool to estimate the required memory to execute an algorithm on a generic multiprocessor system.

Figure 7 shows the sequence of steps to produce resource estimates from available source code as we have currently implemented the tool. The first step is to compile the source code into IF1, a text file representing a data dependency graph. A SISAL to IF1 compiler exists [11], as well as IF1 compilers from other high-level languages. The IF1 file is then used as input to the IF1 compiler/profiler described in [3]. The profiler tool extracts the necessary statistics used as input to the memory resource estimation tool.

5.2. Algorithmic Memory Requirements Estimation

We define two types of memory requirements; those due to machine-dependent details of program execution, the *run-time memory requirements*, and those due to the structure of the application program's data dependency graph, the *algorithmic memory requirements*. Algorithmic memory requirements are unavoidable and pertain to the structure of the program only. Run-time memory requirements contain the algorithmic memory requirements in addition to the extra amount of memory needed for processor synchronization, communication, and other operating system needs.

The following method is used to estimate algorithmic memory requirements for a given program in a high level language:

- 1. Compile the source code to IF1.
- 2. Use the IF1 compiler and profiler to produce a parallelism profile that includes incoming and outgoing arc counts (see Table 3 for an example).
- 3. Begin a count of memory usage at zero.
- 4. Step through each level in the parallelism profile, adding the outgoing arcs and subtracting the incoming arcs to find the net memory usage by level. Accumulate these values during each step to determine the current, total memory usage.
- 5. The maximum (peak) value of the accumulated memory usage is then the memory requirement of the algorithm.

5.3. Experimental Results for Memory Estimation Method

Table 4 shows the results of the algorithmic memory requirement estimation tool for the SISAL code shown in Figure 8. After the first level of instruction threads is executed, the number of outgoing arcs that are stored for level 1 is 7,920. Level 2 has only 2,970 incoming arcs leaving 4,950 data items to be stored. The amount of memory required decreases throughout the rest of the execution as those arcs are consumed by other instruction threads, so 4,950 is the peak amount of storage required for execution under these ideal conditions. Therefore, this value represents the algorithmic memory requirements for Livermore Loop 1.

Figure 9 shows the results of the algorithmic memory requirements estimation when analyzing a set of benchmark applications, the Livermore Loops in SISAL [7]. The graph shows only the amount of temporary storage required by the algorithm. The technique does not include memory estimates for the storage of machine instructions and other synchronization

Level	Parallelism	Incoming Arcs	Outgoing Arcs	Memory Usage
1	1	0	7920	***4950***
2	2970	2970	2970	3960
3	1980	3960	1980	1980
4	1980	3960	1980	1980
5	990	1980	990	99
6	990	1980	990	0
7	990	1980	990	0
8	1	990	1	0
9	1	1	0	0

Table 4. Parallelism and algorithmic memory requirements for Livermore Loop 1 when the loop index $N = 990$

```
% LOOP 1
% Hydro Fragment
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for K in 1,n
       X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
   returns array of X
   end for
end function

function Main( rep,n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for i in 1, rep
      X := Loop1( n, Q, R, T, Y, Z );
   returns value of X
   end for
end function
```

Fig. 8. SISAL code for Livermore Loop 1.

and communication overhead. Therefore, these are algorithmic estimates, not runtime memory estimates.

The results of the technique illustrate the relationship between memory usage and the loop bound for the Livermore Loops. In all cases this relationship is approximately linear with respect to the loop bound N. This trend is not surprising since we varied only a single bound. We would expect a non-linear relationship if more than one loop bound were varied. It is interesting to note that the different applications in Figure 9 can be characterized by the slope of the memory usage curves, thus validating the notion of algorithmic memory requirements.

6.0. Conclusions

An approach for multiprocessor processor/memory resource estimation using only an application's data dependency graph was presented. This approach was implemented, leading to the experimental results given. The methodology is suitable for inclusion in a high-level system architecture design package for estimating required processor/memory resources for targeted or benchmark applications. Also, this technique could be incorporated into a "smart" scheduler to utilize available resources efficiently.

The development of a multipurpose resource estimation package has been initiated. To date, a profiler has been developed that produces information containing data structures from an input application's data dependency graph represented in IF1 [3]. A stochastic model based simulator has also been developed, based on the profiling information produced by the IF1 tool for estimating processor element work-loads.

The current version of the memory resource estimation tool is limited to resource estimation for the ideal case of unlimited available processing elements and equal instruction thread length for all graph nodes. This version is being extended to estimate required memory resources for limited processing elements and variation in execution times for each type of node in IF1.

In addition, the arcs between nodes in the data dependency graphs represent data transfers and require bandwidth between individual processing elements. The memory estimation tool will also be extended to estimate the minimum re-

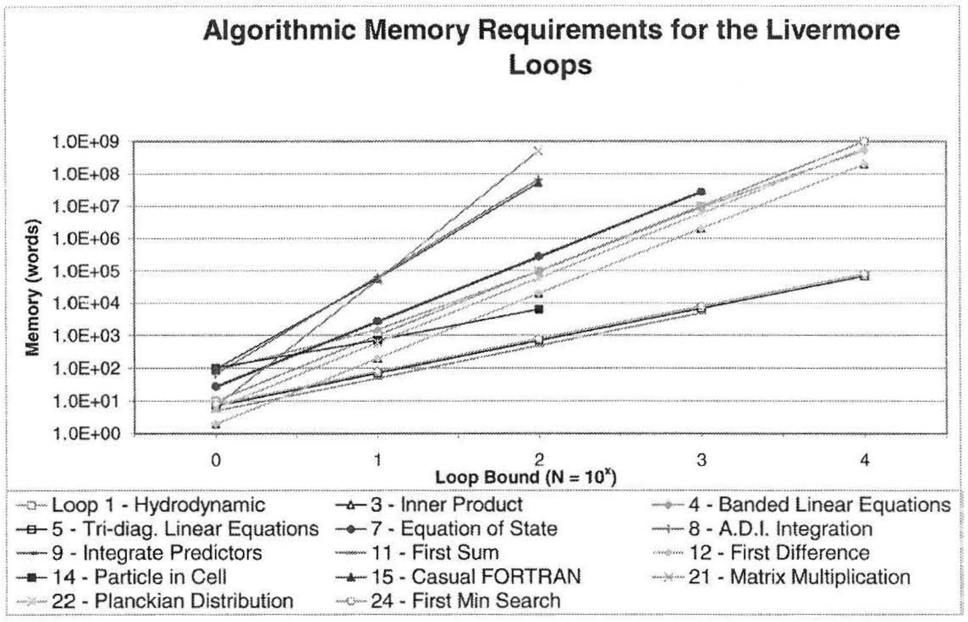


Fig. 9. Algorithmic Memory Requirements for Various Scientific Code Loops and Varying Loop Ranges

quired total system bandwidth to efficiently execute application software.

We have developed a statistical model that can be used to predict needed resources for a parallel architecture based upon the notions of maximal length thread spawning and length probability density functions. This information is easily obtainable from available parallelism profiles or, data dependency graphs. The model was validated through comparisons to actual data and several different halting criteria were evaluated.

The utility of this approach lies in the fact that parameters such as processor latencies and finite resources may be varied and the corresponding characteristics of a parallel architecture may be observed before high level design occurs. Thus, this tool can be valuable for the system designer in the specification phase of the processor architecture. Since the pdfs can be computed directly from a data dependency graph produced by a compiler, this model can be used to predict the required number of processing elements before the program is actually executed.

References

[1] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick, "A Case for Intelligent RAM: IRAM", *IEEE Micro*, April 1997, http://iram.cs.berkeley.edu/publications.html.

[2] John. T. Feo, "An Analysis of the Computational and Parallel Complexity of the Livermore Loops." Elsevier Science Publishers B.V., Series on Parallel Computing 0167–8191/88, #7, 1988.

[3] Suwanto, *Implementation of Compiler, Viewer, and Parallelism Analysis Software for the IF1 Language*, Master of Science thesis at the University of Arkansas at Fayetteville, May 1997, ftp://ftp.engr.uark.edu/user/mat1/pubs/suwanto_thesis.ps.

[4] J. P. Diguet, O. Sentieys, J. L. Philippe, and E. Martin, "Probabilistic Resource Estimation for Pipeline Architecture", in *VLSI Signal Processing VIII*, IEEE Press, October 1995, and *Proceedings of the 1995 IEEE Workshop on Signal Processing*, Osaka, Japan, October 16–18, 1995.

[5] H. Jonkers, A. J. C. van Gemund, G. L. Reijns, "A Probabilistic Approach to Parallel System Performance Modelling", *Proceedings 28^{t}h Annual Hawaii International Conference on System Sciences*, Vol. II (Software Technology), Wailea, Hawaii, January 1995.

[6] R. MORENO, R. HERMIDA, M. FERNANDEZ, "Register Estimation in Unshceduled Dataflow Graphs", ACM Transactions on Design Automation of Electronic Systems, vol. 1, no. 3, pp. 396–403, July 1996, http://www.acm.org/todaes/V1N3/L164/paper.ps.gz.

[7] JOHN T. FEO, "The Livermore Loops in SISAL." Technical Report, UCID–21159, Lawrence Livermore National Laboratory, August 1987.

[8] M. L. JAMES, G. M. SMITH, J. C. WOLFORD, "Applied Numerical Methods For Digital Computation", Harper Row, 2nd ed. 1977.

[9] KISHOR TRIVEDI, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", Prentice Hall, 1982.

[10] E. C. RUSSELL, SIMSCRIPT II.5 Programming Language, 4–th Edition, CACI Products Company, LaJolla, CA, 1987.

[11] J. MCGRAW, S. SKEDZIELEWSKI, R. OLDEHOEFT, J. GLAUERT, C. KIRKHAM, B. NOYCE, R. THOMAS, "SISAL: Streams and Iteration in a Single Assignment Language." *Language Reference Manual, Version 1.2, M–146 Rev. 1*, University of California–Davis, March 1985.

Received: November, 1997
Revised: May, 1998
Accepted: June, 1998

Contact address:
Mitch Thornton
Department of Computer Systems Engineering
University of Arkansas
313 Engineering Hall
Fayetteville, AR 72701–1201
USA
e-mail: mat1@engr.uark.edu
phone: (501) 575-5159
fax: (501) 575-5339

MITCH THORNTON received the BS degree in Electrical Engineering in 1985 from Oklahoma State University, the MS degree in Electrical Engineering in 1991 from the University of Texas at Arlington, the MS degree in Computer Science in 1993 from Southern Methodist University, and the PhD degree in Computer Engineering in 1995 from Southern Methodist University. From 1985 to 1990, he was employed at E-Systems, Inc. and left there as a Sr. Electronic Systems Engineer. In 1995 he was appointed Assistant Professor of Computer Systems Engineering at the University of Arkansas. His research interests include logic synthesis, design verification, and computer arithmetic. Mitch is a member of the computer architecture and CAD/VLSI research group.

DAVID L. ANDREWS is an Associate Professor of Computer Systems Engineering at the University of Arkansas. His research interests are in real time systems, parallel systems, Computer architecture and reconfigurable computing. Dr. Andrews received his BSEE and MSEE from the University Missouri-Columbia, and PhD from Syracuse University. Dr. Andrews is a member of the IEEE.