STAMP: A Universal Algorithmic Model for Next-Generation Multithreaded Machines and Systems

Michel Dubois[1], Hyunyoung Lee[2], and Lan Lin[2]

[1]University of Southern California
Dept. of Electrical Engineering
Los Angeles, CA 90089-2562, USA
dubois@paris.usc.edu

[2]University of Denver
Dept. of Computer Science
Denver, CO 80208, USA
{hlee,llin}@cs.du.edu

Abstract

We propose a generic algorithmic model called STAMP (Synchronous, Transactional, and Asynchronous Multi-Processing) as a universal performance and power complexity model for multithreaded algorithms and systems. We provide examples to illustrate how to design and analyze algorithms using STAMP and how to apply the complexity estimates to better utilize CMP(Chip MultiProcessor)-based machines within given constraints such as power.

1. Introduction

The past 20 years have seen the relentless improvement in processor speed fueled by the exponential explosion of the performance of CMOS technology (Moore's law). In the long term, as feature miniaturization continues unabated, we will reach hard physical limits such as those dictated by quantum effects [16].

Well before this happens, parallel processing is bound to dominate the field of computing across the spectrum of systems because of limits on power. In high-end CMOS microprocessors, power dissipation has reached the limits dictated by their cooling technology and low-end processors such as those used in embedded systems have strict energy limitations, which means that they tend to be optimized for power rather than performance.

It has become clear in the past few years that commercial microprocessor manufacturers are abandoning the route of ever more complex processor cores and higher clock frequencies. For example, Sun's Niagara chip [25] features 8 simple processors with 4 threads each, for a total of 32 threads. The same trends are apparent in IBM's and Intel's processors.

It is expected that, in the future, harnessing the computing power of large scale multithreaded systems will be a major challenge. This challenge is much more complex than in the past because power is now a critical factor in selecting an algorithm for a particular problem and architecture.

To take advantage of multithreaded machines, a framework for algorithms is needed so that researchers in algorithms and systems can invent and create the best possible approaches [26]. Power must be a critical part of the model. Moreover, the model must be general enough to embrace new emerging paradigms such as adaptive and heterogeneous computations and transactional systems and memory [15, 24, 14, 13, 12, 2, 6, 22].

In the past the PRAM model stimulated a lot of research in parallel algorithm design, but performance often did not meet expectation because communication costs were ignored. Recently, more realistic models have been developed such as Queued Shared Memory (QSM) [10] for shared-memory systems, and Bulk Synchronous Parallel (BSP) [27] for message passing systems. However, these models are overly synchronized and restrictive and do not include power models.

The main contribution of this paper is to develop parallel algorithm models for next-generation multithreaded systems and machines. Because we believe that existing parallel algorithmic models are too restrictive, we propose a generic algorithmic model called STAMP (Synchronous, Transactional, and Asynchronous Multi-Processing) as a universal performance and power complexity model for multithreaded algorithms and systems.

The rest of this paper is structured as follows. We overview the background knowledge and literature in archi-

The research by M.D. was supported by NSF grant CNS-0615428. The research by H.L. was supported by NSF grant CNS-0614929 and the University of Denver PROF grant 88197.

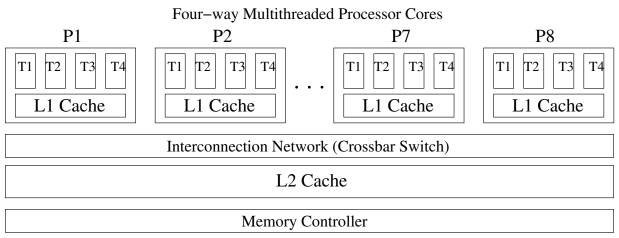
Four−way Multithreaded Processor Cores
P1
P2
P7
P8
T1
T2
T3
T4
L1 Cache
. . .
Interconnection Network (Crossbar Switch)
L2 Cache
Memory Controller

Figure 1. Niagara multiprocessor chip.

tecture and parallel algorithm models in Section 2. Section 3 describes the STAMP algorithm and complexity models. Section 4 illustrates how to map algorithms to the STAMP model through examples. Section 5 concludes the paper.

2. Background

2.1. Architectural Background

Target Architectures. Because of growing technological problems, it will be impossible to maintain the speed improvements observed in the past 40 years in the computing industry unless we unleash the power of parallel processing over large scale multiprocessing systems based on Chip Multiprocessors (CMP), with a large number – hundreds and thousands – of threads. A CMP chip will have several processors and each processor will run several threads (Chip Multi-Threading (CMT)), for a certain maximum number of threads per chip. Hundreds of such chips can then be connected in shared memory or distributed memory configurations. Such CMP/CMT-based architectures and systems are the target of this research. Figure 1 illustrates the overall structure of Sun's Niagara multiprocessor chip.

Controlling Power and Energy. For a given technology and implementation, dynamic power is roughly proportional to f^3 or V^3 where f is the clock frequency and V is the supply voltage and performance is simply proportional to f or V. Thus the easy approach of squeezing more performance by running up the clock frequency has hit a "power wall". Today, parallel processing is the most effective way to improve performance while keeping power under control. To illustrate this consider that 1 processor core clocked at frequency f consumes the same dynamic power as 8 cores, each clocked at $f/2$. Thus if we can get a speedup of more than 2 with the 8 cores, we will get a better performance with the same power [21, 9].

Different algorithms may be preferable in different power/energy environments. Thus a measure of power/energy must be included in the selection of an algorithm for a particular machine. The four classical metrics are D (Delay), PDP (Power-Delay Product), EDP (Energy-Delay Product) and ED^{2}P (Energy-Delay square Product). (Note that Energy is equal to power times delay – E=PD.)

In general, algorithms should be selected according to one of these four metrics (D, PDP, EDP or ED^{2}P), according to the environment where they are deployed.

Transactional Execution Model. Transactional memory is currently the subject of intense research in architecture and systems. Contrary to prior algorithmic models, our STAMP model covers algorithms designed for transactional systems.

The commonly-used method of sharing data via locks allows multithreaded applications to read and write the same data. A new approach, named transactional memory (TM), has been suggested in the early 90's [15, 24] that provides the same capabilities while minimizing the dangers associated with lock-based schemes, such as deadlocks. Lock-based shared memory architectures and TM architectures are both designed to provide safe and error-free access, both read and write, to shared memory data.

At a high level, TM treats data accesses in much the same way as databases do. When accesses are made to critical sections, marked as transactions, each thread acts on its own. When a thread attempts to commit its changes, the TM infrastructure is responsible for determining first if there was a conflict. If so, then it must arbitrate to determine which data is committed and which is rolled back [23, 11].

2.2. Parallel Algorithm Models

A significant body of work exists in the literature to propose and justify various parallel computational models. Maggs et al. [19] promote the use of different models in parallel computation, by contrast with sequential computation. Matias [20] addresses the issue of choosing a suitable model for parallel algorithm design and provides a high level overview of numerous models including the PRAM, the Bulk Synchronous Parallel (BSP), and the Queued Shared Memory (QSM).

The PRAM – the most influential theoretical parallel computational model – allows algorithm designers to concentrate on the inherent parallelism of their algorithms without having to take architectural details into account (e.g. [17]). However, the PRAM model has been much maligned because the run time analysis in the PRAM model is often not a good indicator of an algorithm's run time on any existing parallel machine. This fact has led to the invention of a large number of alternative models. Their aim is to allow accurate performance predictions for existing parallel machines, while still offering a fairly abstract machine view to algorithm designers. Major such models are the BSP model, the LogP model, and the QSM model.

Message-Passing Based Models. The BSP model was first introduced by Valiant [27]. It emphasizes the separation of communication from computation by incorporating bulk-synchrony into a distributed memory model with message passing. In bulk-synchronous computing models, processors compute asynchronously between synchronization barriers [4]. The BSP model has been extended in many directions by introducing additional parameters; examples include the BSP* [3], the E-BSP [18], and the (d, x)-BSP [5] models.

The LogP model was proposed by Culler et al. [8, 7]. The differences between the BSP model and the LogP model are an overhead parameter in the LogP model (which is often ignored) and the omission of required synchronization steps in the LogP model, which allows fully asynchronous behavior of the processors rather than bulk-synchrony. Alexandrov et al. [1] proposed the LogGP model as an extension to the LogP model, by introducing an additional parameter G that represents the bandwidth for long messages.

Shared-Memory Based Models. An example of existing shared-memory based algorithmic models is the QSM model [10]. A QSM algorithm consists of a sequence of *phases*. Two important restrictions distinguish the QSM shared memory from that of a PRAM. The results of shared-memory reads are available only after the end of the current phase (i.e., after the end of the phase in which the read operation is performed). Furthermore, there cannot be both read and write operations to the same shared-memory location within a single phase. Multiple read or multiple write operations within one phase are allowed. These read or write requests are queued and executed sequentially.

3. The STAMP Model

Our goal is to derive a realistic, yet simple, model for parallel algorithms. Our model not only provides a framework for the design and implementation of algorithms, but also provides simple means to estimate execution time and power/energy consumption so that algorithmic approaches can be quickly compared in the context of a multithreaded platform.

In this section, we specify a generic algorithmic model for multithreaded architectures, which we call STAMP (Synchronous, Transactional, and Asynchronous Multi-Processing) and which is equipped with power and performance complexity models. Our STAMP model will be in the general form of parallel, distributed, or nested processes that cooperate with each other.

Local Computation, Communication, and Execution in a STAMP Process. A STAMP process is a direct abstraction of a hardware thread. An execution of a STAMP process is a sequence of local computations and communication operations. A local computation is any operation that can be performed in a single processing unit. A communication operation between two or more processes exchanges information between the processes by message passing or shared memory access.

Distribution of STAMP Processes. STAMP processes can be distributed over different threads within a processor (intra-processor) or over several different processors (inter-processor). We call this attribute of the STAMP model "distribution attribute." The distribution attribute expresses the trade-offs between execution time complexity and power/energy complexity: intra-processor communication is faster than inter-processor communication, however, intra-processor distribution may cause the STAMP processes to exceed the power limit of the processor, so one may be forced to distribute the STAMP processes over different processors. The algorithm designer can designate her STAMP processes in either way by use of the keywords `intra_proc` or `inter_proc` based on the desired synchrony attribute (explained below) and some rough estimates of performance and power/energy.

Structure of STAMP. The smallest unit process in the STAMP model is called an *S-unit*, which is a minimal sequential process, i.e., an S-unit may not be nested or spawn another process. An S-unit consists of a collection of *S-rounds* and any number of local computations. During each S-round, the S-units perform arbitrary local computations. At the end of each S-round, the S-units communicate by sending messages or by writing to the shared memory. At the beginning of each S-round, an S-unit receives messages (by reading from its incoming message queue) or reads the shared memory. Therefore, based on the level of synchrony specified for the algorithm (by the algorithm designer), synchronization can occur either at the end of each S-round or at the beginning of it, or both. A STAMP algorithm can consist of any combinations of S-units, nested STAMPs (by invoking other STAMP processes), or distributed STAMP processes.

Synchrony in STAMP. The synchrony attribute of the STAMP model provides a generic framework for concurrency control in communication and execution of STAMP processes. An execution of a STAMP process can be either transactional or asynchronous, and the communication between STAMP processes can be either synchronous or asynchronous.

Transactional execution, denoted by the keyword `trans_exec`, encompasses the meaning of a traditional transaction such as atomicity, with the additional flexibility of allowing processes to proceed with their executions "optimistically" or "speculatively" and to determine dynamically whether to commit or abort based on the execution results. The `trans_exec` attribute can be associated with the entire process code, any part of the code, or any instruction in the code, which is then executed atomically.

Asynchronous execution, denoted by the keyword `async_exec`, allows each process to proceed with its execution without any restrictions, such as process speed. The async_exec attribute can be associated with the entire code or any part of the code. The async_exec could be used for asynchronous distributed applications or server applications. For instance, asynchronous distributed applications in which replicated servers access a common consistency-critical database (with multiple writers) will be good candidates for async_exec with the synchronous communication mode (explained below). Distributed server applications with single-writer multiple-reader shared memory or database access could use async_exec with the asynchronous communication mode (explained below).

Synchronous communication, denoted by the keyword synch_comm, results in a serialized access to a shared memory or in blocked processes in message passing. This attribute can be associated with the entire code or any particular communication operation in the code. If it is associated with the entire code, every communication operation in the code is assumed to be synchronous.

Asynchronous communication, denoted by the keyword async_comm, allows the communication operation to proceed without any restriction, such as blocking or serialization. However, the algorithm designer should specify some synchronization mechanism explicitly (e.g., a barrier or an acknowledgment) to ensure the consistency of the shared memory or the safe delivery of the messages, whenever necessary. The association rule is similar to the synchronous communication. Table 1 presents all possible combinations of execution and communication modes.

3.1. Complexity Models

The purpose of our model is to use CMP/CMT machines as efficiently as possible. We employ the following set of parameters.

The global parameters are:
P_{a} : the number of intra-processor STAMP processes.
P_{e} : the number of inter-processor STAMP processes.
n : the size of input.

The parameters for local execution are:
c_{fp}, c_{int} : the numbers of floating point operations and integer operations.
w_{fp}, w_{int} : the energy dissipated per floating point operation and integer operation.
c : the time cost for local computation, including floating point and integer operations.

The parameters for communication are of two categories, for message passing and for shared memory access. For shared memory access:
ℓ_{a} : an upper bound on the delay in accessing a shared memory module for intra-processor communication due to the memory hierarchy.
ℓ_{e} : an upper bound on the delay in accessing a shared memory module for inter-processor communication due to the memory hierarchy.
κ : the maximum number of accesses to any shared memory location. In the worst case, it is expressed as the length of serialization or the number of possible rollbacks.
$g_{\mathrm{sh\_a}}$: the *bandwidth* for intra-processor shared memory communication, defined as the ratio of the number of local operations performed by the thread in one time unit to the total number of memory accesses within the processor in the same time unit. This parameter is impacted by, for example, the size of the L1 cache that is used as the shared memory for intra-processor thread communication.
$g_{\mathrm{sh\_e}}$: the *bandwidth* for inter-processor shared memory communication, defined as the ratio of the number of local operations performed by the thread in one time unit to the total number of memory accesses by the processors in the same time unit. This parameter is impacted by, for example, the size of the L2 cache that is used as the shared memory for inter-processor thread communication.
$d_{\mathrm{r\_a}}, d_{\mathrm{w\_a}}, d_{\mathrm{r\_e}}, d_{\mathrm{w\_e}}$: the numbers of shared memory read and write operations for intra- and inter-processor communications.
w_{d_r}, w_{d_w} : the energy dissipated per shared memory read and write operation, where d_r (d_w, resp.) denotes any of intra- and inter-processor shared memory read (write, resp.) operation. We assume that the difference of energy dissipation between intra-processor and inter-processor shared memory operations is negligible.

For message passing:
L_{a} : an upper bound on the *message delay* for intra-processor thread communication, that is, the time between sending and receiving of a message between two threads within a processor.
L_{e} : an upper bound on the *message delay* for inter-processor thread communication, that is, the time between sending and receiving of a message between two threads of two different processors.
$g_{\mathrm{mp\_a}}$: the *bandwidth* for intra-processor thread communication, defined as the ratio of the number of local operations performed by the thread in one time unit to the total number of messages delivered by the processor in the same time unit.
$g_{\mathrm{mp\_e}}$: the *bandwidth* for inter-processor thread communication, defined as the ratio of the number of local operations performed by the thread in one time unit to the total number of messages delivered by the router in the same time unit.
$m_{\mathrm{s\_a}}, m_{\mathrm{r\_a}}, m_{\mathrm{s\_e}}, m_{\mathrm{r\_e}}$: the numbers of message send and receive operations for intra- and inter-processor communications.

Table 1. Possible combinations of modes of execution and communication based on synchrony.

Exec / Comm	Transactional	Asynchronous
Synchronous	transactional exec [trans_exec] synchronous comm [synch_comm]	asynchronous exec [async_exec] synchronous comm [synch_comm]
Asynchronous	transactional exec [trans_exec] asynchronous comm [async_comm]	asynchronous exec [async_exec] asynchronous comm [async_comm]

w_{m_s}, w_{m_r} : the energy dissipated per message send and receive operation, where m_s (m_r, resp.) denotes any of intra-processor and inter-processor message send (receive, resp.) operation. We assume that the difference of energy dissipation between intra- and inter-processor message sending and receiving operations is negligible.

The STAMP complexity model has two facets: execution time (performance) complexity and power/energy complexity. To analyze the execution time complexity, we assume that in one time unit a local operation can be computed by a processing component on data available in memory local to it. Then, for the S-round and S-unit, we add the time needed for local operations, shared memory accesses and message exchanges; for parallel and/or distributed STAMP processes, we take the maximum among the execution times of those STAMP processes.

In the power/energy complexity models we add the energy of each computation, shared memory access and message exchange. We lump local memory accesses with the computation step. This first-order model assumes an architecture, in which functional units are gated off in every cycle if they are not used so that they do not consume dynamic power. While this selective gating may be difficult to achieve in a practical implementation, it is clear that current and future architectures tend to reach that goal to fight the power wall. Moreover this measure gives an algorithmic-based bound on the power dissipated. We summarize the measures of execution times and power/energy:

1. The execution time of an S-round is:

$$\begin{aligned}&T_{\text{S-round}}\\ &\quad = c + [\text{shared memory comm}](\kappa + [P_\text{e} \geq 1]\ell_\text{e} + [P_\text{a} \geq 1]\ell_\text{a}\\ &\quad + g_{\text{sh\_a}}(d_{\text{r\_a}} + d_{\text{w\_a}}) + g_{\text{sh\_e}}(d_{\text{r\_e}} + d_{\text{w\_e}}))\\ &\quad + [\text{message passing comm}]([P_\text{e} \geq 1]L_\text{e} + [P_\text{a} \geq 1]L_\text{a}\\ &\quad + g_{\text{mp\_a}}(m_{\text{s\_a}} + m_{\text{r\_a}}) + g_{\text{mp\_e}}(m_{\text{s\_e}} + m_{\text{r\_e}})),\end{aligned}$$

where the value of the Knuth-Iverson bracket [*cond*] is 1 if the *cond* is true and 0 otherwise. Those terms with the numbers of shared memory read/write ($d_{\text{r\_a}}, d_{\text{r\_e}}, d_{\text{w\_a}}, d_{\text{w\_e}}$) and message send/receive ($m_{\text{s\_a}}, m_{\text{s\_e}}, m_{\text{r\_a}}, m_{\text{r\_e}}$) do not need the conditions on P_e or P_a since, for example, if there is no intra-processor communication, the values of $d_{\text{r\_a}}, d_{\text{w\_a}}, m_{\text{s\_a}}$, and $m_{\text{r\_a}}$ are all zero. The same representation scheme applies to the expression of the total energy needed for an S-round:

$$\begin{aligned}&E_{\text{S-round}}\\ &\quad = c_{fp}w_{fp} + c_{int}w_{int} + w_{d_r}(d_{\text{r\_a}} + d_{\text{r\_e}}) + w_{d_w}(d_{\text{w\_a}}\\ &\qquad + d_{\text{w\_e}}) + w_{m_r}(m_{\text{r\_a}} + m_{\text{r\_e}}) + w_{m_s}(m_{\text{s\_a}} + m_{\text{s\_e}}).\end{aligned}$$

Then the expected power consumption can be obtained by dividing the energy complexity by the execution time complexity: $P_{\text{S-round}} = E_{\text{S-round}} \Big/ T_{\text{S-round}}$.

2. The execution time of an S-unit is the sum of the execution times of all S-rounds and all local computations outside S-rounds, and the energy consumed by an S-unit is the sum of the energy of all S-rounds and all local computations outside S-rounds:

$$\begin{aligned}T_{\text{S-unit}} &= \textstyle\sum_{\text{all S-rounds}} T_{\text{S-round}} + T_c\,;\\ E_{\text{S-unit}} &= \textstyle\sum_{\text{all S-rounds}} E_{\text{S-round}} + E_c\,,\end{aligned}$$

where T_c denotes the execution time of all local computations outside S-rounds and $E_c (= c_{fp} \cdot w_{fp} + c_{int} \cdot w_{int})$ the energy consumed by local computations outside S-rounds. Then the power dissipated in an S-unit is: $P_{\text{S-unit}} = E_{\text{S-unit}} \Big/ T_{\text{S-unit}}$.

3. The execution time of a STAMP process that consists of more than one S-unit (e.g., has iterations) is the sum of the execution times of all S-units, and the energy consumed by such a STAMP process is the sum of the energy of all S-units. The power dissipated in a STAMP process is the energy divided by the execution time of the STAMP process.

4. In general, it is not possible to determine the execution time and the energy/power consumptions of nested STAMPs, however, once given a specific type of problem and a specific class of algorithms, it can be estimated.

5. The execution time of parallel or distributed STAMPs will be the maximum (worst-case) among all the execution times of the parallel or distributed STAMP processes. The energy and the power consumed by parallel or distributed STAMPs will be the total energy and power dissipated by all the parallel or distributed STAMP processes.

With execution time, energy, and power estimates one can check whether the algorithm will meet the power/energy envelope of the target machine. Other metrics related to performance and power as defined in Section 2.1 (D, PDP, EDP, and ED^{2}P) and their complexity can be derived in a straightforward way. This allows a designer to understand the context in which a particular parallel algorithm is well suited (energy limited, workstation/desktop, or server/supercomputer).

4. Examples

Here we illustrate how to map algorithms to the STAMP model through three simple examples: An algorithm for solving a system of linear equations with attributes `intra_proc`, `async_exec` and `synch_comm`, algorithms for banking and airline reservation systems with the `trans_exec` attribute, and an all-pairs shortest-path algorithm with attributes `inter_proc` and `async_comm`.

Solving a system of linear equations. We take the Jacobi algorithm for solving the system $Ax = b$, where x is an unknown vector to be determined, A is an $n \times n$ matrix, and b is a vector in $\mathcal{R}^n$. The Jacobi algorithm starts with some initial vector $x(0) \in \mathcal{R}^n$, and evaluates $x(t), t = 1, 2, \ldots,$ using the iteration $x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j \neq i} a_{ij} x_j(t) - b_i\right]$. A sequential Jacobi algorithm is as follows:

```
Jacobi($A[]$, $b[]$, $x$) [intra_proc, async_exec, synch_comm]
    bool terminated = false; int $t$ = 0
    while not terminated
        forall $i$: $x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j \neq i} a_{ij} x_j(t) - b_i\right]$
        if (termination condition is met) terminated = true
```

Because of the `async_exec` keyword, this algorithm is in the framework of distributed STAMP and will be executed by n threads. However, due to the `intra_proc` keyword, the n threads should be allocated as intra-processor threads as much as possible. The algorithm is translated into a distributed STAMP algorithm in which each process i computes the i-th component of vector x of size n:

```
distributed Jacobi algorithm for process $i$
    bool terminated = false; int $t$ = 0
    while not terminated
        receive $x(t)$ from all other processes
        $x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j \neq i} a_{ij} x_j(t) - b_i\right]$
        send $x_i(t+1)$ to all other processes
        /* implicit barrier synchronization is placed here */
        if (termination condition is met) terminated = true
```

Here, each iteration of the while loop is an S-unit, which consists of a local computation of checking the condition of the while loop, an S-round that consists of receive, local computation, and send, and a local computation of testing for the termination condition and setting the variable terminated if it is true. The send operations in the S-round are followed by a barrier synchronization due to `synch_comm`.

At each round t, a process can proceed to its local computation only after it receives from all other processes, the vector x computed in the previous round. After the local computation is finished, the process sends the resulted vector x_i to all other processes. Assuming that the algorithm uses distributed shared memories (by message passing communication), we compute the following complexity measures. For simplicity, here we do not distinguish between the inter- and intra-processor communications. First we compute the execution time for the S-round:

$$\begin{aligned} T_{\text{S-round}} &= c + L + g(m_s + m_r) \\ &= 2n + L + g(2(n-1)) = 2n + L + 2gn - 2g \end{aligned}$$

The formula for $T_{\text{S-round}}$ (the first equality) is a simplified form of that from Section 3.1 based on the above assumptions. We get the second equality by substituting values for c, m_s, and m_r: there are $n-1$ multiplications, $n-2$ additions and 1 subtraction, and 1 multiplication (outside the brackets), which yield $2n-1$ (floating point) operations. Together with the assignment operation, we get $2n$ local operations. A process receives one message from each of $n-1$ processes and sends one to each of $n-1$ processes, which yields $2(n-1)$ message operations.
The energy for the S-round is calculated in a similar way:

$$\begin{aligned} E_{\text{S-round}} &= w_{fp} c_{fp} + w_{int} c_{int} + w_{m_r} m_r + w_{m_s} m_s \\ &= w_{fp}(2n-1) + w_{int} + w_{m_r}(n-1) + w_{m_s}(n-1) \\ &= (2w_{fp} + w_{m_r} + w_{m_s})n - w_{fp} + w_{int} - w_{m_r} - w_{m_s}. \end{aligned}$$

For the local computations outside the S-round:

$$T_c \geq 2 \quad \text{and} \quad E_c \leq w_{fp} + 2w_{int}\,,$$

where the inequalities are due to the result of the condition checking in the if statement; here we take a lower bound of the execution time and an upper bound of the energy consumption. This is because we want to have an upper bound of power estimates when we divide the energy consumption by the execution time. Now we have the two measures for an S-unit:

$$\begin{aligned} &T_{\text{S-unit}} \geq 2n + L + 2gn - 2g + 2 \quad \text{and} \\ &E_{\text{S-unit}} \leq (2w_{fp} + w_{m_r} + w_{m_s})n + 3w_{int} - w_{m_r} - w_{m_s}. \end{aligned}$$

The above algorithm does not specify the termination condition. A common way of running the Jacobi algorithm would be to execute sufficiently many iterations for the solution vector x to converge close enough to the true solution values. Since we do not have the precise information on how many times the S-unit should iterate, we consider the power consumption of one S-unit, which can be calculated by:

$$\begin{aligned} P_{\text{S-unit}} &\leq E_{\text{S-unit}} / T_{\text{S-unit}} \\ &= \frac{(2w_{fp} + w_{m_r} + w_{m_s})n + 3w_{int} - w_{m_r} - w_{m_s}}{2n + L + 2gn - 2g + 2}. \end{aligned}$$

We elaborate on the complexity measures by considering lower bounds for the parameters L and g. By closely looking at the above algorithm for each process, one can see that the smallest latency value results from having all processes execute in lock steps and the barrier synchronization take no more than unit time. In this case, a message sent by process i to j will be received by process j in the following iteration of j, which requires at least five units of time. The smallest value of the bandwidth factor g can be estimated as the ratio of the smallest number of local computations to the total number of messages delivered by the network within the same amount of units of time, which yields $\frac{3}{n(n-1)}$. Thus, we get a lower bound of

$$\begin{aligned} T_{\text{S-unit}} &\geq 2n + 5 + 2n\frac{3}{n(n-1)} - 2\frac{3}{n(n-1)} + 2 \\ &= 2n + 6/n + 7 \geq 2n. \end{aligned}$$

Now, for example, let us assume the following: $w_{fp} = xw_{int}, w_{m_r} = w_{m_s} = yw_{int}$ for some $x, y \geq 2$. Then we get an upper bound of $E_{\text{S-unit}} \leq (2(x+y)w_{int})n$. Consequently, the power consumption is bounded by

$$P_{\text{S-unit}} \leq \frac{(2(x+y)w_{int})n}{2n} = (x+y)w_{int}.$$

Also, for example, assume that one processor core has the power limit of $3(x+y)w_{int}$ when it is specified in term of w_{int}. Furthermore, assume the same power limit for every processor. Then, the Jacobi algorithm should not be assigned to more than three intra-processor threads per processor, i.e., it can not run on all four threads in one processor, in order to meet the power envelope.

Banking and airline reservation systems. A transfer operation in banking moves a given amount m from one account a to another account b, which is expressed as follows.

```
transfer($a$, $b$, $m$) [intra_proc, trans_exec]
    bool cmit1 = false, cmit2 = false
    cmit1 = $a$.withdraw($m$) [trans_exec, synch_comm]
    cmit2 = $b$.deposit($m$) [trans_exec, synch_comm]
    if (cmit1 $\wedge$ cmit2) then return(true) else return(false)
```

The algorithm transfer is specified as `trans_exec` and has two subtransactions withdraw and deposit, each of which executes atomically. Due to the `intra_proc` attribute together with the `synch_comm` keyword, the two subtransactions should be executed in parallel in intra-processor threads, which can help guarantee the atomic nature of the transfer operation. The transfer transaction can commit only when both subtransactions commit. On the other hand, the airline reservation operation can be specified as follows.

```
reserve(from, to, sect1, sect2) [inter_proc, trans_exec]
    bool cmit1 = false, cmit2 = false, cmit3 = false
    cmit1 = rsrv(from, sect1) [trans_exec, async_comm]
    cmit2 = rsrv(sect1, sect2) [trans_exec, async_comm]
    cmit3 = rsrv(sect2, to) [trans_exec, async_comm]
    if (all three committed) then return(true)
    elseif (none of three committed) then return(false)
    else (the committed leg is not full) then return(true)
```

The `inter_proc` keyword, together with `async_comm`, infers that the subtransactions of reserve can be executed as inter-processor threads. The if statement is an example of a procedure to decide whether the entire transaction should commit or abort when only some of subtransactions commit. This demonstrates the flexibility provided by the optimistic transactional execution together with the use of `async_comm`.

Finding all pairs shortest paths. Finally, we take an all-pairs-shortest-paths (APSP) algorithm to be mapped to the `async_exec` category of the STAMP model with `async_comm` shared memory access and `inter_proc` distribution. The shared vector x to be computed is two-dimensional, n by n, where n is the number of vertices in the graph. Initially each x_{ij} contains the weight of the edge from vertex i to vertex j (if it exists), is 0 if $i = j$, and is infinity otherwise. The algorithm applied to x computes a new vector whose (i, j) entry is $\min_{1 \leq k \leq n}\{x_{ik} + x_{kj}\}$. Each process i is responsible for updating the i-th row vector of x, $1 \leq i \leq n$. The distributed STAMP algorithm for APSP can be expressed as follows:

```
distributed APSP algorithm for process $i$
    bool terminated = false; int $t$ = 0
    while not terminated
        read $x$
        forall $j$: $x_{ij} = \min_{1 \leq k \leq n}\{x_{ik} + x_{kj}\}$
        write $x_i$ /* update the $i$-th row vector of $x$ */
        if (termination condition is met) terminated = true
```

Each iteration of while loop is an S-unit, which consists of a local computation of checking the condition of the while loop, an S-round that consists of shared-memory read, local computation, and shared-memory write, and a local computation of testing for the termination. This asynchronous algorithm does not require any synchronization. This causes no problem for the algorithm because the shared vector x is single-writer and multiple-reader, i.e., each process has its own portion to update in the shared vector. Such algorithms can even take advantage of a faster convergence because faster processors can compute more rounds than slow processors and possibly help the slow processors to be able to terminate after a smaller number of rounds than in a sequential case. Thus, this kind of algorithm can be mapped to a large number of inter-processor threads, even when the processors' available power and processing speeds vary.

5. Conclusions

We proposed a generic algorithmic model called STAMP. Our STAMP model encompasses synchronous, transactional, and asynchronous computation and communication models for multithreaded algorithms and systems and is equipped with a universal performance and power complexity model. By using simple examples, we illustrated how to design and analyze algorithms using STAMP, and furthermore how to apply the complexity estimates to better utilize the CMP/CMT-based machine within given constraints such as the power limit.

By looking at the complexity measures of given algorithms, one can determine if the overall performance can be optimized. For example, reducing inter-processor communication (i.e., assigning the required processes to the threads within one processor) would maximize the performance (i.e., gain higher throughput) within the given power envelope of a single processor or increasing the number of distributed/parallel processes (and assigning them to inter-processor threads) would be needed in order to reduce the local computation and meet the power limit. Our future work includes finding a systematic way of optimizing the overall performance of the multi-threaded machine based on the complexity estimates provided by our STAMP complexity model.

References

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proc. of the 7th ACM SPAA*, pages 95–105, 1995.

[2] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. of HPCA-11*, 2005.

[3] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proc. of European Symposium on Algorithms*, pages 17–30, 1995.

[4] R. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford Univ. Press, 2004.

[5] G. Blelloch, P. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. In *Proc. 7th ACM SPAA*, pages 84–94, 1995.

[6] C. Blundell, E. Lewis, and M. Martin. Deconstructing transactional semantics: The subtlties of atomicity. In *Proc. of WDDD*, June 2005.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39:78–85, 1996.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. of the 4th ACM PPoPP*, pages 1–12, 1993.

[9] M. Flynn and P. Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, May/June 2005.

[10] P. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. 9th ACM SPAA*, pages 72–83, 1997.

[11] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *Proc. of SCOOL*, October 2005.

[12] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of ISCA-31*, June 2004.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd PODC*, pages 92–101, 2003.

[14] M. Herlihy, V. Lunchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd ICDCS*, pages 522–529, 2003.

[15] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of ISCA-20*, pages 289–300, May 1993.

[16] International technology roadmap for semiconductors: Executive summary. `http://public.itrs.net/`, 2003.

[17] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[18] B. Juurlink and H. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Proc. of the 2nd Euro-Par*, volume 11, 1996.

[19] B. Maggs, L. Matheson, and R. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. of the 28th HICSS*, volume II: Software Technology, pages 61–70, Washington D.C., 1994. IEEE Computer Society Press.

[20] Y. Matias. Parallel algorithms column: On the search for suitable models. *SIGACT News*, 28(3):21–29, 1997.

[21] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of ISCA-32*, June 2005.

[23] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of the 24th PODC*, pages 240–248, 2005.

[24] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th PODC*, pages 204–213, July 1995.

[25] Sun's Niagara pours on the cores. *Microprocessor Report*, pages 1–3, September 13, 2004.

[26] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, pages 54–62, September 2005.

[27] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.