Vol.2016-HPC-153 No.9
2016/3/1

経済シミュレーションのためのDebtRankアルゴリズムを用いたグラフ探索コードのスレッド並列化と性能評価

寺井 優晃[1,a)]　藤原 義久[2]　南 一生[1]　庄司 文由[1]

概要：
近年，金融機関や企業をノード，それらの信用関係や取引関係をエッジとしたグラフで表現した生産ネットワークを用いて，特定のノードからネットワーク全体への経済的なストレスの伝搬をシミュレーションすることで，経済システムの状態を定量化する試みが行われている．シミュレーションに用いられるDebtRank アルゴリズムは，最短経路探索問題と同様に，始点ノードから他の全ノードに対して探索を行う．探索に沿って，親ノードから子ノードに対して評価量を伝播していく．探索する始点ごとに独立に処理できるため，比較的簡単にジョブ全体のターンアラウンドタイムを短縮することができる．一方，グラフ構造の特性上，始点ノードごとに計算量は大きく異なるためロードインバランスの原因となる．本研究では，プロセス並列に先立ってタスクが不均一な場合のスレッド並列化手法の検討と性能評価を行った．評価のために，DebtRank アルゴリズムの主要なループに対して OpenMP の omp parallel for/omp task ディレクティブを用いた計 3 パターンのスレッド化を行い，そのオーバヘッドの影響を調査した．さらに評価量の伝播に寄与しないノードを省略することで高速化を試みた．結果，SPARC64 XIfx において 32 コア使用した場合，ランダムに選択したグラフ中の 64 ノードの標本平均で約 17 倍の速度向上を達成した．また，生産ネットワークと類似の性質をもった kronecker グラフによるベンチマークデータを作成し評価を行うことで，今回用いたスレッド並列化手法は，次数分布のべき乗則，スモールワールドの特徴を有する生産ネットワーク以外のグラフに対しても効果があることが分かった．

キーワード：経済シミュレーション, 性能評価, OpenMP, DebtRank, 生産ネットワーク, 複雑系

1. はじめに

現在，国内を対象とした個々の企業の財務諸表や地理的な情報，あるいは債務・債権者間の信用関係や取引関係といった，従来，金融機関が単独で公表していた統計情報と比べて非常に詳細なデータベースを元に，金融機関および企業をノードとし，信用関係や取引関係をエッジとした生産ネットワークと呼ばれる重み付き有向グラフを作成し，このグラフの一部のノードからネットワーク全体への経済的なストレス（評価量）の伝搬をシミュレーションすることで，経済システムの状態を定量化する試みが行われている [1], [2], [3]．系全体の現象を素過程に分解して評価することが困難な複雑系 [4] と呼ばれる領域で，最終的にシミュレーションによって得られた知見により，金融危機や震災等により，ある企業でデフォルトが発生した場合の経済システム全体の影響度を予め計ることができるため，経済政策に示唆を与えることが期待されている．

従来，経済政策のような国家や市場といった大きな括りでの経済問題は，一般的にマクロ経済学と呼ばれる分野が取り扱ってきた歴史的な経緯があるが，昨今，経済に関する膨大なデータベースの蓄積とともに，この分野でも経済シミュレーションが重要視されつつある．本研究ではそのような経済シミュレーションの手法として，Battistonらが提案した DebtRank アルゴリズム [5], [6] を用いる．DebtRank アルゴリズムは，ネットワーク上に評価量を伝搬させる際に，最短経路探索問題と同様に始点ノードから他の全ノードに対して探索を行う．アルゴリズムの特性上，各始点の探索は独立して行うことができるため，生産ネットワークのデータサイズが一般的なサーバに搭載されているメモリ量で足りている場合は，複数の始点ノードを単位として MapReduce のようなフレームワークを用いて分散処理させることで，比較的簡単にジョブ全体のターンアラウンドタイムを小さくすることが期待できる．その一方で，経済シミュレーションの高性能計算の報告例は少な

[1] 理化学研究所 計算科学研究機構
[2] 兵庫県立大学 大学院シミュレーション学研究科
[a)] teraim@riken.jp

く，とくに DebtRank アルゴリズムの性能向上や並列化に関する報告は一切ない．仮に現状の DebtRank の実装系をスレッド並列化せずに単純に分散処理させた場合，プロセス生成のオーバーヘッドや，グラフ構造のデータをタスク間で共有できないことによって，今後扱える問題サイズに制約が生じる可能性がある．また，生産ネットワークは，少数の巨大なハブとなるノード群と，多数の末端に属するノード群から構成され，ノードの次数分布はべき乗則に従うことが既にわかっている．このような複雑ネットワークの持つ不均一なグラフ構造に対する探索は始点ごとの計算量が大きく異なるため．単純に分散処理させた場合，タスク間にロード・インバランスが生じる．この場合，グラフ構造は，アルゴリズムの選択と同程度に性能に影響を与える要因となるが，生産ネットワークの構造が計算性能に与える影響については一切わかっていない．

大規模なデータベースに対する処理が期待されているにも関わらず，近代的なハードウェア資源が有効に利用されているとは言い難い背景があるため，まずは単体性能チューニングの一環として，DebtRank アルゴリズムのスレッド並列化を行う．並列化手法のスケーラビリティの評価をするにあたり，グラフのノード数やエッジ数が異なる複数のデータサイズの入力セットを用意する必要があるが，現状，生産ネットワークはほぼ同じノード数およびエッジ数のデータセット 4 種類に限られ，またグラフデータのサイズを自由に変更することができない．この性能評価を行う上での課題を補うために，生産ネットワークと同様にグラフ構造が不均一な kronecker グラフ [7] と Erdős-Rényi によって提案された random グラフ [8] を任意のノード数とエッジ数で生成し，それらと生産ネットワークを比較することでグラフ構造と性能について議論する．

以上，今後より大規模な経済シミュレーションを行うにあたり，ジョブ全体のターンアラウンドタイムを短くする必要があるが，プロセス並列に先立ってタスクが不均一なグラフのスレッド並列化に取り組み，アルゴリズとグラフ構造の両面から詳細な性能解析を行う必要があると判断した．本稿では，はじめに不均一な構造をもつ 3 種類のグラフについて整理を行い，それらを用いて複数の評価環境で 1 コアを用いた基礎的な評価によりグラフ構造と性能に関する議論を行う．その後，OpenMP によるスレッド並列化を行い，多数のスレッドを生成したときの強スケーリングによる性能評価を報告する．

2. 経済ネットワーク概要

1 つの企業活動に着目した場合，金融機関より運転資金の借入れを行い，その資金を元に仕入先より原材料や中間財を購入し，それに付加価値を与えて製品とし，顧客に販売する．そして，得られた利益を再投資することで，資本を拡大し事業活動を継続する，といった経済活動が一般的に行われている．個々の企業は経営の独立性を保っているが，実際，多くの企業，金融機関との依存関係の上に成り立っており，この連鎖的な経済システムをモデル化したものをここでは経済ネットワークと呼ぶ．さらに経済ネットワークは，金融機関間のネットワーク (inter-bank network)，企業-金融機関間のネットワーク (credit network)，そして企業間のネットワークである生産ネットワーク (production network) から構成されるが，とくに生産ネットワークは，国民経済の状態を知る上で重要な意味を持っている．例えば，マクロ経済の指標において我々に馴染みが深い国民総生産 (GDP) は，財・サービスの付加価値額の総計であるが，それが重視される背景には，実体経済の状態を知る上で極めて重要であるからである．同様に，実体経済において付加的に生産された価値の総量をより詳細に知る上で，生産ネットワークは極めて需要な位置を占める．

生産ネットワークでは，ある企業 A が企業 B と取引関係があった場合，有向グラフ (A $\rightarrow$ B) を用いて表現する．この場合，B は A の販売先であり，A は B の仕入先を意味する．生産ネットワークは，フローの上流にあたる仕入先 (supplier) と，下流である販売先 (customer) の関係性をリンクで表現することから，supplier-customer network とも呼ばれる．ここでは家計は扱わないが，究極的にリンクの下流は消費者とつながっている．このような生産ネットワークを用いてシミュレーションすることで，特定の企業の経営状態が悪化した場合の経済システム全体への影響を定量的に把握することができる．

一方，生産ネットワークを作成するには，企業間の詳細な取引情報が必要となるが，本研究では，東京商工リサーチが有償で提供している国内約 100 万の企業の 2006 年 9 月時点のデータベースを利用した．企業をノード，取引実績に基づいてノード間にエッジを生成することで，約 100 万ノード，約 400 万エッジのグラフが生成される．またグラフでは，企業間の取引量に基づいてリンク間の重みを定義している．なお，データベースには，企業ごとに資本金，従業員数，売上，利益，取引先金融機関・企業，商品，所在地などの詳細な情報が記録され，また総務省が定める日本標準産業分類に基づいて，農業，鉱業，製造業などの 19 のセクタに分類され，さらに，100 の大分類，420 の小分類によって階層的にグループ化されているが，単純化のため，今回の性能評価では分類などの詳細な情報は使用せずに，企業名を匿名化し，グラフの再構成に最低限必要なリンク情報のみ使用した．また，時系列のシミュレーションも可能であるが，ここでは 2006 年 9 月時点でのデータベースのスナップショットを用いた．

3. グラフ構造

3.1 概要

性能評価の観点において，グラフ探索の実行時間は，アルゴリズムだけでなくグラフ構造にも強く影響を受けることが予想される．例えば，次数 k の正則な circular グラフの場合，ノード数が大きくなるとグラフの直径 (グラフの最大頂点間距離) も線形に大きくなるため，生産ネットワークのように大きなノード数に対して小さな直径を持つことはない．一般的にスモールワールドと呼ばれる特徴であるが，ある始点ノードにストレスを与えた場合，生産ネットワークの方が circular グラフより早く伝搬することが十分に考えられる．また単に直径が小さいだけでなく，生産ネットワークは特定の少数のノードがハブとなることがわかっている．ハブが形成されることでノード間の連結に偏りが生じ，始点ノードごとの探索ノード数が異なってくる．結果，グラフ構造が実行時間にも影響を与える．また始点ノードごとの探索ノード数の差は，スレッド並列化した場合にロード・インバランスの原因にもなる．

このような生産ネットワークの特徴は，複雑ネットワークの一般的な性質として知られている．グラフ構造の違いが実行時間に影響を与える可能性が十分ある場合，生産ネットワークに類似した特徴を持つ他のグラフに対しても性能評価を行うことで，今回のスレッド並列化やチューニングの効果が生産ネットワークに限定されたものか，それとも汎用性があるものか明らかにすることができる．そこで本研究では，生産ネットワークと同様に，次数分布がべき乗則に従い，スモールワールド性を持つことが既に知られている kronecker グラフを生成し，それを評価に用いることにする．また，複雑ネットワークが単にランダムなグラフと異なる点を考慮し，比較のために Erdős-Rényi によって提案された random グラフ（以後，単に random グラフ）についても評価を行った．このように人工的にネットワークを生成することで，データサイズを変更させた場合の性能評価を簡単行うことが可能となり，生産ネットワークのデータセットが現状 100 万ノードのセットに限られている問題点についても補うことができる．

なお，ランダムグラフは graph-tool[9] を用いて，circular グラフを元にエッジのつなぎ変え (rewiring) を行うことで生成した．設定はデフォルトとし，この場合，一つのエッジに対するつなぎ変えは一回だけとする．また，同じ 2 つのノード間に並行したエッジ (parallel edge) も生成しない．

kronecker グラフは graph500 の specification で公開されているスクリプトを使用し，初期マトリックスは，$A = 0.5, B = 0.19, C = 0.19, D = 0.05$ とした．なお，graph500 では kronecker グラフの初期マトリクスの要素 B と C を同じにし，対称な隣接行列を生成することで無向グラフとして扱っているが，元になっている kronecker グラフの生成アルゴリズム [10] は有向グラフとして提案されていることと，生産ネットワークの平均入出次数がほぼ同じであるため，本稿ではそのまま使用することにする．

3.2 グラフの特徴

表 1 に, 生産ネットワーク (economic), 生成した kronecker グラフ (kronecker) および random グラフ (random) の特徴を整理した．項目はそれぞれ，ノード数 (#vertices)，エッジ数 (#edges)，平均入出次数 (average #in/out-degree)，平均入出次数の標準偏差 (sd #in/out-degree)，到達可能な 2 ノード間の最頻出距離 (freq. distance)，最大ホップ数 (max #hops)，到達可能な 2 ノード間の組み合わせ総数の 95%以上になるホップ数として有効ホップ数 (effective #hops) を意味する．なお，random グラフと kronecker グラフについては，128 から 1048576 ノードまで 2 倍の刻みで生成したが，ここでは最大ノード数のものを示す．

表 1 評価に用いたグラフの概要

	economic	kronecker	random
type	directed	directed	directed
#vertices	1017510	1048576	1048576
#edges	4506501	4194304	4194304
average #in-degree	4.43	4.00	4.00
sd #in-degree	0.031204	0.042624	0.001952
average #out-degree	4.43	4.00	4.00
sd #out-degree	0.030260	0.042589	0.001953
freq. distance	5	4	10
max #hops	22	11	21
effective #hops	8	5	11

構造が不均一なグラフの特徴の一例として，図 1 にそれぞれのグラフの次数分布を示す．(a),(d) は生産ネットワーク，(b),(e) は kronecker グラフ，(c),(f) は random グラフの入出次数分布を表す．kronecker グラフと random グラフについては，生成時にグラフのノード数と，ノードあたりの平均次数をパラメタとして指定することができる．このノードあたりのエッジ数を edge factor (ef) と呼ぶ．図中凡例の kronecker グラフの ef=4，記載の都合で random グラフの ef=2 が，表 1 に対応し，今回評価に用いた平均次数 4 のグラフに対応する．

生産ネットワークの場合，ノードあたりの平均入出次数は両方とも約 4 でほとんどが 10 以下となっており，1000 を超えるハブとなるノードがわずかに存在していることがわかる．また，次数分布はベキ乗則に従っていることがわかり，入出次数分布の間に大きな違いは確認できない．

kronecker グラフでは，周期的なピークに次数分布が集中しつつ，全体として右下がりな分布になっていることがわかる．若干歪な次数分布をしているが，1000 を超えるハブとなるノードもわずかに存在していることがわかる．

random グラフでは次数分布は Poisson 分布に従うので，ピークを持つことがわかる．edge factor を 1 から 4 に変更することで，ピークの形状が右にずれており，また大きな次数を持つハブが存在しないこともわかる．なお，今回評価に用いたデータサイズでは最大ホップ数が生産ネットワークとほぼ同じ 21 になっているが，edge factor を 1 にすると最大ホップ数が 50 に増加し，反対に edge factor を 4 にすると最大ホップ数は 12 と半分程度になり，グラフの直径がノード数の変化に対して非常に敏感であることを確認した．

4. DebtRank アルゴリズム

生産ネットワーク上のある始点ノードから他のノードに対する評価量（ストレス）の伝播をシミュレートするために，DebtRank アルゴリズムを用いる．DebtRank アルゴリズムは、起点となるノードからエッジによって連結された隣のノードへの評価量の伝播を基本動作とし，これを繰り返すことで，システム全体に波及する影響を計算する．グラフは有向グラフを対象とする．親ノードから子ノードに対して評価量を伝播していく過程で，始点ノードから他の全ノードに対しての探索処理が伴うため，処理内容としては最短経路路探索問題に極めて類似している．ここでは，探索は Breadth-First Search (BFS) に基いたものを用いる．なお，始点は，単一ノードだけでなく，産業セクタを想定した複数のノード群を指定することができるが，以後，簡略化のために始点は 1 ノードのみとする．

ここで，ノード $v_0, \cdots, v_{i-1}, v_i, v_{i+1}, \cdots v_{N-1}$ から成るノードの集合を V とし，$u, v \in V, u \neq v$ なる 2 つのノードを接続するエッジ e_{uv} とエッジの集合 E を $e_{uv} = (u, v) \in E$ とした場合，グラフ全体を $G = (V, E)$ として定義できる．ノード数，エッジ数はそれぞれ $|V|, |E|$ と表現する．なお，有向グラフを扱うため，$(u, v) \neq (v, u)$ である．

エッジの向きについて補足すると，ある企業がストレスを受けるとその仕入先の売上や利益にダメージが生じるため，販売先から仕入先，すなわち下流から上流の企業に影響が伝播すると考えられる．したがって，これ以降は 2 章で説明した生産ネットワークのエッジの向きを反対にしたグラフを用いる．

生産ネットワークでは，属性値として，それぞれの v_i に評価量 $h_i \in [0, 1]$ と訪問管理のためのステータス $s_i \in \{\mathrm{U}, \mathrm{D}, \mathrm{I}\}$ を持ち，またそれぞれの e_{ij} には重さ $w_{ij} \in [0, 1]$ が定義される．ここで，ステータスの定義値 U は Undistressed で未訪問ノード，D は Distressed で探索のフロントとなるノード，I は Inactive で訪問済みで他ノードにストレスを与えないノードを意味する．

始点ノードを $v_{src} \in V$ とした場合，初期値は式 (1), (2) のように与える．

$$h_i = \begin{cases} 1 & \text{if } v_i = v_{src} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$s_i = \begin{cases} \mathsf{D} & \text{if } v_i = v_{src} \\ \mathrm{U} & \text{otherwise} \end{cases} \tag{2}$$

以後，更新前の評価量とステータスを h_i, s_i，一時的に保存される更新後の評価量を h'_i, s'_i とする．式 (3) に従って，ノード v_j が持つ評価量 h_j とエッジ e_{ji} が持つ重み w_{ji} の積が，ノード v_i の評価量 h_i に加えられ，更新後の評価量 h'_i が定まる．ここで h'_i は，1 を超えないように補正される．

$$h'_i = \min(1, h_i + \sum_j w_{ji} h_j) \tag{3}$$

ここで j についての和は，$e_{ji} \in E$ かつ $s_j = \mathsf{D}$ についてとるものとする．すなわち，企業 i の販売先のストレスの重み付き和である．

また，各ノードが持つステータス s_i に関しても式 (4) に従って，探索ステップごとに s'_i に更新される．DebtRank の計算では，ストレスを反復して与えることを避けるため，$U \to D \to I$ のシンプルなステータス遷移のみ許している．

$$s'_i = \begin{cases} \mathsf{D} & \text{if } h'_i > 0 \text{ and } s_i \neq \mathsf{I} \\ \mathsf{I} & \text{if } s_i = \mathsf{D} \\ s_i & \text{otherwise} \end{cases} \tag{4}$$

最終的に，ステータス D となっているノード v_i から，ステータス U となっているノード v_j への $e_{ij} = (v_i, v_j) \in E$ なるエッジが存在しなくなった時点で計算が終了する．計算終了後，始点ノードを除く全ノードの評価量 h_i の平均値が，あるノードを始点とするネットワーク全体へのストレスの代表値 R となる．

今回の評価コードは C 言語を用いて実装した．すべてのノードにユニークな番号を割り当て，隣接行列の代わりにノード番号を添え字にしたポインタ配列を用いて，ノードに接続される入力エッジをリスト構造で管理した．この方法により，本来ノード間の接続を表現する隣接行列が疎になるような場合は，隣接行列をそのまま用いる方法に比べて使用するメモリ量が節約できる．また，OpenMP の omp parallel for ディレクティブでスレッド並列化するに際して，ループ変数は整数型のプリミティブである必要があるが，今回はポインタ配列の添え字をループ変数として使用した．

スレッド化するにあたり，ループ構造を中心としたアルゴリズムの概要を疑似コード (Algorithm 1) に示す．なお，擬似コード中の Index はノードに割り当てたユニークな番号を得るための関数，Weight はエッジに割り当てた重みを得るための関数，IndegreeNodes は引数で与えたノードに入力エッジとして接続されている接続元ノード群を返す

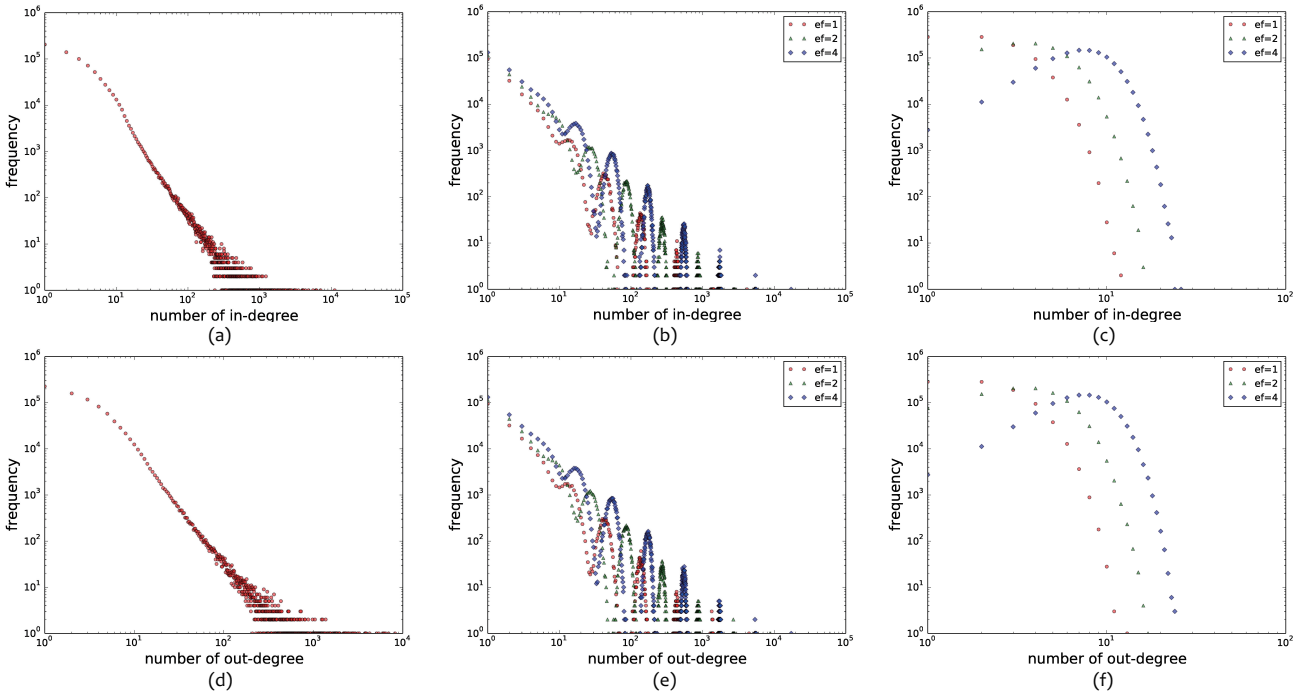

図 1 入出次数分布，生産ネットワーク: (a) 入次数分布 (d) 出次数分布，kronecker グラフ: (b) 入次数分布 (e) 出次数分布，random グラフ: (c) 入次数分布 (f) 出次数分布

関数である．Initialize は式 (1) と式 (2) に従って評価量とステータスの初期値を返す関数，UpdateState は式 (4) に従ってノードのステータスを変更する関数である．

5. 1 コアでの性能評価

5.1 評価方法

はじめに，グラフ探索の性能評価を行う際に注意すべき点について言及する．

一般的に，通常の単体性能評価では，一度測定を行えばその実験条件下での性能値を確定することができる．一方，3 章で述べたような構造が不均一なグラフの場合は，グラフ中の一つの始点ノードにおける性能値を測定したとしても，それがグラフの代表値になっているとは限らない．すべての始点ノードについて測定することで通常の単体性能評価と同様に測定に曖昧さはなくなるが，ノード数が大きくなった場合にプロダクションラン並みの計算リソースが必要になる．これはループや関数単位を対象として，複数のパラメタの組み合わせに対する試行錯誤が伴う単体性能評価には適していない．

そこで今回，graph500 で用いられている評価方法を参考に，グラフ中のすべてのノードから 64 ノードをランダムに選択し，それらを始点ノードとして探索した際の経過時間の最小，最大，平均を評価に用いた．性能比較の際に，探索する始点ノードが異なると計算量そのものが異なってしまい評価が難しくなるので，同じグラフデータについては同じ 64 始点ノードになるように評価を行った．この方法を採用することで，単一始点を対象とした評価に比べてグラフの非均一な構造による影響を考慮でき，またグラフ全体について性能評価を行った場合に比べ，ノード数が極めて大きくなった場合でも現実的な実行時間で性能評価が行える利点がある．

なお，実装したコードでは一度の実行でまとめて複数始点ノードの探索処理することが可能となっており，開始の始点ノードから終了の始点ノードまでの連続したノード番号の値域のことを，ここでは探索幅と呼ぶ．1 コア評価で探索幅は 1 とし，複数コアを用いたスレッド並列の評価ではスレッド数より分割対象の始点ノードの数を大きくする必要性から探索幅は 100 とした．

以後示す結果において，経過時間には Algorithm 1 の処理時間のみ含み，グラフの生成や，その他入出力 I/O は含まない．また今後，単にノードという用語を使用した場合は，断りがない限りグラフのノードのことを指す．

5.2 評価環境

本稿で使用した評価環境のスペックを表 2 に示す．各項目は，動作周波数 (Frequency)，コア数 (#core)，キャッシュサイズ (L1/L2/L3 Cache), Simultaneous Multithreading (SMT)，メモリ規格 (Memory)，最大メモリ転送幅 (Max mem. bandwidth)，コンパイラ種別 (Compiler)，OpenMP のバージョン (OpenMP) である．コンパイラオプションは主なものとして，SPARC64 VIIIfx/XIfx は-

Algorithm 1: Sequential, DebtRank

```
input : G = (V, E), v_src ∈ V
output: R: DebtRank
1  begin
2      N ← |V| − 1; R ← 0; k ← Index(v_src); flag ← false
       // (1) Initializing
3      for i ← 0 to N do
4          h_i, s_i ← Initialize(i, k) // eq.(1) and (2)
5      end
       // (2) traversing connected nodes
6      while flag ≠ true do
           // (3) calculating stress value
7          for i ← 0 to N do
8              h'_i, s'_i ← h_i, s_i
9              V' ← IndegreeNodes(v_i)
10             foreach v ∈ V' do
11                 j ← Index(v)
12                 w ← Weight(e_ji) // e_ji ∈ E
13                 h'_j ← h'_j + w × h_j // eq.(3)
14                 s'_j ← UpdateState(s_j) // eq.(4)
15             end
16         end
           // (4) updating state // eq.(4)
17         for i ← 0 to N do
18             h_i ← h'_i
19             if s_i = D then s_i ← I
20             else s_i ← s'_i
21         end
22         flag ← true
           // (5) loop-exit check
23         for i ← 0 to N do
24             V' ← IndegreeNodes(v_i)
25             foreach v ∈ V' do
26                 j ← Index(v)
27                 if s_j = D then flag ← false
28             end
29         end
30     end
       // (6) reduction to calculate DebtRank
31     for i ← 0 to N do
32         if i ≠ k then R ← R + h_i
33     end
34     R ← R/N
35 end
```

Kfast, POWER7 は-O3 -q64 -qarch=pwr7 -qhot -ipa, Xeon E5-2670v3 は-O3 を用い，スレッド並列についてはそれぞれに OpenMP を有効にするオプションを使用した．なお測定に用いた Xeon E5-2670v3 は，GPU のホストノードに用いられている CPU であり SMT が disable になっているが，1 コア性能評価では影響はないと考え，そのまま測定した．

5.3 生成したグラフのノード数と経過時間の関係

準備として，単一始点（探索幅 1）における，生成したグラフのノード数の増加が経過時間に与える影響を検証する．ノード数 128 から 1048576 ノードまで 2 倍の刻みで生成した kronecker グラフと random グラフのデータを用いて，ノード数と経過時間の関係について SPARC64 VIIIfx の 1 コアを用いて評価したものを図 2 に示す．横軸がノード数，縦軸が経過時間 [s] を表しており，3 つの棒グラフはそれぞれの条件における探索に要した経過時間の最小，最大および平均を表す．random グラフについては経過時間の最小についてはノード数に対して単調増加にはなっていないが，これは不均一なグラフの一部の特性を表しているものと考える．実際，代表値である経過時間の平均についてはノード数に対してほぼ線形な増加傾向を示していることがわかる．以後の評価では，DebtRank アルゴリズムのノード数の増加に対する経過時間への影響はほぼ線形であるとして，kronecker グラフ，random グラフそれぞれで生成したすべてのグラフデータではなく，生産ネットワークのノード数と同等な 1048576 ノードのデータのみを用いて評価を行う．

5.4 結果

各評価環境において 1 コアで実行した場合の結果を図 3 に示す．横軸が評価環境で，縦軸が経過時間 [s] を表す．(a) が生産ネットワーク，(b) が kronecker グラフ，(c) が random グラフであり，図中の評価環境名は，SPARC64 VIIIfx (sparc8)，SPARC64 XIfx (sparc11)，POWER7 (poewr7)，Xeon E5-2670v3 (x86haswell) に対応する．

はじめに SPARC64 VIIIfx を用いた場合のグラフ間の性能比較を行う．結果からは，SPARC64 VIIIfx を用いた場合，3 種類のグラフの中で生産ネットワークが平均 2.03 秒で処理できているのに対して，kronecker グラフで平均 5.00 秒，random グラフについては平均 9.45 秒近くかかっていることがわかる．グラフ特性としては生産ネットワークと kronecker グラフは近いと考えていたが，性能については数倍程度の差があることがわかる．random グラフについては，kronecker グラフよりさらに処理に時間がかかることがわかる．この理由については，より詳細なグラフ構造の調査が必要であるが，大きなハブが存在しない random グラフでは，生産ネットワークよりも評価量の伝播により時間がかかっているものと考えられる．

次に，SPARC64 VIIIfx 以外のアーキテクチャを含めた性能比較を行う．(a) の生産ネットワークは SPARC64 VIIIfx，SPARC64 XIfx と POWER7 アーキテクチャでほぼ同じ程度の平均 2.03 ∼ 2.31 秒で処理されているが，(b) kro-

表 2　評価環境のスペック

	SPARC64 VIIIfx	SPARC64 XIfx	POWER7	Xeon E5-2670v3
#core	8	32	8	12
Frequency [GHz]	2.0	1.975	3.1	2.3
L1 Cache	16KB(I)/16KB(D)	64KB(I)/64KB(D)	32KB(I)/32KB(D)	32KB(I)/32KB(D)
L2 Cache	6MB/chip	24MB/chip	256KB/core	256KB/core
L3 Cache	-	-	32MB/chip	30MB/chip
SMT	-	-	4/core	(SMT disable)
Memory	DDR3	HMC	DDR3	DDR4
Max mem. bandwidth [GB/s]	64	480	136.4	68
Compiler	fcc	fcc	xlc_r7 (v11.1)	icc (v16.0.0)
OpenMP	3.1	3.1	3.0	4.1

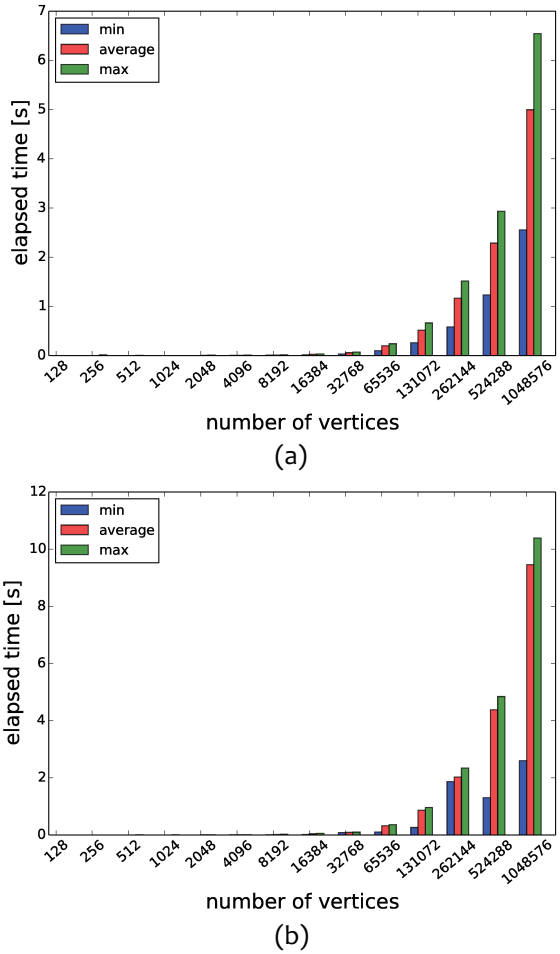

図 2　ノード数に対する経過時間の推移 (すべて SPARC64 VIIIfx, 1core) (a) kronecker グラフ (b) random グラフ

necker グラフと (c) random グラフについては，SPARC64 XIfx と POWER7 は SPARC64 VIIIfx の約 1.8 ∼ 3.3 倍程度の計算時間を要しており，グラフによってアーキテクチャ間で性能が異なることがわかる．この場合，(b) と (c) については SPARC64 VIIIfx と同程度の経過時間になることが期待されるが，実際は数倍程度の差が生じている．比較的類似したアーキテクチャである SPARC64 VIIIfx と SPARC64 XIfx で生じていることを考慮すると，(b) kronecker グラフと (c) random グラフでは性能を劣化させる別の問題が発生していることが推察される．

調査のために，SPARC64 VIIIfx と SPARC64 XIfx のハードウェアカウンタを用いたプロファイリング結果を表 3 と表 4 に示す．結果は，図 3 に示した結果において，最大経過時間となった始点ノードに対するものである．各項目は，経過時間 (elapsed time)，メモリースループット (mem. throughput)，L2 スループット (L2 throughput)，ロードストア数 (#load-store)，ロードストア数あたりの L1D ミス率 (L1D miss rate)，L1D ミス数あたりの L1D ミス dm 率 (L1D miss dm rate)，L1D ミス hwpf 率 (L1D miss hwpf rate)，L1D ミス swpf 率 (L1D miss swpf rate)，ロードストア数あたりの L2 ミス率 (L2 miss rate)，L2 ミス数あたりの L2 ミス dm 率 (L2 miss dm rate)，L2 ミス pf 率 (L2 miss pf rate)，ロードストア数あたりの μ DTLB ミス率 (μDTLB miss rate)，mDTLB ミス率 (mDTLB miss rate) を表す．

表 3 と表 4 を比較すると，メモリスループット，L2 スループットともにほとんど使われていないことがわかる．全体的に L1D ミス率と L2 ミス率は一般的なステンシル・アプリケーションに比べると高めであり，グラフ探索がランダムアクセスに依っていることが推察される．ただし，生産ネットワークについては L1D ミス率が 7.7%程度で，L1D ミス hwpf 率が 17 ∼ 19%であること，また他のグラフよりも L2 miss pf が高めになっていることから，プリフェッチの効果が多少あることがわかる．一方，kronecker グラフと random グラフについては，プリフェッチの効果がほとんどなくなっており，L1D dm ミス率が極めて大きな値を示している．なお，μDTLB ミス率も大きな値を示しているが mDTLB ミス率については十分に小さく，過去に京で行ったチューニングの事例に基づくと，これは性能低下の主要な問題点ではないと考える．

また DebtRank アルゴリズムには式 (3) の計算による浮動小数点演算が含まれるが，基本的には，どのグラフも整数ロードキャッシュアクセス待ち，整数ロードメモリアクセス待ちが実行時間の大部分を占めていることを確認し

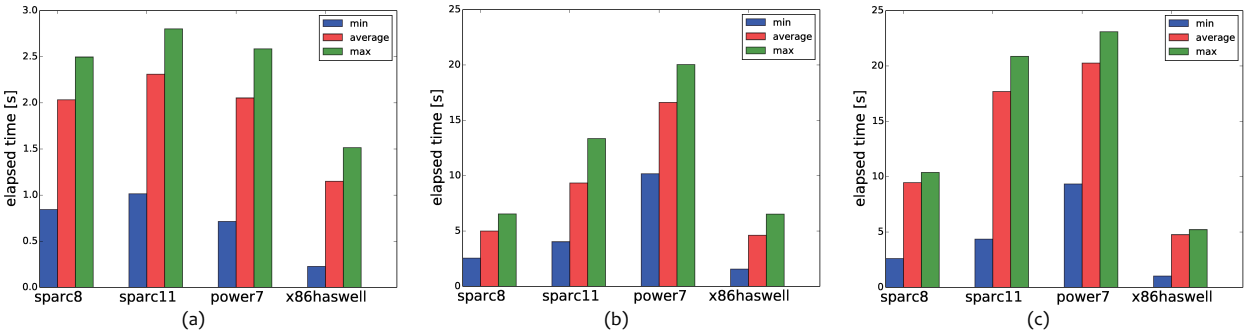

図 3 グラフごとの経過時間比較 (すべて 1core) (a) 生産ネットワーク (1017510 ノード), (b) kronecker グラフ (1048576 ノード), (c) random グラフ (1048576 ノード)

た．プロファイラの結果と合わせると，キャッシュの再利用性が極めて低く，メモリへのデマンドアクセスによってレイテンシが露わになっているものと考える．

表 3 プロファイラ結果 (SPARC64 VIIIfx, 1core)

	economic	keronecker	random
elapsed time[s]	2.50	6.55	10.4
mem. throughput[GB/s]	1.86	2.29	2.10
L2 throughput[GB/s]	4.56	2.77	2.79
#load-store	8.09×10^8	5.81×10^8	8.21×10^8
L1D miss rate[%]	11.0	24.4	27.6
L1D miss dm rate[%]	80.4	98.2	98.6
L1D miss hwpf rate[%]	19.5	1.56	1.21
L1D miss swpf rate[%]	0.14	0.20	0.23
L2 miss rate[%]	3.9	16.8	18.2
L2 miss dm rate[%]	29.7	80.2	82.6
L2 miss pf rate[%]	70.3	19.8	17.4
μDTLB miss rate[%]	1.3×10^{-3}	4.91	10.8
mDTLB miss rate[%]	2×10^{-5}	4×10^{-5}	3×10^{-5}

表 4 プロファイラ結果 (SPARC64 XIfx, 1core)

	economic	kronecker	random
elapsed time[s]	2.88	11.7	16.1
mem. throughput[GB/s]	1.88	1.90	1.92
L2 throughput[GB/s]	5.54	2.71	3.01
#load-store	8.05×10^8	5.77×10^8	8.09×10^8
L1D miss rate[%]	7.70	21.4	23.5
L1D miss dm rate[%]	82.5	99.1	99.0
L1D miss hwpf rate[%]	17.4	0.78	0.72
L1D miss swpf rate[%]	0.11	0.14	0.72
L2 miss rate[%]	2.13	11.9	12.7
L2 miss dm rate[%]	28.4	93.5	95.0
L2 miss pf rate[%]	71.6	6.53	5.03
μDTLB miss rate[%]	1.1×10^{-3}	4.90	11.0
mDTLB miss rate[%]	1×10^{-5}	1×10^{-5}	1×10^{-5}

最後に，命令数の観点で考察する．表には示していないが，SPARC64 XIfx において生産ネットワークの有効総命令数が 2.61×10^9，kronecker グラフが 1.95×10^9，random グラフが 2.60×10^9 となっており，データサイズの設定としてノード数とエッジ数を同程度にすることで，CPU コアに対する処理量としては各グラフ間で対等な比較ができているものと考える．一方で，グラフ間の経過時間に明確な差があり，命令の内訳を含めてより詳細な調査は今後の課題としたい．

6. スレッド並列での性能評価

6.1 概要

今回評価で行った DebtRank アルゴリズムのスレッド並列化の実装方法について述べる．なお，DebtRank アルゴリズムは C の関数で実装され，始点ノードの番号を引数として与える．以後，ここでは DebtRank 関数と呼ぶ．複数の始点に対してまとめて処理する場合は，関数呼出しの外側に探索幅だけ回転するイテレーションを用意する．スレッド並列化の評価においては探索幅を 100 としたので，単純には 1 コアでの性能評価の 100 倍の計算量となる．

以下，OpenMP の omp parallel for ディレクティブを用いた 2 種類のスレッド並列化 (omp1,omp2) と，omp task ディレクティブを用いたスレッド並列化 (omp3) を実装についての詳細を示す．また，スレッド並列化したものの中から最も効果があったものに対して実施したチューニング (omp2.flt3) についても述べる．

6.2 関数内部ループに対するスレッド並列化 (omp1)

4 章で示したように，単一始点の探索処理を行う DebtRank アルゴリズム中にノード数をループ変数とするイテレーション (1),(3),(4),(5),(6) が含まれている．その内，(5) と (6) はリダクション演算が可能である．これらイテレーションに対して，omp prallel for ディレクティブを用いてスレッド並列化を行った．DebtRank 関数内部の各イテレーションが持つノード数分の回転数をスレッドで分割することになるため，parallel region あたりの回転数は次節で示す omp2 より大きくなる．ロード・インバランスの観点では，大きな回転数についてスレッド並列化した方

が処理量の偏りが緩和できる可能性がある．その一方で parallel region の fork-join を繰り返すため，OpenMP のオーバヘッドが大きくなる問題点がある．この実装を以後 omp1 と呼ぶ．

6.3 関数外部ループに対するスレッド並列化 (omp2)

DebtRank アルゴリズムは始点ごとに独立に探索が行えるため，単純に始点ごとにスレッドに分割して処理する方法が考えられる．ここでは関数呼出しの外側にあるループに対して，omp prallel for を用いてスレッド並列化を行った．この実装を以後 omp2 と呼ぶ．今回，探索幅を 100 としたので例えば 32 スレッドで処理する場合，omp2 では 1 スレッドあたり 3,4 始点を探索することになる．なお，DebtRank 関数の内部ではスレッド並列のネストは行わない．

6.4 omp task を用いたスレッド並列化 (omp3)

omp1,omp2 で使用する omp parallel for ディレクティブは比較的簡単にスケーラビリティを出すことができるが，問題点として，適用されるイテレーションの変数が整数型のプリミティブである必要がある．例えば，boost ライブラリを用いてグラフ探索を実装する等，イテレータがオブジェクトであったり，回転数そのものがコンパイル時に判定できないものには適用できない．そのようなイテレーションに対して，OpenMP 3.0 から導入された omp task ディレクティブを用いることでスレッド並列化が可能である．また，タスク並列処理は，ツリー構造を用いた計算，負荷が一様でない計算における有用性が指摘されている [11]．今回の評価では，関数呼出しの外側ループに入る直前で parallel region を生成し，関数呼出しに対して omp task ディレクティブによるスレッド並列化を行った．これも，DebtRank 関数の内部ではスレッド並列のネストは行わず，1 スレッドあたりに処理する始点数は omp2 と同程度となる．この実装を以後 omp3 と呼ぶ．

6.5 チューニング (omp2.flt3)

最後に，今回評価に用いたスレッド並列化実装の中で，最も高速化されたものに対してチューニングを実施した．具体的には，始点ノードに対して入力エッジがない場合と，探索中のフロントノードから出力エッジがない場合について処理を省略するように変更を行った．ここでは omp2 をベースにしたため，この実装を以後 omp2.flt3 と呼ぶ．

6.6 結果

表 2 に示した評価環境の中で，最もコア数が多くスレッドの評価に適していると判断し，はじめに SPARC64 XIfx を用いた場合の各グラフのスレッド並列化による経過時間と速度向上比の推移を図 4 に示す．なお，SPARC64 XIfx の 1 コア評価において μDTLB ミス率の増加による性能低下が確認されているが，ほぼすべての評価環境においてコア数 (またはコア数と SMT の積) のスレッド数まで性能向上する傾向が確認できたため，5 章で明らかになった問題はスレッド並列のスケーラビリティの評価には影響はないと判断した．なお評価では，表 1 に示したデータサイズのグラフを使用した．横軸がスレッド数で，一部の評価環境では SMT の効果を調査するために評価環境の最大コア数を超える 48 スレッドまで測定した．この際，スレッド数の調整は，実行時の環境変数 OMP_NUM_THREADS を用いた．棒グラフは経過時間 [s]，折れ線はそれぞれの条件下でスレッド数を 1 とした場合の平均経過時間による速度向上比を意味する．この評価は，問題サイズを変更せずに並列数を大きくしているので強スケーリングを表している．

図からは，omp1,omp2,omp3 のすべての実装において，スレッド数増加が性能向上に効果があることがわかる．生産ネットワークについて，omp1 を用いて探索した場合，1 スレッド時の経過時間の平均で 201[s] 処理に要しているのに対して，omp2,omp3 ではそれぞれ 124[s] と 126[s] になっていることから，単純に parallel region の fork-join のオーバヘッドの影響がでているものと考える．kronecker グラフ，random グラフについても同様な傾向がわかる．一方で，omp task ディレクティブを用いる omp3 では，オーバヘッドの影響が予想されたが，omp2 と同程度のスケーラビリティが得られていることがわかる．詳細はスペースの都合で省略するが，プロファイル結果からロード・インバランスが omp1,omp2,omp3 の中で最も小さいことを確認した．

3 つのグラフ間で比較すると，実装に関係なく，経過時間の最小，平均，最大について，生産ネットワークが最も小さく，random グラフが大きくなる傾向が確認でき，1 コアの性能評価の結果に準じたものであることがわかる．また，最小と最大経過時間の差は，生産ネットワークが最も大きく，random グラフが最も小さいことがわかる．同じノード数とエッジ数を持ったグラフであっても，random グラフのようにハブを持たないグラフは始点ノードに関わらず平均的に処理に時間を要していると推察され，スレッド並列の評価結果からもグラフ構造が計算時間に影響を与えていることがわかる．

結果はいずれも 32 スレッドまで性能向上しているが，それ以上のスレッド数を生成した場合，速度向上比が一定になっていることがわかる．SPARC64 XIfx は 32 コアで SMT を持たないので妥当な結果であるが，32 コア以上で性能向上が不自然にフラットになってしまうのは，ランタイムライブラリがスレッド数の生成に制限を設けているためであると考える．これについては実行時に，環境変

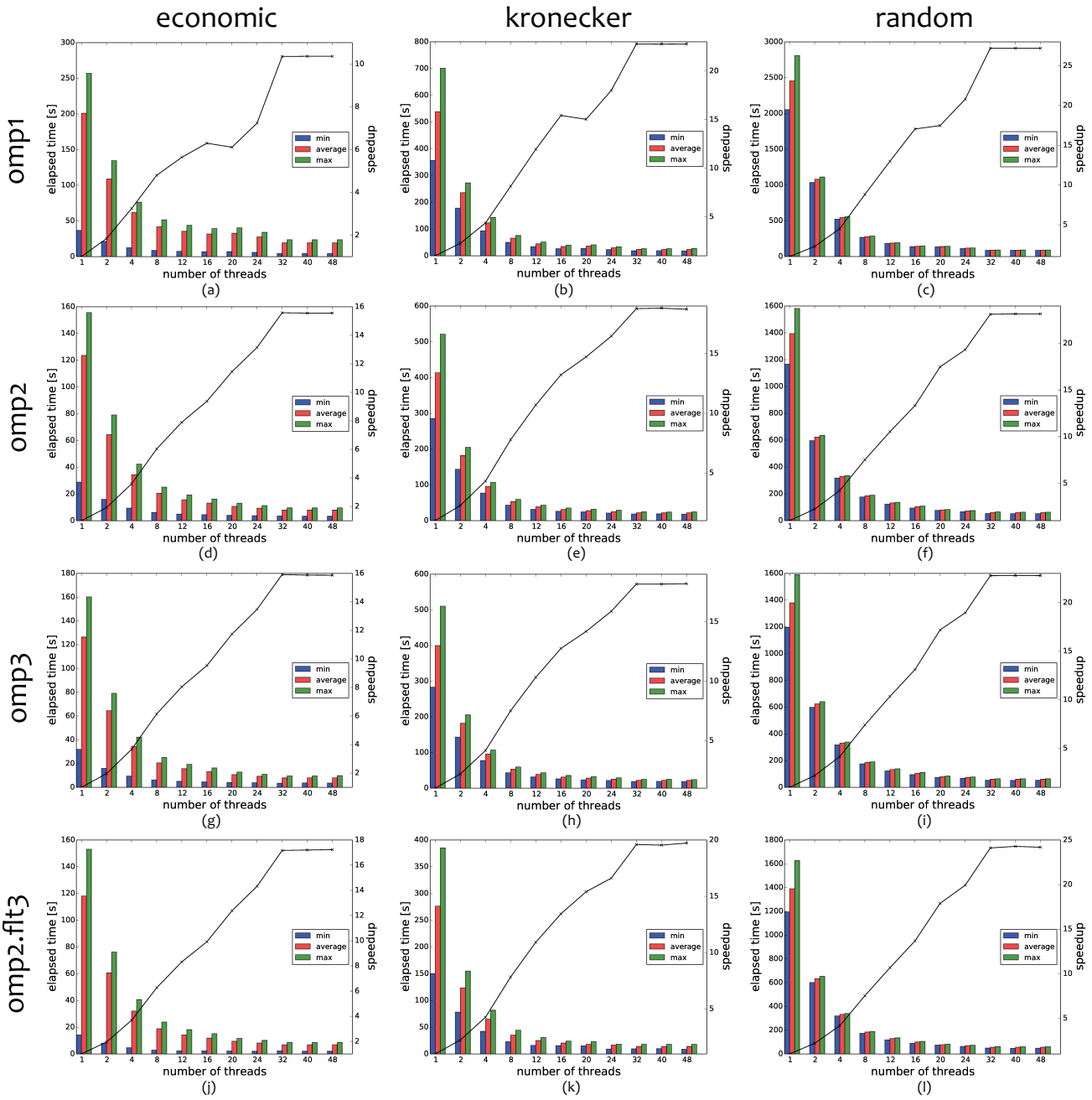

図 4 スレッド並列化による経過時間の速度向上比の推移 (SPARC64 XIfx)

数 FLIB_FASTOMP と OMP_DYNAMIC を FALSE にすることでハードウェアバリアと動的スレッド調整機能が disable になり，スレッド数制限を解除することができるが，その場合でも 32 スレッド以上には性能が向上しないことを確認した．最終的に，SPARC64 XIfx を用いて生産ネットワークについて探索した場合は，omp2 の実装において 1 スレッドで 124[s] かかっていたものが，32 スレッドで 7.93[s] に短縮された．さらに，評価量の伝播に寄与しないエッジの探索を省略するチューニングを行うことで，omp2.flt3 において 1 スレッドで 118[s]，32 スレッドで 6.88[s] に短縮された．

最後に SMT の効果について，図 5 に POWER7 を用いた場合の omp2.flt3 の結果を示す．POWER7 はコアあたり 4SMT となっており，kronecker グラフの場合は 40 スレッドまで，それ以外のグラフでも 32 スレッドまで性能が改善していることがわかる．なお昨今の x86_64 アーキテクチャも 2SMT となっており，POWER7 同様に SMT の評価としては適しているが，先述したように評価環境の Xeon E5-2670v3 については SMT が disable になっており，実際に評価したところコア数と同等の 12 スレッドが速度向上比のピークとなったため省略する．ただし，POWER7 と同様にコアあたり 2SMT が本来のスペックであるので，SMT が有効な場合はすくなくとも 24 スレッドまでは性能が向上するものと予想する．

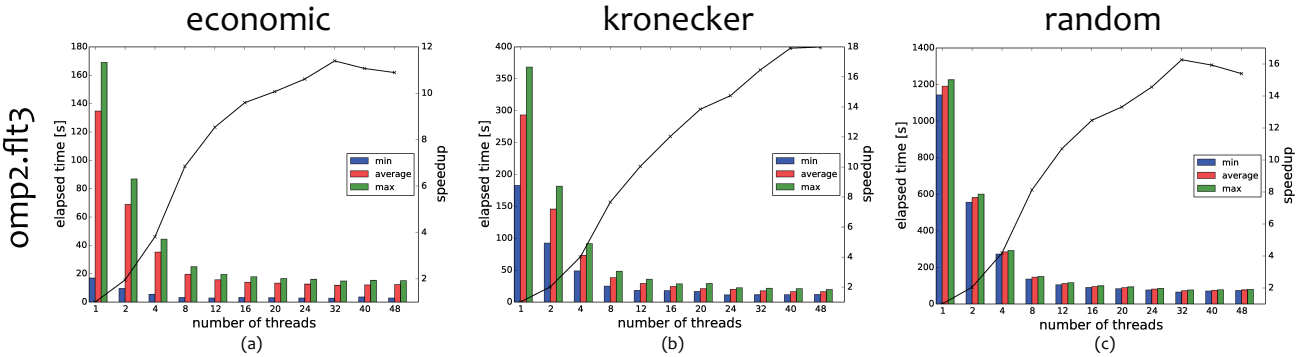

図 5 スレッド並列化による経過時間と速度向上比の推移 (POWER7)

7. まとめ

経済システムの状態を定量化するために，国内企業の取引関係を表現した生産ネットワークに対して，経済的なストレス（評価量）を伝播するシミュレーションが提案されている．現状，シミュレーションに使用される DebtRank アルゴリズムの性能向上や並列化に関する報告はないため，本研究ではプロセス並列に先立って，グラフ探索におけるタスクが不均一な場合のスレッド並列化手法の検討と性能評価を行った．また評価では，データセットが十分に用意されていない生産ネットワークの性能評価を補うために，生産ネットワークと同様にグラフ構造が不均一な kronecker グラフと Erdős-Rényi によって提案された random グラフを生成し，グラフ構造の特徴について整理した．

はじめに，1 コアでの性能評価において，生成した kronecker グラフと random グラフを用いて，探索幅が 1 の場合のノード数と経過時間の平均がほぼ線形になることを確認した．その後，4 つの評価環境を用いて，各グラフの計算時間の特性について評価した．結果，生産ネットワークでは SPARC64 VIIIfx，SPARC64 XIfx，POWER7 の評価環境間で大きな性能差は確認されなかったが，kronecker グラフと random グラフでは著しい性能の低下が確認された．プロファイラからはメモリへのデマンドアクセスが生産ネットワークより支配的になっており，グラフ構造がアルゴリズムと同様に性能に大きな影響を与えることが確かめられた．

次にスレッド並列の評価として，DebtRank アルゴリズムを実装した関数内部にある 5 つのイテレーションに対して OpenMP の omp parallel for ディレクティブを適用し，また，関数外部のイテレーションに対して omp parallel for/omp task ディレクティブの両方を適用した．結果，関数外部のイテレーションに対する omp parallel for ディレクティブによるスレッド並列化が最も性能が向上した．また，omp task もそれと同等な程度の性能向上が確認された．さらに，そのスレッド並列化を行った実装に対して，評価量の伝播に寄与しないノードの探索を省略するチューニングを実施することで．計算時間の短縮を図った．最終的に，生産ネットワークについて SPARC64 XIfx の 32 コア使用した探索を行った場合，ランダムに選択したグラフ中の 64 ノードの標本平均で約 17 倍の速度向上を達成した．また，今回評価のために生成した kronecker グラフと random グラフについても，生産ネットワークと同様に，本研究で用いたスレッド並列化手法の効果が確認できた．

今後の課題としては，一般的なマルチコア・プロセッサだけでなく，メニーコア・プロセッサを用いた評価を行うことが望ましいと考える．またスレッド化手法としても，軽量スレッドライブラリで多用されるワーク・スチーリングなどがロード・インバランスの緩和の効果があるか確認する必要がある．最終的には，プロダクション・ランの際のジョブ全体のターンアラウンドタイムを短くするために，分散フレームワークを利用した評価を行う予定である．

謝辞 本論文の結果の一部は，理化学研究所のスーパーコンピュータ「京」および HOKUSAI-GreatWave システムを利用して得られたものです．また生産ネットワークに関するデータは，独立行政法人経済産業研究所の企業相関データベースを利用しています．

参考文献

[1] Fujiwara, Y. and Aoyama, H.: Large-scale structure of a nation-wide production network, *The European Physical Journal B - Condensed Matter and Complex Systems*, Vol. 77, No. 4, pp. 565–580 (2010).
[2] Aoyama, H., Battiston, S. and Fujiwara, Y.: DebtRank Analysis of the Japanese Credit Network, Vol. RIETI Discussion Paper Seriese 13-E-087 (2013).
[3] Yoshikawa, H., Aoyama, H., Fujiwara, Y. and Iyetomi, H.: Deflation/Inflation Dynamics: Analysis Based on Micro Prices, *SSRN: http://ssrn.com/abstract=2565599* (2015).
[4] Wang, X. F. and Chen, G.: Complex networks: small-world, scale-free and beyond, *Circuits and Systems Magazine, IEEE*, Vol. 3, No. 1, pp. 6–20 (online), DOI: 10.1109/MCAS.2003.1228503 (2003).
[5] S.Battiston, M.Puliga, P.Kaushik, P.Tasca and G.Caldarelli: DebtRank: Too Central to Fail? Financial Networks, the FED and Systemic Risk, *Scientific Reports*, Vol. 1 (2012).
[6] M.Bardoscia, S.Battiston, F.Caccioli and G.Caldarelli:

DebtRank: A microscopic foundation for shock propagation, *PLoS ONE*, Vol. 10(7): e0134888. doi:10.1371/journal.pone.0134888 (2015).
[7] graph500.org: `http://www.graph500.org/`.
[8] P.Erdős and A.Rényi: On random graphs, I., *Publicationes Mathematicae (Debrecen)*, Vol. 6, pp. 290–297 (1959).
[9] Peixoto, T. P.: The graph-tool python library, *figshare*, (online), DOI: 10.6084/m9.figshare.1164194 (2014).
[10] Chakrabarti, D., Zhan, Y. and Faloutsos, C.: R-MAT: A recursive model for graph mining, *In SDM* (2004).
[11] 田浦健次朗, 中島潤：6 種のタスク並列処理系の比較評価，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2013-HPC-140(16), pp. 1–10 (2013).